POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING

ENABLING AND EXPLOITING PROCESS-LEVEL TASK MIGRATION IN OPEN MPI WITH BARBEQUERTRM

Author:
dott. Federico Reghenzani

Student ID (Matricola):
837328

Supervisor (Relatore):
Prof. William Fornaciari

Co-Supervisor (Correlatore):
Ph.D. Giuseppe Massari

A.Y. 2015/2016

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my thesis supervisor *prof. William Fornaciari* and my thesis co-supervisor *Giuseppe Massari* for the time dedicated to guide me on the process of researching and writing this thesis. I would also like to thank *Simone Libutti* and *Gianmario Pozzi* for the help in the development of work proposed in this thesis.

I would like to acknowledge the invaluable support provided by the *CRIU* and *Open MPI* developers for the provided advices on the code integration proposed in this work. I would also to thank the IT4Innovations (IT4I) supercomputing center[1] and its experts for providing us the systems for performing the experimental evaluations.

Finally, I would like to express my deep gratitude to my parents *Manuela* and *Mauro* and my girlfriend *Lara* for providing me unfailing encouragement during these years of study and thesis writing.

This accomplishment would not have been possible without the support of the previously mentioned people. Thank you.

[1] https://www.it4i.cz/

Abstract (Italian version)

I sistemi High Performance Computing (HPC) sono tipicamente caratterizzati da un gran numero di risorse – CPU, GPU, ecc – implicando, di conseguenza, la necessità di affrontare il problema di una loro efficace utilizzazione. In aggiunta a ciò, non è possibile ignorare le strategie di risparmio energetico e dissipazione del calore essenziali in ambiente HPC. Questo quadro è reso ancora più complesso dal fatto che i moderni sistemi sono caratterizzati da un livello decrescente di affidabilità. Per tutte queste ragioni, meccanismi e politiche poco invasive di gestione delle risorse diventano essenziali al fine di risolvere i problemi presentati.

Proprio a causa della loro natura distribuita, i sistemi HPC utilizzano paradigmi di programmazione parallela, tra i quali annoveriamo uno dei più utilizzati: Message Passing Interface (MPI). Questa tesi presenta un'estensione dell'implementazione Open MPI, chiamata `mig`, al fine di supportare in modo trasparente la migrazione di processi di un'applicazione.

Questo meccanismo è integrabile con un gestore delle risorse e in questo lavoro ne viene proposto uno basato su Barbeque Run-Time Resource Manager. Questo approccio ci consente di superare la limitazione di MPI che impedisce la ridefinizione dell'assegnamento delle risorse computazionali a runtime una volta che l'applicazione è stata lanciata. Ciò rappresenta anche una limitazione sia per quanto concerne l'implementazione di strategie di resilienza ai guasti, sia nei riguardi dell'uso efficace delle risorse.

L'estensione `mig` aumenta la flessibilità introducendo una granularità più fine di allocazione del carico di lavoro, attraverso la possibilità di eseguire la *migrazione dei processi*. A tal proposito, in letteratura esistono già molte tecniche di

migrazione, ma mancano principalmente di trasparenza rispetto all'applicazione. In passato in Open MPI esisteva un supporto per la tolleranza ai guasti basato su tecniche di Checkpoint/Restart, ma fu successivamente rimosso a causa della difficile manutenibilità.

In questa tesi proponiamo il framework `mig` come un'estensione di Open MPI per risolvere i problemi precedentemente descritti, in particolare in termini di trasparenza e manutenibilità. L'implementazione di una politica di allocazione delle risorse per BarbequeRTRM è proposta come un possibile caso d'uso del framework.

Abstract

THE High Performance Computing (HPC) systems typically include a large number of computing resources – CPUs, GPUs, etc. As a consequence, we must face with the problem of an effective utilization of them. In addition, we cannot avoid from taking into account power saving and thermal management strategies. The overall picture is made more complex by the fact that modern systems are affected by decreasing level of reliability. For all these reasons, we need effective and poorly invasive resource management mechanisms and policies to address these issues.

Moreover, HPC systems need specific parallel programming paradigms, among these one of the most widespread is Message Passing Interface (MPI). This thesis presents an extension of the Open MPI implementation, called `mig` to transparently support the migration of application processes. This mechanism may be driven by a resource manager and in this work an example of exploitation based on the Barbeque Run-Time Resource Manager is proposed. This approach allows us also to overcome a limitation of the MPI paradigm. In fact, once the application is launched it is no more possible to redefine the assignment of computing resource at run-time. This represents also limitation from the point of view of effective usage of the resources and implementation of fault-tolerance strategies.

The `mig` extension introduces more flexibility by enabling a more fine grained workload allocation, through the possibility of performing *process migration*. In this regard, a lot of migration techniques are already available in literature, but they suffer from the lack of transparency with respect to the application. In the past, Open MPI had fault-tolerance support based on Checkpoint/Restart tech-

niques, but they were subsequently removed due to hard maintainability requirements.

In this thesis we propose the `mig` framework as an Open MPI extension that overcomes the aforementioned issues in terms of transparency and poor maintainability. The implementation of a resource allocation policy for the BarbequeRTRM is also proposed as an example of exploitation of the framework.

CHAPTER 1

Introduction

1.1 The evolution of HPC systems

High Performance Computing (HPC) term refers to computer technologies used in advanced software applications requiring large computing power. The applications are usually parallel in order to run on large clusters of machines, called **supercomputers**.

HPC systems are currently considered one of the most important resources both in research and in industry; the raising of performance-hungry scientific applications lead European Union and other subjects to allocate huge amount of funds to HPC development. HPC is considered strategic for Europe's future and essential for industry to innovate in products and services [1]. However, the research is called to solve several technological limits to the performance scaling, along with addressing the problem of providing guarantees in terms system reliability too, as described in the subsequent paragraphs.

The increasing number of computing nodes, thus CPU cores, the end of Dennard's scaling [2] and the moving towards Exascale computing[1] indeed introduce numerous challenges, in particular regarding thermal and energy optimizations, dependability and resilience concerns, resource allocation scheduling, and par-

[1] see next section for the Exascale definition.

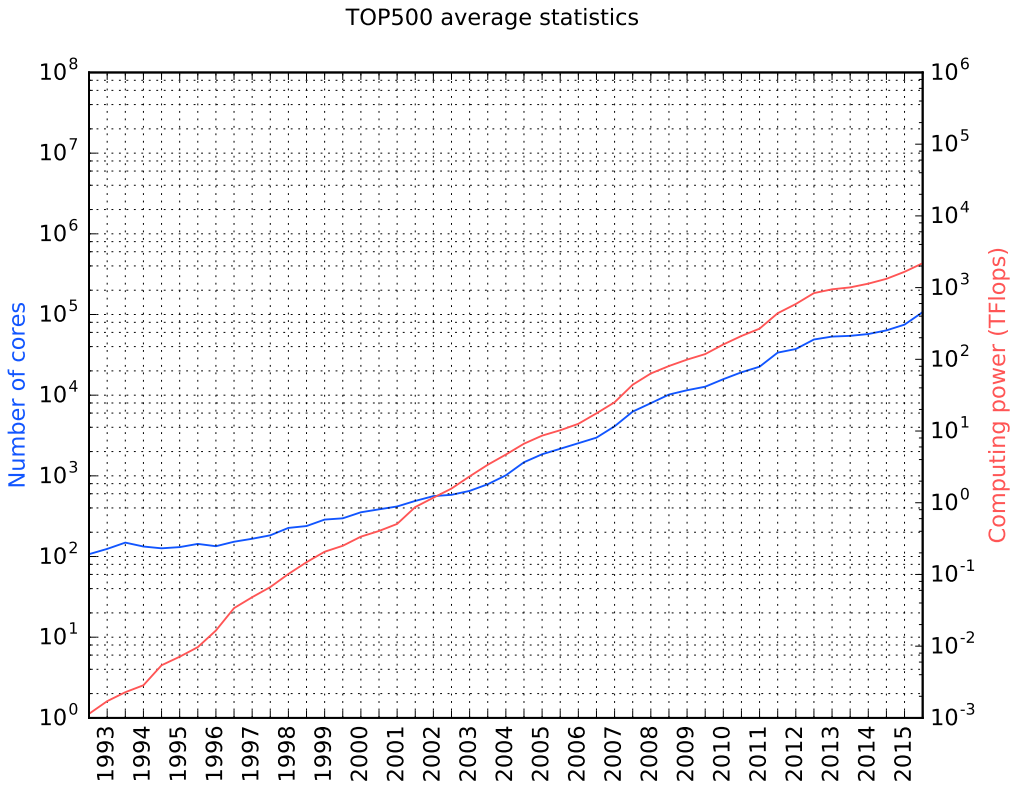

Figure 1.1: *Average number of cores and computing power of TOP-500 supercomputers (TOP500.org data, retrieved 5 August 2016)*

allel programming models [3].

Currently, the main component of HPC infrastructure variable costs is related to thermal and energy considerations. More power consumption means more electricity costs and heat dissipation, more heat dissipation means higher cooling needs and consequently again more electricity costs. Considering the large numbers of servers in HPC clusters, introducing an optimization in a small part of the system may lead to not negligible economical and environmental advantages. In fact, Exascale requires strong efforts in all related fields, from the infrastructure to the software in the direction of increasing power efficiency and programmability.

In last decades the rapid development of processing units maintained acceptable levels of power consumption while increasing the performance delivered. The computational units and computational power trends over past two decades is shown in Figure 1.1 and Figure 1.2.

Unfortunately, the miniaturization of semiconductors cannot go on forever, thus the End of the *Moore's Law* is one of the big concern. In fact, the plateauing of voltage levels and the increasing of leaking current is leading to a power wall

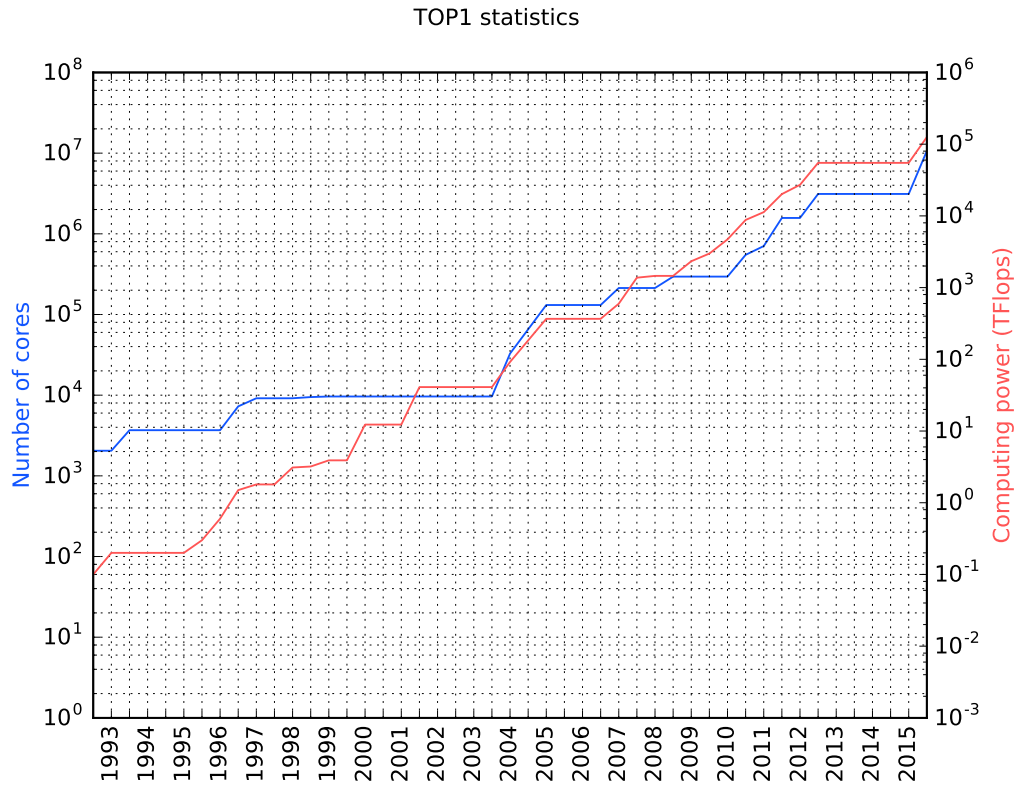

Figure 1.2: *Number of cores and computing power of the TOP-1 supercomputer (TOP500.org data, retrieved 5 August 2016)*

[4]. The reduction of the performance increasing trend, or even the reaching of a performance plateau, would significantly slow down the scientific research [5]. In 2014, the Department of Energy of United States planned to achieve the too ambitious goal of Exascale with 20 MW of power in 2018 [6], but later the deadline was extended.

1.1.1 Exascale as a key goal to reach

Exascale computing refers to systems having a minimum computing power of more than 1 exaFLOPS, i.e. 10^{18} FLOPS.

Research towards Exascale is not limited to the computer science domain, but it strongly affects all the areas of science and engineering. The increasing complexity of mathematical models and the growing size of Big Data requires a growing amount of computing resources.

The Exascale goal is in fact of great interest for a wide range of applications. We can mention several examples, like the study of astrophysical phenomena, weather forecasting, product market simulations, the development of new drugs, the analysis of health risks, etc. [7]. For all these applications, Exascale would

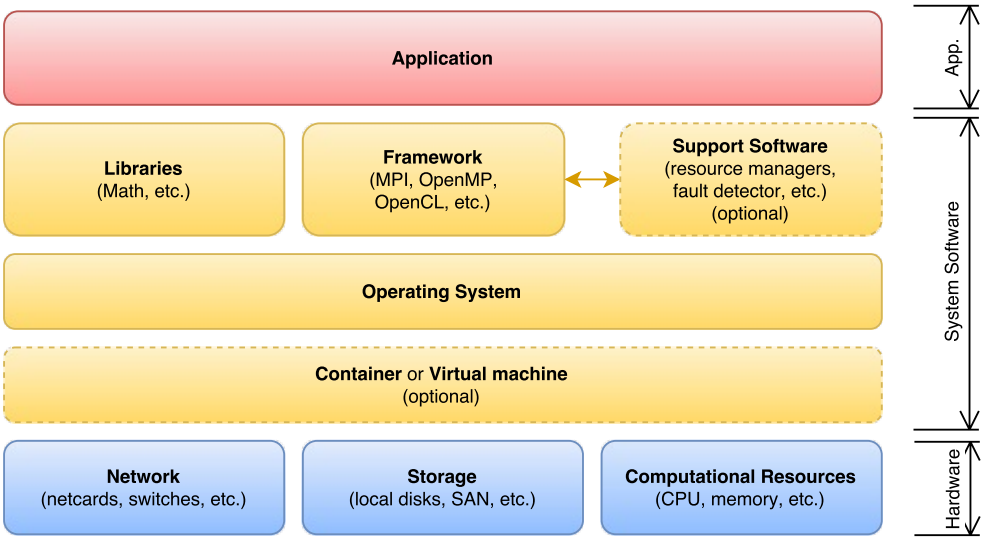

Figure 1.3: *The general architecture of one node in a HPC system.*

enable the possibility of deploying more complex mathematical models, capable of providing much more accurate and reliable results.

Vice versa, other non-computer scientists are studying to find alternative technologies to the current one, in order to mitigate the today issues. Several physicist are studying new types of semiconductors, for instance the very promising research on silicon photonics [8].

From the previous considerations we can state that HPC is still a very hot topic of research and solving the related problematics is not just a matter of computer science, but it will influence and it will be influenced by the research activities in almost all fields.

1.1.2 HPC systems architecture

Since HPC systems require by definition high computational capabilities, the computational resources are typically distributed across different high-end machines (nodes). They are connected via a high speed networks, typically 10Gigabit Fiber Optics Ethernet or InfiniBand.

The storage is also provided through distribution solutions like Storage Area Network (SAN) or Network Attached Storage (NAS). Both solutions have to be designed for HPC environment. In particular, the storage performance and capacity should possibly scale linearly with the numbers of nodes and disks.

The general architecture of a single node in a HPC cluster is shown in Figure 1.3. End-user applications run over a stack of software and, in particular, exploit the API provided by a specific parallel programming framework like MPI or OpenMP. Since the application computational requirements are far behind

the performance capabilities provided by a single CPU, the application must be designed in order to run multiple threads or processes. Parallel frameworks simplify the development of the applications, providing suitable API to manage the execution of multiple tasks, the communication and the synchronization among them.

Furthermore, the application may use other library providing specific functionalities (e.g. math functions) or interact with other software frameworks, like resource managers. The operating system as usual provides to all of software the abstraction of the hardware (virtualized or not).

1.2 Dependability issues in HPC

In HPC, dependability – a broad term including reliability, resilience, fault tolerance, etc. – is another possible future wall in reaching Exascale computing [9]. To highlight the size of the problem is sufficient to say that the Mean Time To Failure (MTTF) for current HPC systems is way below 100 hours [10].

The research community is therefore focusing its effort towards five key directions [11]:

1. statistical and technical characterization of hardware faults;
2. development of a standard fault interface from hardware to software;
3. improving fault prediction, containment, detection, notification and recovery;
4. the development of programming abstractions for resilience, especially fault-tolerant algorithms;
5. proposing fault-tolerant approaches both in hardware and software design.

In this work, we focus on the item 4 proposing a tool that can be used in HPC parallel applications in conjunction with a fault-detection mechanism to support fault-tolerant executions on distributed systems.

1.2.1 Fault-Tolerance requirements and techniques

As already presented, one of the most important discussed theme in HPC is the dependability concern. In particular, since the MTBF is very low compared to the applications timespan, it is required fault-tolerance techniques able to guarantee the termination of the application even if one or more faults occur. In fact, some HPC applications take days or weeks to terminate and after a fault a restart from

the beginning is not acceptable. Therefore, fault-tolerance is no longer a *nice-to-have* feature, but it became a *mandatory* one in HPC systems.

To highlight the previous considerations, a simplification of the Mean Time To Failure (MTTF) and Mean Time Between Failure calculus (MTBF) is proposed, with the objective to provide a trivial qualitative analysis. Assuming the system non repairable, thus MTTF $=$ MTBF, let's consider a cluster of 1.000 CPUs Intel Xeon processor of E7 Family that has MTTF $= 100.000h$ (~11y) [12]. The overall MTTF can be calculated as:

$$\lambda_i = \frac{1}{\text{MTTF}_i} \tag{1.1}$$

$$\lambda_{\text{overall}} = \sum {\lambda_i}^n \tag{1.2}$$

$$\text{MTTF}_{\text{overall}} = \frac{1}{\lambda_{\text{overall}}} \tag{1.3}$$

Applying (1.1), (1.2), (1.3) to our scenario:

$$\text{MTTF}_{\text{overall}} = \frac{1}{1.000 \cdot \frac{1}{100.000}} = 100h$$

The overall MTTF was drastically reduced from 11 years to just few days. Please also note that modern supercomputers have more than 30.000 physical CPUs, leading to MTTF to be less than 4 hours. It is clear that most of the HPC applications, that requires more than few hours to conclude, need a sort of abstract of a fault-free system, in order to execute ideally without being affected by hardware faults. Several approaches have been proposed in literature and industry. The state of the art of this techniques will be discussed in the next chapter.

1.2.2 Failures taxonomy and sources

Following the classification provided by *Snir et al.* [13], the HPC failures may be grouped in three categories: *detected and corrected by the hardware* (DCE), *detected but not corrected by the hardware* (DUE) and *non-detected silent errors* (SE). We do not consider DCE, since they are transparent to the software; for instance, the ECC memory correction is an example of DCE and it is usually performed transparently to the software. We neither deal with SE: the correctness of the result has to be checked by the application since the framework has no tool to infer it.

In the example about the calculation of MTTF, the CPU fault rate was considered. However, other components like memory may be the source of the failure,

further deteriorating the reliability. Regarding **hardware faults**, they can be divided – following the *Snir et al.* classification – in *compute soft*, *compute hard*, *network*, and *I/O* errors. Vice versa, **software faults** can be classified in: *pure software*, *hardware propagating up*, and *software propagating down* errors. This taxonomy is better presented in the subsequent Table 1.1.

Hardware faults	Compute soft errors	Errors caused by transient faults in electronics, e.g. memory corruptions caused by an electromagnetic interference
	Compute hard errors	Permanent fault of a computational component (CPU, RAM, etc.) due to a physical problem, e.g. electromigration.
	Network errors	Total or partial loss of network connectivity, often caused by external component w.r.t. machine, e.g. network apparatus failures.
	I/O errors	Transient or systematic errors during the reading or writing to disks or other storage
Software faults	Pure software errors	The category containing the classical programming issues: correctness errors, concurrency errors, etc.
	HW propagating to SW	A bug in the hardware that propagates up to the software, typical an unmanaged DUE.
	SW propagating to HW	A bug in the software that damages the hardware; it is typical of firmware in embedded appliances.

Table 1.1: *The fault taxonomy in a HPC system according to* Snir et al. *classification.*

The number of possible faults that may lead to a failure in the system and/or in the running HPC applications explains why the thematic of fault tolerance is today not only bound to the embedded world, but it has a fundamental importance also for HPC environments.

1.2.3 Checkpoint/Restart

In response to these faults, most of long-run jobs require a **Checkpoint/Restart (C/R)** mechanism. The C/R paradigm consists in performing periodic *check-*

points, i.e. save the status of the system on non-volatile memory, in order to *restart* the job if a fault occurs. The restart is triggered after the fault is detected and corrected.

Provided the images are saved in a persistent storage[2], this technique guarantees the ability to recover from any type of fault. However, the cost of checkpoint is extremely high and it has to be executed periodically. The overhead can easily reach 50% of the total execution time, reducing considerably the impact on the efficiency of the system [14].

1.2.4 Performance variation and degradation

Performance variation and degradation are increasing problems in semiconductors. The component aging has an important effect in these two problematics.

In HPC, not only faults have to be taken in account: the performance degradation and fluctuation affect also the overall performance of the jobs. Then, it makes important to preserve an acceptable level of aging of all components, through periodical maintenances. Unfortunately, most of operations on machines require to shutdown it; in this direction, migration allows a system administrator to request the freeing of a machine without the necessity to wait the completion of current tasks or freeze entirely the job via C/R.

1.3 Resource management in HPC

The resources management in supercomputers is a prominent challenge. Resource allocation in HPC is a problem studied since 1980s, trying to find a model to allocate jobs in optimal distribution across supercomputers. Albeit the question is old, the increasing of resources spread over several nodes and the diversity of the software require specific policies to schedule and allocate jobs over the cluster. In this regard, we may choose between applying static or dynamic policies. Static policies are in most cases suboptimal or totally inadequate to manage parallel workloads. While dynamic policies allows us to adapt resource allocation decisions to the current workloads characteristics and system status. Moreover, heterogeneous computing is considered by AMD one of the essential capabilities to reach Exascale computing [15].

The goal of this policies may be vary, for instance obtain the maximum performance for certain category of applications or reduce the resource underutilization to minimum, in order to maintain an efficient system. Different applications may have different priorities or they have to generate results (e.g. predictions)

[2] In this context *persistent storage* is intended as fault-free non-volatile memory.

within a given mandatory rate. In this case, it's more important to satisfy the strongest requirement than to have the maximum efficiency.

Most of the large clusters are utilized not only for HPC, but also for Cloud Computing. This mixed environment requires to manage different types of workloads: resource-intensive long-run for HPC applications and a resource flexible adjustment for Cloud. Resource management techniques capable of dealing with mixed workload are consequently required [16] for large computing centers.

In 2016 *Cela et al.* provide an overview [17] of energy-related issues in new exascale HPC applications. Resource management at MPI level is considered essential in order to achieve high scalability. One of the main topics proposed for future research is precisely the migration of MPI processes, that is the main topic of this work. Migration opens up a wide range of possibilities. For instance the resource manager can adapt the computing resource assignment to time varying application performance requirements. Moreover, the application load can be balanced among the system nodes, in order to level down the power consumption and the temperature peaks.

This thesis presents a novel migration technique in MPI and its integration and exploitation in Barbeque RunTime Resource Manager, a resource manager part of the BOSP open source project.

1.4 Message Passing Interface

The **Message-Passing Interface** standard (abbreviated in MPI) is the *de-facto* standard for parallel computing across different nodes. The MPI Forum – composed by both academic and industrial people – released the first version of this standard in 1994 and the last version (3.1) was released in 2015 [18].

The MPI standard describes the syntax and the semantics of function calls in the Application Program Interface (API) provided by a MPI library to the user software. This API allows the communication, the management and the coordination between processes of a parallel application. MPI is mainly used for distributed computing, despite in theory it can be used for the execution on local machine only. In the latter case, *OpenMP* is usually preferred, since it is optimized specifically for single node executions. To reach better performance, most applications are implemented combining the usage of MPI and OpenMP together.

The MPI standard is written through a language-independent specification, in order to be implemented in any programming languages. The most common languages are C, C++ and Fortran.

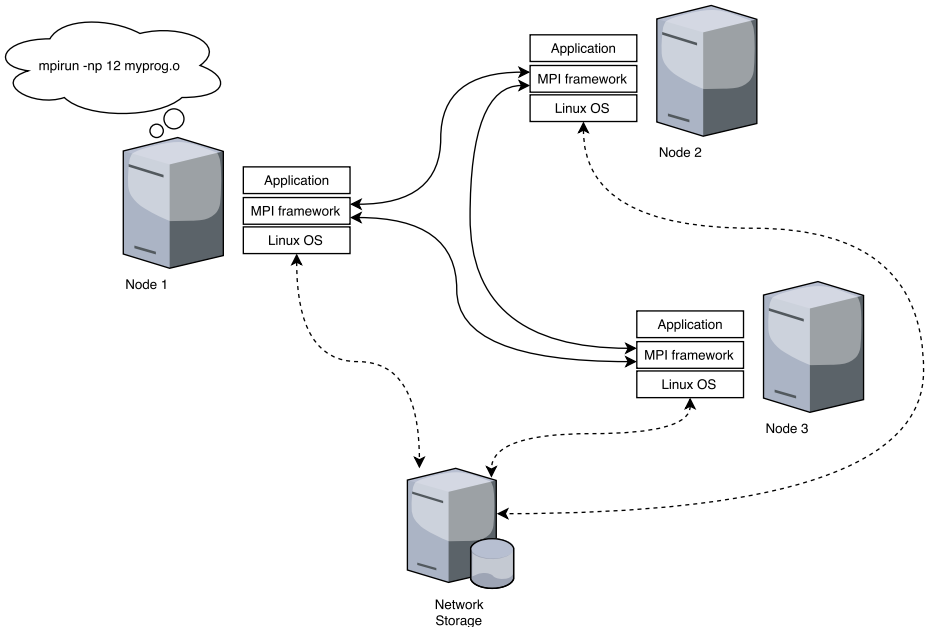

Figure 1.4: *A typical configuration of MPI systems: the user launches the* `mpirun` *command on Node 1 that distributes the computation also over the Node 2 and Node 3. All nodes have access to a common network storage to get the program and the data.*

1.4.1 MPI Application Execution Flow

A MPI typical MPI execution is presented in Figure 1.4. The user launches the job via the `mpirun` command, then the MPI framework spawns the processes in the other available machines (how this is performed is implementation dependent).

The application must be linked with the static or dynamic library of MPI, that provides `MPI_*` function calls. Usually every program starts with `MPI_Init` and ends with `MPI_Finalize`. Both functions are implementation-dependent, but in general *MPI_Init* performs some setup required before any other MPI routines and the `MPI_Finalize` coordinates the execution conclusion freeing the allocated resources.

The communication between processes is divided in two categories: *point-to-point* (direct communication between two processes) and *collective* (communication one-to-many or many-to-many). The latter is performed through the coordination of the various MPI frameworks on all nodes. A typical example of collective feature is the *barrier*: every program should synchronize at the same point before continuing.

Having a common set of functions allows to perform easily and equivalently benchmarks of the same application on different MPI frameworks or to port the

program on another framework without refactoring the code.

1.4.2 MPI implementations

Today, several implementations of MPI are available, among which we cite the two most commonly used: MPICH [19] and Open MPI [20]. They are both open source and have reached a good level of stability even with a considerable number of features, thanks to the very active communities and the large funding of big companies and universities.

The history of MPICH can be split in two part: MPICH-1 and MPICH-2. The development of the latter (at a later time called simply MPICH) starts in 2001 to add the feature of MPI version 2 and subsequently version 3 to MPICH-1.

Open MPI represents the union of three previous implementations: LA-MPI, FT-MPI and LAM/MPI. The third was extensively used in research literature. All of which ceased their development shortly after the begin of Open MPI project in 2003.

The two implementation mainly differs in the purpose of application: MPICH is a very stable basis and standard reference for the development of special purpose needs. Open MPI targets more general cases and it already offers several pre-implemented features, e.g. different types of network communication channels and topology.

Since our framework is supposed to be a general tool that tries to address the issues of the most of HPC applications, we selected the Open MPI implementation to develop the feature presented in next sections. Furthermore, the extreme high modularity of Open MPI internal code was a big advantage in the development of `mig` framework[3].

1.5 Migration of MPI processes

This thesis presents a technique implemented in Open MPI able to allow the migration of MPI processes among different nodes of the cluster. Moving a process across different machines is not a straightforward task and it constitutes a specific research topic. This work exploits existing process migration tools applying them to Open MPI for HPC applications.

Migration of processes can be exploited to solve or mitigate some of the previous presented issues. Obviously, the migration is not for free and introduces an overhead that must be taken in account, while being triggered by an appropriate software, for instance by a fault detector or a resource manager.

[3]Note that the keyword ‘framework‘ may generate misunderstanding: as explained in Chapter 4, the `mig` framework is a module of Open MPI, not a new MPI framework

Migration is particularly interesting for resource management: current run-time resource managers in HPC are limited to assign resources during the application startup, therefore they are not usually able to reschedule the application over different nodes. Migration adds the capability of rescheduling to resource managers, that they have to take in account the significant overhead of moving processes between two nodes.

In large clusters the topology and consequently the location of the processes of the same job significantly impacts on overall performance; even if all nodes are in the same network, the distant of two machines noticeable affects the network performance. If the topology of the cluster changes – despite the application is not involved – a rescheduling may be convenient to reach better performance. Since C/R is too expensive in terms of time and required resources, this scenario can be an interesting exploitation of migration techniques.

Regarding fault tolerance, the migration may in fact lead to a the reduction of the number of C/R required: an appropriate pre-fault detection system would trigger the migration if an imminent fault is detected, avoiding the long restart required if the fault happens.

Certainly, a sudden unexpected undetectable fault is not manageable with a migration technique. This is why C/R mechanisms cannot be fully replaced. Instead, the frequency of the checkpoints can be effectively reduced, provided an appropriate fault probability analysis.

The software architecture considered for this work does not provide a fault detector, but assume the presence of an external one that signals the resource manager in case of imminent fault. Subsequently, the resource manager and the MPI framework will trigger the migration. With this setup, the resource manager is the only interlocutor with the MPI framework and it can allocate and possibly reallocate resources over available nodes.

1.6 Thesis structure and objectives

The main topic of this thesis is a novel approach to process migration implemented in Open MPI. This technique was also presented at the EuroMPI 2016 conference [21]. In addition, the exploitation of this technique with the **Barbeque Run-Time Resource Manager** (from now simply *BarbequeRTRM*) is presented in conjunction with a basic centralized resource management policy for distributed systems.

In Chapter 2 the State of the art related to migration and C/R techniques is discussed and the novelty of the proposed method is highlighted. The Open MPI and CRIU frameworks are described in Chapter 3. Subsequently in Chapter 4

and 5 the design, the implementation and the integration with *BarbequeRTRM* are explained, detailing and arguing all design choices. The testing results are discussed in Chapter 6, focusing on the introduced overheads. Eventually, in Chapter 7 future research directions and developments are proposed. It is also the thesis end containing the conclusions.

CHAPTER 2

State of the Art

This chapter presents the state of the art approaches for addressing the issues presented in the previous chapter. The first part is characterized by the analysis of Checkpoint/Restart techniques available in literature, with a particular focus on HPC environments and MPI applications. The second part instead aims at providing an overview of solutions to perform *process migration*, including the migration of MPI processes. Finally the chapter is closed by a brief survey of distributed resource management approaches.

2.1 Checkpoint/Restart approach

Checkpoint/Restart (C/R) – sometimes called Checkpoint/Restore – is a widespread technique to enforce fault tolerance in computing systems. C/R are essential for parallel and in particular HPC applications for the reason presented in Chapter 1. C/R tools perform periodical *checkpoint*s to save the state of the processes in a persistent storage medium. This state is usually called *image*. After a system fault, the image can be recovered in order to *restart* the execution from the saved checkpoint state. The idea is depicted in Figure 2.1. C/R tools usually adopt the following schema:

1. Synchronization of the application processes execution to reach a global

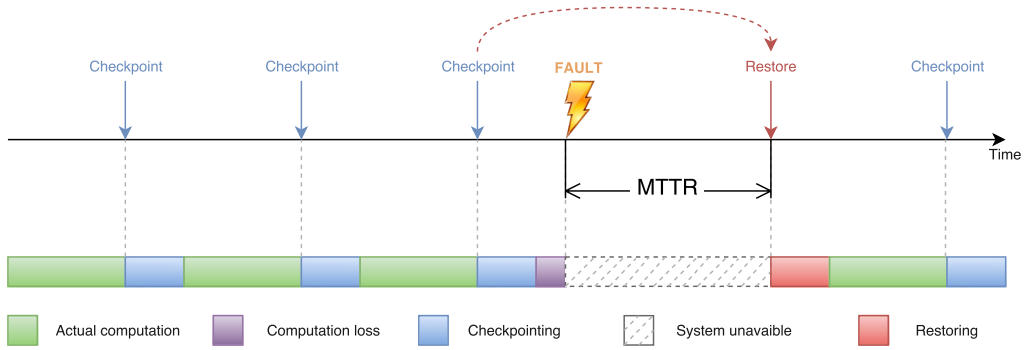

Figure 2.1: *The Checkpoint/Restart approach.*

consistent state;

2. Application execution state saving (*checkpoint*);

3. Application execution resuming (*restart*).

In case of fault of parallel applications, all the running processes are killed and the application is restarted resuming the state from the last checkpoint. Fault-tolerance approaches like C/R are called **Rollback-recovery techniques** [22].

2.1.1 Classification

At first glance, we can classify Checkpoint/Restart approaches on the basis of the level at which they operate:

- **Application-level**: the C/R functionalities are provided by a specific library linked with the application. Usually the application is aware of C/R and must provide barriers or similar synchronization point to reach a consistent state of all processes and threads. Reaching a so called **Global Consistent State** allows the library to perform a safe checkpoint and restore;

- **System-level**: the C/R is performed by the operating system or by the programming framework (e.g. MPI). The application is usually unaware of C/R and cannot therefore handle C/R events.

Alternatively, we can distinguish C/R approaches on the basis of the software layer, or granularity, at which they act:

- **Virtual Machine-level**: the classic approach in Infrastracture-as-a-Service virtualization. The application processes are distributed in several virtual

machines governed by a hypervisor. It provides the virtual machine requirements (e.g. isolation) and also the capability of C/R (called *snapshots*). A virtual machine can be moved to another physical server or stopped and resumed if needed;

- **Container-level**: the operating system-level virtualization provides *containers* on top of which the application processes run in a isolated environment. The approach is similar to the one based on virtual machines with the exception that there is only one instance of guest OS running on the host machine. Similarly to Virtual Machines, the C/R tools for containers are usually available and easy to implement and use;

- **Process-level**: C/R saves and restores the execution of each single process. This approach is more complex than moving a virtual machine or a container because the process is not completely isolated, In fact, in order to correctly save the execution status, all the input/output channels (file descriptor, sockets, etc.) need to be closed. The restore stage also presents several problems, for instance when the execution is resumed, the process identification number (PID) must be guaranteed to be the same. However, running natively the application on the system does not add the typical overhead of virtual machines and containers.

As shown by the timeline in Figure 2.1, the cost of C/R is very high in terms of wasted time, especially, due to the periodic checkpoints required. Consequently, the overall overhead may reach over the 50% [14] of the total execution time.

2.1.2 Libraries and frameworks

The scientific interest in Checkpoint/Restart started in early 1990s with some *application-level* libraries. The first approaches to *system-level* checkpoints started in the same years, however the *system-level* tools most used today were created after 2005.

Regarding the application-level mechanism, one of the most known libraries is `libckpt` developed in 1994 by University of Tennessee for UNIX systems and later ported to Linux [23]. The user application has to be modified by adding the `libckpt` function calls, in order to instruct the linked library to periodically perform the checkpoints (default interval 10 minutes). This library implements also *incremental checkpoints*, that instead of create and save a full image for every checkpoint, it stores only the difference between the last and the current state images. With incremental checkpoints the overall size of subsequent images

on disk is dramatically reduced and consequently it also limits the I/O overhead due to the image storing.

In 2003 the Department of Computer Science of the Berkeley Laboratory (US Department of Energy) developed the **Berkeley Lab Checkpoint/Restart (BLCR)** software [24]. The approach is a process-level C/R for Linux, with a hybrid kernel/user space implementation, designed with HPC applications in mind. This open source tool was improved and maintained until 2013. BLCR requires to load a custom kernel module, that is in charge of providing the access to data usually neither visible nor modifiable from user-space. For instance, the possibility to alter the PID sequence or to know the filename associated to one file descriptor. The user-space BLCR software uses the API provided by the kernel module to implement the checkpoint and the restart of the process.

Starting from Linux version 3.3 (released on March 18, 2012), it became possible to produce full user-space C/R tool, thanks to the

```
CONFIG_CHECKPOINT_RESTORE
```

kernel option. This option enables the application to access to kernel parameters, like modify the next PID in the PID sequence. To the best of our knowledge, **CRIU (Checkpoint/Restore in Userspace)** [25] is currently the only available tool that exploits that kernel option. Since an user-space tool presents several advantages (higher portability, security, maintainability, etc.) compared to a kernel-space one, CRIU was selected as C/R tool for this thesis. In the next chapter, a description of CRIU and the motivations of this choice will be extensively discussed.

2.1.3 Checkpoint/Restart in MPI

Since the early years of development of MPI implementations, C/R was one of the most hot research topic. Several C/R tools have been proposed to be integrated in MPI framework or designed to directly work with MPI, like the previously cited BLCR.

One of the first C/R implementations for MPI was proposed in 1997 by *Li et al.* [26]. This C/R consists of multicast daemons that assist the MPI runtime to perform communications and checkpoints. The checkpoints are locally coordinated at single node level, while the processes running on remote nodes are considered out of scope. Moreover, the multicast daemons replace the MPI library communication module in order to provide a retransmission mechanism when messages are lost due to an ongoing checkpoint. This C/R was implemented in the LAM/MPI library.

Hursey et al. [27] extended the Open MPI stack with additional layers providing C/R capabilities. The approach is based on a dedicated Checkpoint/Restart Coordination Protocol (CRCP), to support the synchronization among the nodes of the distributed systems. The coordination is split is three phases: *pre-checkpoint*, *continue* and *restart*. The first phase has the duty to bring the system to a checkpointable state, i.e. it has to ensure the synchronization of the processes, the absence of in-flight messages, a safe state of all file descriptors, etc. The *continue* phase is indeed the checkpoint: close all socket connections and call the BLCR routines in order to save the image. Finally, the *restart* occurs on demand (e.g. after a fault-recover) and restores the previously saved image. A similar approach, still exploiting, BLCR was proposed in 2005 for LAM/MPI [28].

One of the advantages of this technique is the ability to be network agnostic. The application processes can be stopped and then restarted on a different set of nodes, potentially characterized by a different network topology. The coordination protocol has to consider also the exchange of information about the new topology over all nodes and their processes. However, this solution introduces noticeable code dependencies between internal Open MPI modules. Moreover, it induces significant overheads: copying the process state images onto an external storage server becomes in fact a real bottleneck for the system. The previous drawbacks, along with the poor maintainability of the software, led the Open MPI developers to disable these additional layers since Open MPI version 1.7. Nevertheless, this work is still the only working C/R available in the Open MPI mainline code repository.

For the sake of completeness, C/R can be used also for purposes different from fault-tolerant. For example, it can be used to efficiently debug parallel applications [29]: if the programmer want to *rewind* the execution, he or she does not necessarily restart the application from the begin, but only from the last saved checkpoint.

2.1.4 Limitations

As already briefly discussed in Chapter 1, the common limitation of C/R based approaches is the overhead introduced by performing periodical checkpoints, which is done during the entire application lifespan. In some use cases this overhead impacts dramatically, even doubling the execution time of the applications [30]. For example, to quantify this overhead, consider that in 2013 some TOP500 HPC systems requires a checkpoint time between 40 to 60 minutes [10]. A further problem is that this side-effect increases exponentially with the system

size, i.e. the number of computing nodes. Considering a large HPC system with thousands of nodes and not negligible power supply costs, the overhead must be evaluated not only in terms of time, but also in terms of energy consumption [31].

In systems based on Virtual Machines or Containers, implementing C/R mechanisms – even if in conjunction with MPI – is relatively easy. However, virtual machines also have a significant impact on the application performance, especially for I/O intensive workloads in HPC systems [32]. The lack of shared memory communication between processes on different virtual machines indeed has been considered an important inefficiency since the beginning of virtualization's use in HPC environments [33]. Although many approaches have been proposed to mitigate this problem, the significant overhead persists even today, inducing an increment of latency by a factor up to 16x for communication intensive operations [34].

2.2 Process migration in MPI

Process migration – or **task migration** – is a technique that can be used in alternative or in support to C/R. When the allocation of nodes has to be changed (e.g. due to a failure of one node), classic C/R techniques consist in checkpoint the application a restart it over a different subset of nodes. This job can be performed more efficiently with process migration, in order to involve only the nodes that have to be moved.

In large cluster, process migration is useful to add reliability and to balance the resource allocation across the cluster [35]. The migration request can be done centralized (e.g. by a resource manager) or by the single node, that for instance may request a migration if it's overloaded or an imminent fault is going to occur. While it has the potential benefits described, the entity that triggers the migration must take in account the considerable migration cost.

In 1996, *Stellner* proposed the first migration technique in MPI: the *Cocheck* environment [36]. This environment was built on top the MPI framework and not inside (actually small modifications to MPI framework were applied). The global consistent state is achieved by imposing no message in-flight over the network. Then, checkpoint or a migration of a subset of processes is performed according to what is required.

Process migration can be used also to supported classical C/R approaches, as presented by *Wang et al.* [37]. Their work introduced a process-level migration that allows us to potentially achieve a higher utilization of the system resources, with respect to virtualization based approach. The basic idea of the authors is to try to minimize the number of C/R by using a proactive approach: health moni-

toring of the computing node state and migration of all the running processes on a different node, in case of imminent fault prediction. Their approach reduces the number of performed C/R with respect to periodical checkpoint based techniques. However this solution requires to synchronize all the running processes into a global consistent state, before stopping and migrating them to a new node. This can represent an issue in case of imminent faults that require short time to act. The approach was implemented in LAM/MPI (predecessor of Open MPI) using the BLCR tool.

According to the aforementioned works, we can argue that the main issues to be tackled when dealing with processes migration in HPC systems are:

- Design an easy to maintain migration framework;
- Enable migration support without introducing changes in the application code;
- Enable the possibility of migrating just part of the application, i.e. a subset of processes;
- Do not bind the migration overhead to the synchronization of the processes execution into a global consistent state;
- Provide interfaces to allow a resource manager to drive the processes migration according to smart policies.

The solution we propose addresses all the aforementioned issues:

1. It is a process-level migration mechanism whose granularity can be tuned by the resource manager;
2. It does not require any change to the application code;
3. The migration is almost completely transparent with respect to the application execution;
4. Migration can be triggered by a resource manager through a suitable API.

2.2.1 Heterogeneous process migration

Process migration is usually available in homogeneous clusters, since migration in heterogeneity conditions is much more complex [35].

In 1998, *Smith et al.* presented the Tui System [38], an experimental framework to perform process migration between heterogeneous machines. The article was well received by the scientific community and it shows the several issues

affecting the heterogeneous migration. The main problem is the conversion between different ISA, that may require different instructions, register numbers, register size, etc. For this reason, the compiler is compulsory involved, because it must produce a code that matches one-to-one between different architecture. As a consequence, in order to simplify the problem, the Tui System has strong constraints regarding the two architectures involved, that leads to a migration much more *quasi-homogeneous* than heterogeneous.

Other few solutions were proposed, e.g. *Cabello et al.* [39] proposed an Open MPI middleware to provide migration mechanism in heterogeneous systems. In this case the process is not directly migrated but a new process is started in the destination machine. Unfortunately, this middleware provides dedicated MPI calls, violating the standard and requiring substantial rewrite of all MPI applications.

2.3 Distributed resource management

The academic interest in resource management significantly increased since 2005, when the first multi-core processors became available. However, the resource management of distributed systems goes back to 90s, when the resource scheduling and allocation became a necessity for large data-centers.

Classical feature of a resource management systems is to provide an efficient assignment of resources to jobs. This problem is called **resource allocation** or **resource scheduling** [40].

For the first Grid systems[1] resource management was essential and still plays a crucial role in this type of distributed systems. The resource manager has to cooperate to guarantee the properties of scalability, responsiveness, fault-tolerance, stability – that are in common with Distributed Computing Environments – but also security, heterogeneity, isolation, distributed ownership, and QoS [41]. In this regard, a variant of MPICH – called MPICH-G – was proposed in 1998 to enable MPI to work in Grid computing [42].

Nowadays, delivering high throughput and low latency in supercomputers getting close to Exascale requires a resource management system sufficiently scalable. Current centralized resource managers seem not a feasible solution to guarantee the performance requirements in Exascale, so the current research is oriented in distributed resource management [43].

Not all HPC environments require a perfect efficient scheduling of resources, due to policy reasons. In distributively owned environments, the owner of a re-

[1] A Grid system is a large distributed Network Computing system with machines distributed over several data-centers

source defines the access policy for that specific resource, for instance setting time constraints. These policies introduce more complexity in modeling the system, and consequently in the resource management performance [40].

Furthermore, in last years, distributed resource management becomes not only important for HPC applications, but also for energy-aware resource management in Cloud Computing infrastructures [44].

2.3.1 Resource Management and Open MPI

Name	Developer	Last stable release	Reference
ALPS	Cray User Group	April, 2012	[45]
Grid Engine	Proprietary and open source implementations available	January, 2014 (open source version)	[46]
LoadLeveler	IBM Corporation	unknown	N.A.
LSF	IBM Corporation	August, 2016	[47]
SLURM	Open Source community	July, 2016	[48]
Torque (ex PBS)	Adaptive Computing	August, 2016	[49]

Table 2.1: *The supported resource managers in Open MPI.*

At the time of writing Open MPI supports several resource managers, as summarized in Table 2.1. Most of them statically assign the resource to the MPI application at launch-time. Two different approaches are used in current resource managers: interactive or wrapped. In the first case, the `mpirun` command is invoked while Open MPI subsequently interrogates the resource manager asking to which nodes dispatch the application processes. In the second case, a wrapper script interrogates the resource manager and subsequently execute the `mpirun` command. Then, Open MPI typically retrieves the information via environment variables or similar mechanisms.

Even if some of the previous resource managers offers features like C/R and migration (e.g. SLURM via BLCR), they are not integrated in Open MPI. The first part of our work is to add the support of the BarbequeRTRM to Open MPI, including the dynamic rescheduling of application during runtime. This feature is implemented with process migration.

CHAPTER 3

Open MPI and CRIU internals

The goal of this chapter is to describe the current implementations of the Open MPI and CRIU frameworks, in order to introduce the concepts necessary in Chapter 4. The development of both frameworks is very active and the information in this Chapter may be outdated in latest releases. The information reported in this chapter refer to version 1.10 for Open MPI (November 2015) and version 2.1 for CRIU (April, 2016).

Open MPI works both on 32-bit and 64-bit architectures, running in POSIX-like operating systems. Linux and OS X are fully supported, while Solaris only partially. On Solaris Open MPI can therefore not work properly. Finally, Microsoft Windows compatibility was dropped since version 1.8 (March, 2014).

CRIU instead supports only Linux operating system, starting from kernel version 3.11.

3.1 Open MPI architecture

3.1.1 Modular Component Architecture

Open MPI design is based on the **Modular Component Architecture** (MCA) [20], which is composed by three entities:

- *MCA*: the backbone, in charge of the correct instantiation of all other modules. It loads the run-time parameters and passes them to the correct frameworks;
- *Frameworks*: the main functional parts in which Open MPI is structured. Each framework is devoted to a specific task. For example, the process lifetime management or the input/output forwarding services. For each framework several implementations can be available, called components;
- *Component*: an implementation of a framework. Each functional part may be implemented in several ways, for instance by relying on different protocols (e.g. TCP or InfiniBand). More concretely, a framework implementation can be made by more than one component: a *base* component exposing the common functionalities functions to all the other components in the framework, plus other optional components.

Frameworks are usually built all together, but according to the requirements it is possible to disable the framework at compile-time. The components can be instead selected at compile-time and even at run-time, via appropriate commands. If more than one component is available at run-time, the MCA selects the one with the highest priority. The priority is assigned by the components developers.

Furthermore, in order to maintain a logical separation between different functional areas, the frameworks are grouped into three sections:

- **OMPI**: *Open MPI*, the application-level API. most of the frameworks in this section are compiled into the shared library, that will be linked to the user application.
- **ORTE**: *Open Run-Time Environment*, the underlying subsystem controlling the life-cycle of application processes, coordinating the execution and providing for the MPI collective communications.
- **OPAL**: *Open Portable Access Layer*, an utility library that provides to OMPI and ORTE a set of common frameworks, like event management, memory allocation, etc.

The structure of Open MPI repository reflects the architecture design. The code is split over several directories, whose filesystem path is structured as follow:

`/<section>/mca/<framework>/<component>`

For instance, the component that provides the packet routing for coordination messages in a *de Bruijn network* is located at:

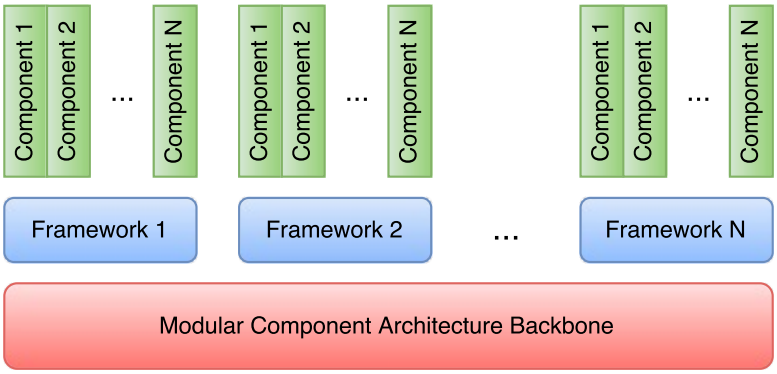

Figure 3.1: *The conceptual diagram of Modular Component Architecture implemented in Open MPI*

`/orte/mca/routed/debruijn`

The features provided by *ORTE* are exploited by a dedicated daemon, called **ORTE daemon** or shorten `orted`. An instance of the daemon is spawned in each node assigned to Open MPI. It is in charge of launching and managing the local processes of the application. When `mpirun` is executed in one node, a *ORTE daemon* is started[1] getting the name of **Head Node Process (HNP)**.

The overall software design is shown in Figure 3.1, while in the next subsections the functions provided by some frameworks are described.

3.1.2 Execution Flow and Communication Layers

After the execution of `mpirun` the HNP interrogates the **Resource Allocator Subsystems** (`ras`) to get the list of available nodes. Depending on which RAS component is active, it may contact a resource manager or perform other actions, like reading the node list from the host file (default action if no resource manager specified). How the interaction with resource managers works is described in the next Section 3.2.

The **Process Lifetime Manager** (`plm`) is subsequently in charge of spawning the *ORTE daemons* on the other nodes specified by the list provided by the RAS. The PLM maintains also the communication between the various instances of *ORTE daemons* at high-level, i.e. it decodes the messages and dispatches them to the framework in charge of handle it. The `plm` framework implementation can delegate the process spawning the resource manager. The default selected component – and the component used in this work – is the well known **Secure SHell (SSH)**. The `orted` commands are executed in the other nodes directly via

[1] To be precise, the `mpirun` command after the elaboration of command line parameters starts to act as an `orted`. This means that in the system process list, we will see `mpirun` and not `orted`, even if it actually executes the code of `orted`.

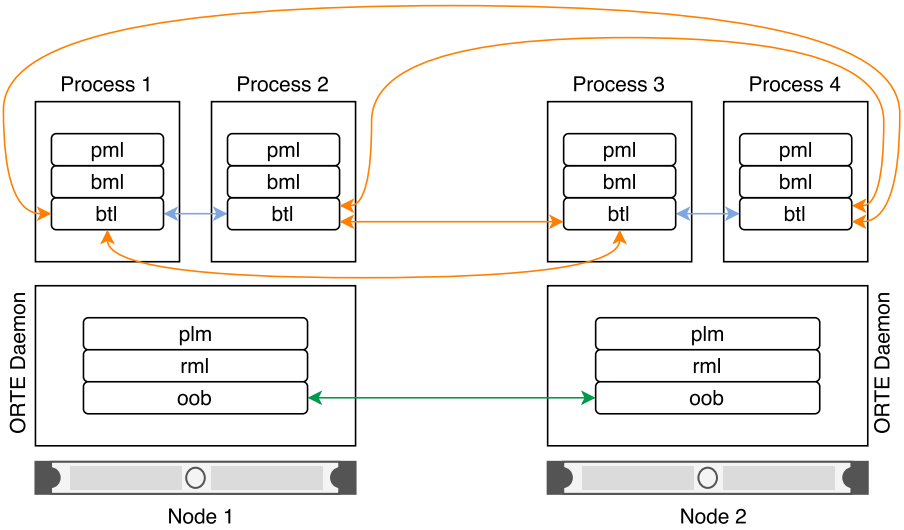

Figure 3.2: *Network diagram. The orange channels represent the inter-node application-level communications, the cyan channels represent the intra-node application-level communications, and the green one represents the* `orted`*-level communications.*

`ssh` calls.

Subsequently to the spawning of *ORTE daemons*, the coordination of the execution takes place distributing a series of messages (having a semantic like "start X processes on node Y") performed at high-level by the `plm`. The `plm` entrust on two *ORTE* stacked layers, that are in charge of managing the low-level communication: **Run-Time Messaging Layer** (`rml`) and **Out Of Band** (`oob`). The latter manages the low-level byte exchanges that is usually performed via a TCP/IP network.

Specularly, the application-level messaging is managed by *OMPI* subsystems:

- **P2P Management Layer** (`pml`) catches MPI calls and manages fragmentation/reassembly of high-level messages;
- **BTL Management Layer** (`bml`) provides the routing to the correct and optimal network device;
- **Byte Transfer Layer** (`btl`) low-level layer in charge of perform system calls (e.g. socket). The current available implementations are: Transmission Control Protocol (TCP), InfiniBand, Portals 4, User-Level Generic Network Interface (uGNI), Ultra low latency Ethernet (usNIC). For intra-machine communications there are also available shared memory, Intel Symmetric Communications Interface, Vader.

The described communication architecture is summarized in Figure 3.2.

Input/Output

The standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`) of the application are exchanged via a framework called **I/O Forwarding** (`iof`). Particularly, each process of the application may send bytes over `stdout` or `stderr` and may receive bytes on `stdin`. The `iof` framework is in charge of distributing these streams across different nodes, usually `stdout`/`stderr` towards HNP and `stdin` from HNP to remote daemons. This framework, like `plm`, is on top of `rml` and `oob` layers.

When a *ORTE daemon* receives `stdin` data from `iof`, it dispatches the data to its children via a previously opened pipe[2]. Vice versa, when the *ORTE daemon* receives data on pipes dedicated to `stdout` and `stderr` it forwards the data to HNP via `iof`.

3.1.3 Event Management

The internal Open MPI events are managed by the *OPAL* framework `event`. This framework is in turn based on **libevent** [50], a cross-platform event notification library. Every component may delegate to the `event` framework the check for an event. When the event happens, the `event` framework executes the callback function provided. Open MPI by default executes on a single-thread, both the *ORTE* and the *OMPI* subsystems. Therefore, all functions have to execute their code and terminate quickly in order to give back the control to the event manager routine.

By default Open MPI adopts an **aggressive mode**: the event manager does not voluntarily yield the processor, but only if the operating system scheduler preempts it. Instead, if it is waiting for a specific event (like a message coming from a socket) it keeps spinning in order to check the event firing. This technique may help in order to reduce the *reaction time*, avoiding the context-switch overheads.

3.1.4 Auxiliary Executables

Open MPI has several executables in addition to the commands required by the MPI standard (`mpicc`, `mpiexec`, etc.). Each executable is linked with *ORTE*, *OMPI* and/or *OPAL* depending on the needed functions and the commands usually communicate with the running instances of *ORTE daemon* in order to retrieve information or perform actions. The list and the description of available commands is presented in Appendix A.

[2] A pipe is an unidirectional data channel used for inter-processes communication in Linux

3.2 Open MPI resource management

The Open MPI framework that provides resource management is the **Resource Allocator Subsystem** (`ras`). The implementations correspond to the supported resource managers presented in Table 2.1 plus the implementation for reading the node list from a file. Each `ras` component has to expose four functions (in addition to the usual functions required by MCA):

- `init`: a procedure called during the startup to initialize the component;
- `allocate`: it receives the job and it has to fill a list of nodes. It is in charge to contact the resource manager in order to request the node list and how many resources available for each node;
- `deallocate`: it receives the job that will be deallocated from the nodes. Currently, all `ras` components do not implement this function;
- `finalize`: a procedure called during the shutdown to clean the component and other finalization actions.

The components are called only during the setup of MPI applications, if they do not set other events. Multiple approaches are possible, e.g. *Torque* component analyzes the environment variables or *SLURM* component tries to contact the resource manager via socket.

3.3 CRIU architecture

Checkpoint/Restore in Userspace – abbreviated in **CRIU** – [25] is a software that exploits the recent configuration entry `CONFIG_CHECKPOINT_RESTORE` in Linux.

The open source software is developed by a community mostly made of Virtuozzo developers – a company located in Russia born from a spin-off of Parallels.

It provides several features, from the basic Checkpoint/Restart functionality to live migration. The services are provided through three different interfaces: *Command Line Interface* (via the `criu` command), *Remote Procedure Call* (using the Google Protocol Buffers), and the C *Application Program Interface*.

The CRIU checkpoint is invoked using the CLI interface with the following command:

```
$ criu dump -D <directory> -t <pid>
```

The `-D <directory>` parameter indicates the path where the image will be saved. The `-t <pid>` parameter indicates the PID of the process to checkpoint. CRIU checkpoints also all the children of the specified process. The image is saved in the provided directory, in form of several files that can be categorized as follows:

- **Inventory files**: they contain the general information about the image, e.g. version number, system general information, etc.
- **Image files**: the regular files that represent the process status. The most important are the memory dump files, that contain the pages correspond to the memory content of the process, both private and shared. Other files contain the open file descriptors, the address space information, time states, the task credential, the mountpoints list, etc.
- **Auxiliary files**: they keep statistics and other non-essential information; they do not include any process data.

Similarly, the CRIU restart is performed using the CLI interface with the following command:

```
$ criu restore -D <directory>
```

3.3.1 Linux Kernel Configuration

The `CONFIG_CHECKPOINT_RESTORE` is a kernel configuration entry present in Linux starting from version 3.11. This configuration enables additional kernel features in order to enable Checkpoint/Restart actions in user-space. In particular, it introduces:

- Some additional flags for the `prctl` system call. This system call allows the user to perform some actions on the process kernel descriptor. For example it is possible to change the behavior after a floating point exception and to request a signal send when the parent process terminates. Checkpoint/Restart requires to modify certain kernel memory map descriptor fields in order to restart correctly the process, e.g. the start address of the stack and the data segment size.
- Additional `/proc` filesystem entries, again in order to ensure a correct restore:
  - `/proc/[pid]/map_files/`: this directory contains the **memory-mapped** files, i.e. segments of virtual memory that have been assigned to particular file descriptors via `mmap` system call.

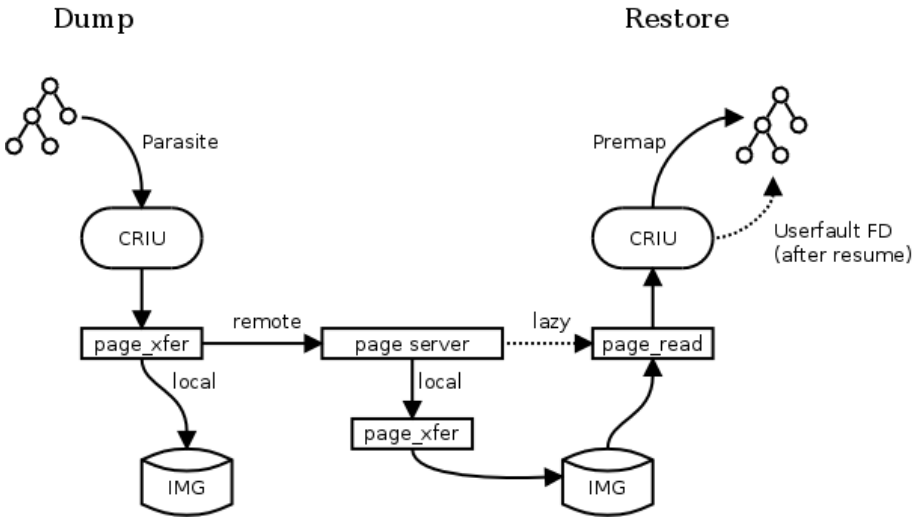

Figure 3.3: *The flow diagram of the Checkpoint (Dump) and Restart (Restore) phases of CRIU. (Authors: CRIU Developer. Republishing permitted under GNU FDL 1.3)*

  - `/proc/[pid]/timers`: this file provides the list of POSIX timers enabled for the process.

- The `/proc/sys/kernel/ns_last_pid` file: it enables a privileged user to change the kernel internal counter used to select the next Process IDentifier (PID).

3.3.2 The Execution Flow

The flow diagram of CRIU execution is depicted in Figure 3.3. The *page server* is a CRIU component that enables the transfer of the image to a centralized server or to another machine. It is not currently considered for this work and we omit other details.

Initially, a **parasite code** – built in *Position-independent code format (PIC)* – is injected into the *victim* application via the `ptrace` tool. The *parasite code* starts a daemon that will receive CRIU internal commands, preparing the application for the checkpoint. The most important purpose of this code is to save the execution context of the victim (registers, etc.). Subsequently, the full images of the victim process and its children are transfered into the disk, ready to be restored.

However, since the *parasite code* is a very limited code – it cannot be linked to any library due to the *PIC* nature – some actions are executed directly by CRIU process. The corresponding restore phase is called *Restorer context*. The data managed outside the *parasite code* is:

- the memory content

- all the timers information
- the credentials (permissions, `chroot`, etc.)
- the threads information

Before restarting the processes, the `/proc/sys/kernel/ns_last_pid` has to be set with the number immediately before the PID of the process we want to restart, providing the file is locked exclusively. Obviously, that PID and the PIDs of its children must be available in the system, otherwise the process restart is not possible[3].

It is possible to summarize the checkpoint phases in:

1. Inject code in the processes tree;
2. Collect processes resources and save it;
3. Cleanup: kill the application or remote the injected code to continue execution.

Specularly the restore phases perform the following actions:

1. Resolve shared resources in order to avoid to duplicate shared memory region;
2. Change the PID kernel counter and fork the processes tree;
3. Restore the processes resources;
4. Restore the processes context;
5. Restore the last information (timers, threads, etc.).

3.3.3 Advantages over other C/R tools

The first advantage is the high number currently supported architectures compared to other C/R tools:

- x86
- x86_64
- ARM
- AArch64
- PPC64le

[3]This is a strong limitation for migration, but it will be addressed in the next Chapter.

Part of CRIU code – in particular the **parasite code** – is strongly architecture dependent. Apart that code, CRIU relies on machine-independent Linux system calls.

The second, but probably the most important advantage of CRIU compared with other C/R tools, is the ability to run completely in user-space, even if it requires the `root` user permissions. Removing the limitation of administrative privileges is an on-going development. Running in user-space instead of kernel-space presents several advantages, among which it is possible to find: better maintainability and portability, less security vulnerabilities and safety risks.

The third main reason that leads us to select CRIU, is the very active development community, that provides not only bug fixing, but also new features, with monthly cadence releases. Some communities that developed other tools are no more active, like BCLR.

CHAPTER 4

Enabling Process-Level Migration in Open MPI

The software implementation proposed in this thesis can be split in two parts. The first one – subject of this chapter – is the implementation of the `mig` framework and the related changes in Open MPI. The second one – presented in Chapter 5 – is the integration of Open MPI with Barbeque Run-Time Resource Manager, including a resource management policy that exploits the migration mechanism introduced.

4.1 Process migration flow design

The migration mechanism proposed is integrated in the Open MPI as a novel framework, leaving to the resource manager the implementation of the logic according to which is it worth to require a migration to Open MPI. The idea is to relieve the MPI framework all the aspect of resource management, fault tolerance and other tasks that would be better performed by an external manager. The resource manager may have a wider visibility on the overall distributed system (or cluster) and may consider also non-MPI application in the resource accounting.

The core idea of the proposed migration technique is to be as much as possible transparent not only with respect to the applications, but also to other Open MPI frameworks, consequently proposing a system-level and process-level migration. At the time of writing the number of frameworks is over 50, each of them

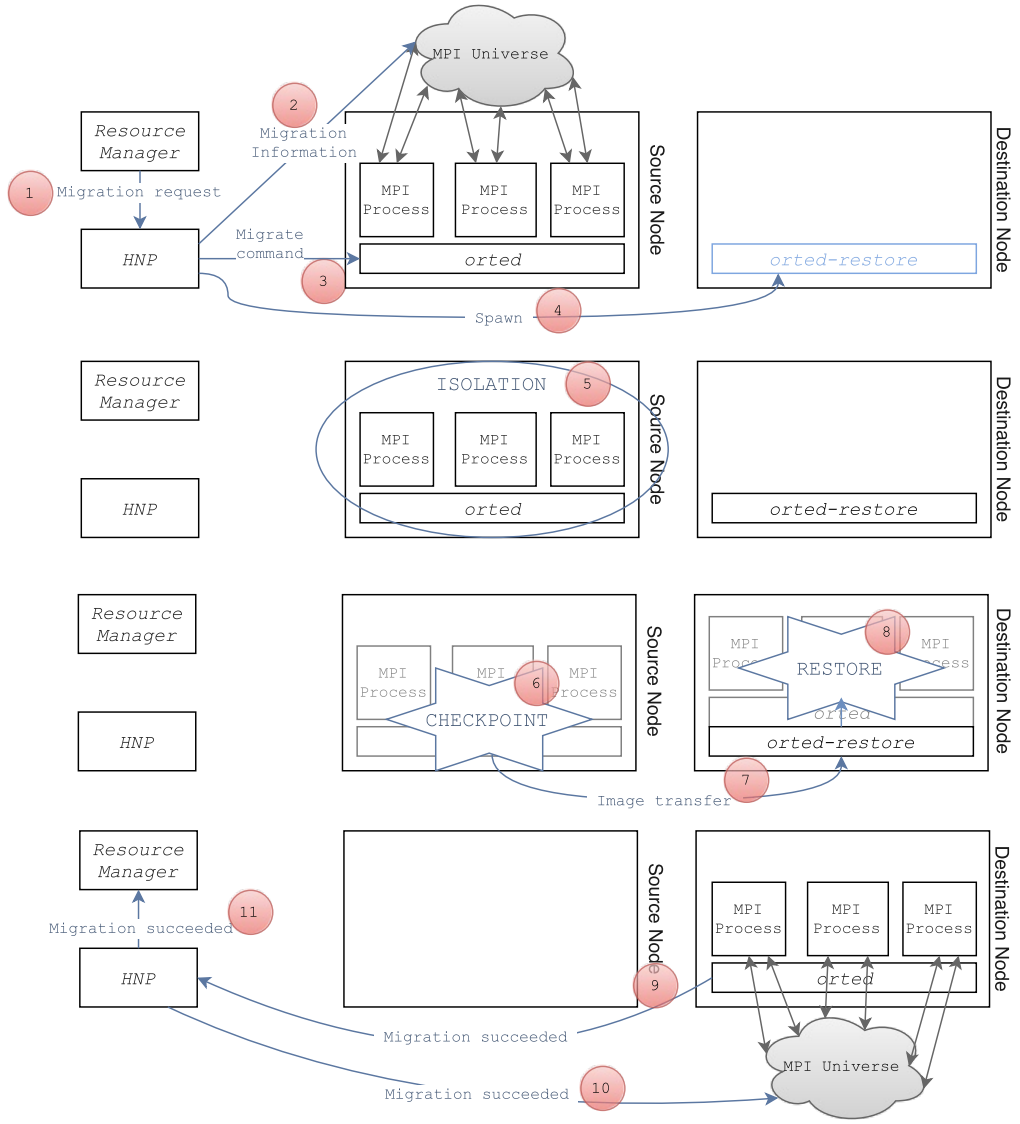

Figure 4.1: *Description of migration mechanism implemented through the* `mig` *framework.*

having between 1 to 10 components. The last consideration is the main reason to guarantee the maximum transparency with respect to other frameworks and it is one of the key differences with previous system-level approaches in C/R and migration in Open MPI and it ensures simplicity and maintainability of the proposed solution.

Our migration mechanism consists of migrating not the single MPI process, but the overall *ORTE daemon*. This means that the migration involves batch of Open MPI processes, in first approximation all processes in each node. Subsection 4.1.1 contains further discussions about the migration granularity.

Following the flow depicted in Figure 4.1, the designed sequence of migration

is:

1. The resource manager asks a migration to the HNP;
2. The HNP informs all *ORTE daemons* and for extension all MPI processes;
3. The HNP issues the migration command to the *ORTE daemon* on node requested to be migrated;
4. Simultaneously with the previous command, `orted-restore` is spawned in the destination node (see Subsection 4.1.2 for details);
5. The migrating *ORTE daemon* issues the command to own processes to isolate themselves with respect to remote MPI processes;
6. The checkpoint is performed;
7. The resulting image state is transfered to the destination node;
8. The `orted-restore` performs the restart;
9. The restarted `orted` informs the HNP of the successful migration;
10. The HNP notifies all other processes about successful migration. They starts the restoring of the connections;
11. The HNP notifies the resource manager about successful migration.

To actually execute the steps 6 and 8 the Checkpoint/Restore In Userspace (CRIU) tool is used, as described in next Subsections. The most important overhead is the step 7, as subsequently confirmed by the benchmark presented in Chapter 6.

4.1.1 Multiple ORTE daemons on the same node

The migration mechanism works at `orted`-level and this entails the limitation of moving only the entire set of processes assigned to a node. Therefore, it is not possible to move a single process.

In order to mitigate the previous limitation, we added to Open MPI the capability to spawn multiple *ORTE daemons* on the same node. This is not usually desired since two processes in the different *ORTE daemons* cannot communicate via shared memory, but they need to use *TCP* or other network protocol. However, network protocols routed in local does not impact significantly. The overhead introduces will be analyzed in Section 6.

In HPC oriented systems, even a small local overhead may have a not negligible impact on the execution of parallel applications on multiple nodes. However, migrating with the granularity of *ORTE daemon* presents several advantages for process migration:

- The MPI processes does not need to change the active component used in communication. If two processes are spawned by the same *ORTE daemon* they communicate via shared memory and after migration they are still able to communicate via shared memory. Vice versa, if two processes are in different *ORTE daemon* groups, they communicate via network (e.g. TCP) and they still use that network (e.g. TCP) after the migration of one of *ORTE daemon*;

- It is not necessary to close and reopen the communication channels between *ORTE daemon* and its MPI processes, i.e. the POSIX pipes. The CRIU framework guarantees that they will properly work also after the restore;

- The Open MPI internal states have to be changed in a minimal part. Assuming all nodes homogeneous, the most important parameter that will change after migration is the IP address of that daemon and processes. Other information that will change are the hostname, the domain name, etc;

- Most of the Open MPI frameworks, as well as the applications, are completely un-aware of the migration. They notice only a long delay in the communication network from and towards the migrating processes.

4.1.2 The `orted-restore` helper program

After the checkpoint of the source *ORTE daemon* the image has to be transfered on the destination node. Subsequently, the `orted` has to be restarted via the appropriate CRIU API calls.

Initially, we thought about using `ssh`: a connection from the source node to the destination node is used to copy the image in the new node and another connection from *HNP* to destination node would issue the command necessary to the restart. However, this idea was put aside due to the large overhead of `ssh` connection, both for the connection latency and for data encryption overhead. A standard AES256-CTR chiper may easily halve the throughput [51].

Currently, the image transfer is performed in plain-text via TCP socket between the *ORTE daemon* and the program `orted-restore` spawned on the destination node for this purpose. The `orted-restore` has also the task of

invoking the restart function of CRIU. Once the migrating `orted` and its processes are successfully restarted, the `orted-restore` waits for their termination before exiting.

In order to spawn the `orted-restore` on the remote machine, the `mig` framework relies on the `plm` framework functions. The `plm` spawns the `orted-restore` using the same method used to spawn the *ORTE daemons*. Currently, we implemented the functionality only for `ssh` component of `plm` - the most used one -, anyway it's easy to extend other components to support this feature.

We decided to implement also the possibility of compressing the image before the transfer. The performance measures presented in the Chapter 6 reveal that for the most HPC applications, compress the image does not give any advantage. The resource manager is in charge of choosing between compressing or not the image.

4.1.3 The global consistent state

As previously discussed in Section 2.1, a **Global Consistent State** is typically reached by the existing C/R tools before performing the Checkpoint of the application.

In our implementation a *Global Consistent State* is simply never reached. Since we are interested in achieving the maximum transparency with respect to the application with the minimum migration overhead, reaching a safe-point by all MPI processes is an avoidable feature. Furthermore, processes not involved in migration are able to continue execution if they do not require to communicate with migrating processes.

Even if a *Global Consistent State* is not reached, the correctness of the application is guaranteed by the fact that no messages are lost during the migration. This is enforced thanks to a careful coordination of the migration and the modifications applied to `btl` described in next sections.

4.2 Open MPI modifications

4.2.1 The `mig` framework

The `mig` framework, shorten for *migration*, is the new framework added by this work. It is the core of migration mechanism both in HNP and in processes. Part of the framework is executed only in HNP and it has the role of coordinating all the migration phases. Other functions are executed only in the *ORTE daemon*

(e.g. the CRIU calls) and others only in the MPI processes (e.g. to perform the IP address changes).

Components

The framework has several common functions in the `base` component while other components provide the Checkpoint/Restore functionalities. Currently only the `criu` component is present, since we selected CRIU to perform the C/R of processes.

The functionalities provided by the base component may be overwritten by the active component. At present, the functionalities of `criu` component are complementary to the `base`, thus no overwrite is necessary. The `base` components provides:

- The entry point function called by `ras` to start the migration procedure;
- The exported callback function that `plm` will use when a migration message is incoming;
- A function called by the HNP, in order to refresh the data changed after migration, e.g. the hostname of migrated *ORTE daemon*;
- The image transfer functions between the migrating `orted` and the `orted-restore` on the destination node;
- The timing to perform benchmarks on the migration mechanism.

Instead, the `criu` component implements the API calls to CRIU, in order to actually perform the Checkpoint and the Restore of MPI processes. These functions are never executed in the MPI processes, but only in the migrating `orted` and in the `orted-restore` executable.

4.2.2 Other frameworks

Besides the newly introduced `mig` framework, the migration mechanism required the modification of other already existing frameworks. These changes are not very intrusive and usually they have been added in separated source files, with the exception of the `btl` framework, that required several injected lines of code. This separation guarantees a good level of maintainability.

Changes required for enabling the migration mechanism have been introduced in:

- `ras` (part of ORTE): currently, Open MPI uses this framework only in the application initialization, to get the full list of available nodes. We extended

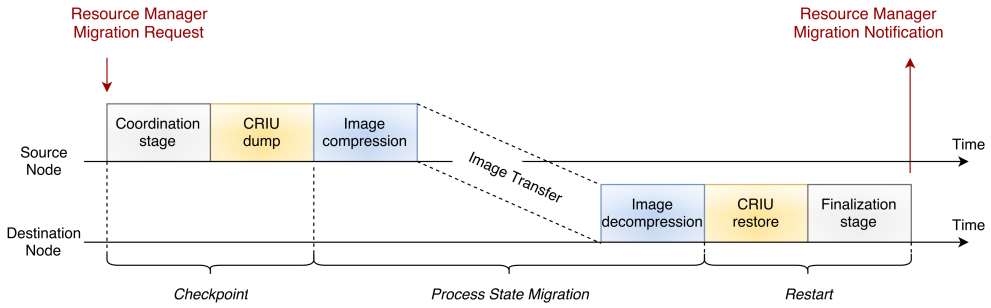

Figure 4.2: *The migration phases.*

the `ras` API to allow the resource manager to send migration requests during the applications execution and to be notified about the status of the requests.

- `oob` (part of ORTE): the "out-of-band" framework provides the low-level API for the communication between HNP ⇔ `orted`, and `orted` ⇔ its child processes. The current Open MPI implementation includes TCP as the only transport layer for *ORTE daemons* inter-communication. This framework contributes to the process migration by managing the opening and closure of the pending TCP socket connections towards the migrating *ORTE daemon* instance.

- `plm` (part of ORTE): high-level HNP ⇔ `orted` communication framework. We implemented the protocol necessary to coordinate the *ORTE daemon* instances in the *base* component. We also added the `ssh` call that spawns the `orted-restore` daemon on the destination node. This daemon is in charge of resuming the processes execution once the checkpoint image transfer is completed.

- `btl` (part of OMPI): this is the application-level peer-to-peer communication framework. In our work we modified the *TCP* component to manage the opening/closure of the TCP socket connections among migrating application processes. See Section 4.4 for details.

4.3 The migration phases

This section describes in detail how the proposed migration mechanism is structured. We have divided the migration procedure in five phases:

- Coordination stage;

- CRIU dump;

- Process state migration;
- CRIU restore;
- Finalization stage.

This schema is shown in Figure 4.2.

4.3.1 Coordination Stage

The coordination stage starts when the `mig` framework on the Head Node Process receives from `ras` a migration request specifying a source and a destination nodes.

The `mig` framework spawns an `orted-restore` daemon on the destination node, which is therefore able to receive the migrating ORTE daemon. Then, via `plm`, the framework issues a `MIGRATION_PREPARE` command to all the ORTE daemon instances running over the system, broadcasting the information related to the migration request. When the ORTE daemon instances receive the command, they notify the request to their children (the application processes) using the signals provided by Linux-OS[1].

The signal handler, implemented in the OMPI library, intercepts the `MIGRATION_PREPARE` signal/command. The `btl` TCP component of the processes that are not migrating performs the following actions:

1. Caching of any future send request towards the migrating processes;
2. Terminating any ongoing data transmission (`send()` system calls) towards the migrating processes;
3. Flushing the transmission buffer;
4. Performing a `shutdown` system call on the transmission-side of the TCP socket.

After that, the processes send back an acknowledgement to their own ORTE daemon instances, ensuring that no further transmissions will be performed towards the frozen processes. In turn, the ORTE daemons forward the acknowledgement to the Head Node Process. This synchronization protocol, which is depicted in Figure 4.3, is involved in all the subsequent phases.

[1]Open MPI developers have planned to release the `pmix` framework, which allows the ORTE daemon to communicate via Unix sockets to its children. Our approach will be changed accordingly

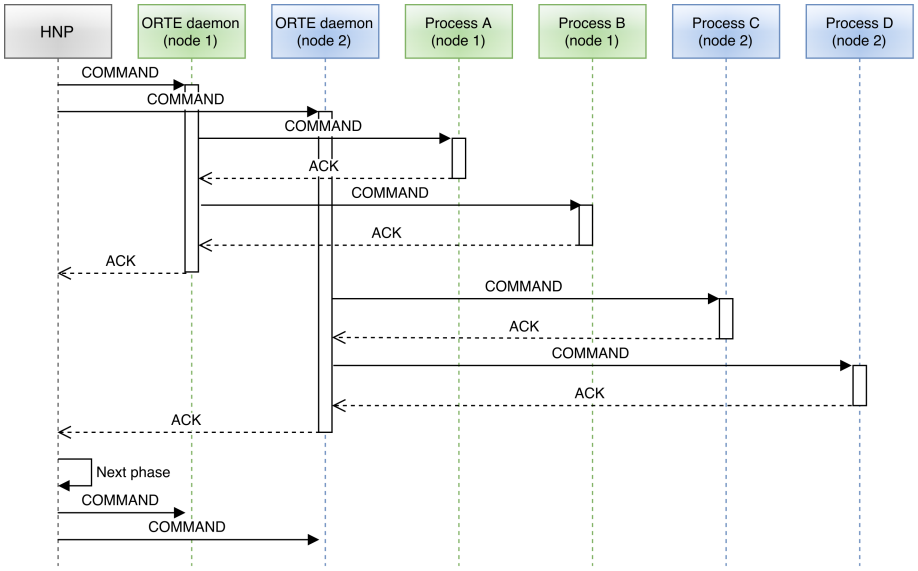

Figure 4.3: *Sequence diagram migration messages exchange.*

4.3.2 CRIU Dump (Checkpoint)

Once all the ORTE daemon instances are aware of the migration request, the Head Node Process can issue the `MIGRATION_EXEC` command and effectively start the migration procedure. When an application process receives the `MIGRATION_EXEC` command, it waits until all the in-flight packets have been received by the destination side. At this point, all the TCP connections towards processes involved in the migration can be safely closed, and an acknowledgment can be sent back to the ORTE daemon.

When the migrating ORTE daemon receives the acknowledgment, it uses the API provided by the CRIU library to perform the checkpoint of its execution status. The checkpoint outcome, i.e. the generated process dump, is stored in a temporary directory. Following the Open MPI common practice, the temporary directory is set to `/tmp`. However, a problem may arise if `/tmp` is mounted in the main memory (most common case) and the amount of memory available is not enough to store the dump. To address this issue, the user or the resource manager can specify a different directory.

4.3.3 Process State Migration

As previously described, the outcome of the CRIU checkpoint (or dump), i.e. the process dump, is a collection of files. To simplify the transfer of such files over the network, the next step is to create an archive containing such files and optionally compress them. For brevity we are going to call the checkpoint archive ‘’image”.

Generally, the decision of compressing or not the archive requires the evaluation of the trade-off between compression time and transfer time savings due to compression. This task can be in charge of the resource manager, which should consider several factors, e.g. network bandwidth, network traffic, image size, shared memory occupation and disk performance.

The image is now ready to be moved to the destination node. CRIU has a server functionality that allows the *disk-less* transfer the image. However, at time of development this feature was not very mature and the C API relative to the server is not complete. Therefore, we decided to do not use the integrated CRIU server.

The transfer can be achieved according to two strategies:

1. using a TCP connection between source and destination nodes;

2. using a Network File System (NFS).

At the time of writing, we do not have a storage unit with NFS available for testing, therefore we selected the TCP based option to transfer the image between nodes.

4.3.4 CRIU Restore (Restart)

When the `orted-restore` daemon running on the destination node receives – and possibly decompresses – the image coming from the source node, it restarts the ORTE daemon and its children processes using the CRIU API. Since the C/R approach of CRIU is totally transparent to the (frozen) processes, after the restart we need to send a signal to the restored ORTE daemon to advise it that a node migration has occurred. Accordingly, the ORTE daemon reopens the connection to the Head Node Process and sends the `MIGRATION_DONE` message.

Finally, the Head Node Process broadcasts the `MIGRATION_DONE` message to all the other ORTE daemons and processes using the same synchronization protocol previously described.

4.3.5 Finalization Stage

When all the processes have received the `MIGRATION_DONE` message, the migration procedure enters the *Finalization stage*. In this phase, to minimize the overhead, the migrated processes reopen the connections towards other processes only if needed. This happens if there are packets waiting in the buffer or if the application has new data to transmit.

Once again, it is worth underlying that the entire migration procedure is performed without the awareness of the application, which only experiences a network delay in the communication towards migrating processes.

Moreover, the performance degradation is additionally mitigated by having all the nodes not involved in the migration still communicating between each other. In such a way, we avoid using complex algorithms to reach a global consistent state. This is another key advantage of our solution, since in application-level C/R schemes the coordination phase presents several problems on both user and framework sides [52].

4.4 The btl TCP component

This section shows the details on how the coordination between MPI processes is performed, when they are using the tcp component of btl framework. The btl framework is used exclusively by the MPI processes to communicate with other processes in the application universe.

The tcp component exposes several functions used by other Open MPI frameworks to send/receive messages and to open/close the communication towards processes. The functions are always non-blocking: a callback function is provided by the caller and it is called when the requested operation terminates successful.

4.4.1 The TCP endpoints

Each process is identified by one **endpoint**, i.e. a data structure containing several information, like the socket file descriptor, the high-level reference of the process, the connection status, etc. The *endpoint* contains also the data fragments waiting to be processed in both directions. A process has only one valid endpoint towards the same process. The *endpoint* may be in these status:

- **Closed**: the endpoint is closed and there is no active attempt to connect to the process. Typically this is the case when no messages are sent towards that process yet.
- **Connecting**: the connect system call is trying to contact the receiver.
- **Connect Ack**: the connect system call has successful established the connection towards the process, however it's waiting the confirmation *ack* message. Open MPI implements a three-way handshake-like protocol.
- **Connected**: the node is connected and ready to send/receive messages.

- **Failed**: an error has occurred and the socket is no more available. Usually this condition leads to an unstable condition that in turn leads to the application crash.

In order to perform the migration, we add the **Frozen** to the possible endpoint statuses. This status is used to identify the *endpoint* condition towards the migrating processes or, if it is the migrating process itself, the conditions of all others *endpoint* not in the local *ORTE Daemon*.

The *Frozen* status is used to provide a sort of "lock" on the endpoint. Next calls to send, receive and similar functions stacks the outcoming messages to a queue of fragments to be sent. This queue will be flushed only when the migration has occurred and the processes resumed. The send caller does not notice any difference with respect to a normal send, but only a long delay on the execution of the callback function.

4.4.2 Freezing the connections before migration

The `base` component of `btl` framework provides the interface with the *ORTE Daemon*. At present, it uses a sequence of signals to coordinate with the daemon. Then, it calls the `mig_event` functions exposed by the active `btl` components, in our case the `tcp` component.

The `tcp` component will perform the freeze of all involved endpoints (endpoints of migrating processes if it is a non-migrating process, all endpoints of non-migrating processes if it a migrating processes). Then it will send back to the `base` component the confirmation, which will be forwarded to the *ORTE Daemon*.

Freezing the *endpoints* entails also ensuring that no in-flight messages are present. In order to ensure this, each process initially closes only the trasmission-side of the socket. Subsequently, it waits that the other side closes its trasmission-side, in this way the `recv` system call will fail. Then the `recv` system call fails, the socket can be safely closed and the migration ack can be sent back to the `base` component.

4.4.3 Restoring the connections after migration

When the migration ends, the `base` component calls again the `mig_event` functions. The statuses of frozen *endpoints* are changed to *Closed* and the connections are restarted only if there are fragments waiting in the queue. If the queue associated to the *endpoint* is empty, the connection is not restored immediately, in order to limit the congestion due to the reconnection of other processes.

The connection will be re-established only when the first message arrives via the `send` call.

It may happen that two processes try to re-establish the connection simultaneously. In this case, the `tcp` component will recognize this scenario and close one of the two connections, actually using only one connection in a full-duplex mode.

4.5 Solving system resource conflicts on restart

The restart stage is not straightforward if it occurs on a node different from where the Checkpoint has been performed. Program executable, libraries and data files must be in fact present and identical in the destination node. Moreover, remote file-systems must be mounted and the process identification numbers (PIDs) must be available because the processes cannot change their PIDs after the restore. Given that, there is no guarantee about the fact that the PIDs of the migrating processes have not been already assigned on the destination node.

CRIU does not allow to migrate processes having opened device files, established network sockets with different machine, `O_DIRECT` pipes, etc. (the full list can be found in the CRIU documentation).

In order to solve all the above mentioned issues the `orted-restore` performs specific system calls in order to isolate the processes in different *Linux Namespaces*. The **Linux Namespaces** allow an application to fork children in a detached environment.

4.5.1 Linux Namespaces

Linux Namespaces is a Linux kernel feature that provides an abstraction of system resources, such that processes within a namespace run in a isolated environment. The changes to the system resources inside a namespace are visible only to other processes members of that namespace. They are described in the *Overview, conventions, and miscellaneous* (Section 7) of the Linux Programmer's Manual [53]. The available namespaces are:

- **CGroup** [54]: isolation of Control Groups, providing a different `/proc/self/cgroup` file for each namespace;
- **IPC**: POSIX message queues and other inter-process communication;
- **Network**: network devices, TCP/UDP ports, etc.;
- **Mount**: the mount points;

- **PID**: isolation of process identifiers;
- **User**: user and group identifier;
- **UTS**: UNIX Timesharing System, it isolates the hostname, domain name, etc.

In this work both the PID and the Mount namespaces are used.

The PID namespace, as already discussed, is necessary in order to guarantee that the Process Identifiers of checkpointed processes are available on the destination node. Instead, the Mount namespace is needed for more than one reason:

- Detaching the Mount namespace is a pre-condition for detaching the PID namespace, since the `/proc` directory has to be remounted;
- Open MPI uses the Linux `ttys` present in `/dev/` directory. These special files are created mounting the `devpts` pseudo-filesystem. To avoid conflicts with `ttys` already present in the system, the `devpts` has to be remounted, not for the entire system, but only to migrated processes;
- Any other mountpoint, e.g. *network filesystems*, may require the remount for the specific application.

4.5.2 Unsharing the context

To exploit the Linux Namespaces, the `<sched.h>` header provides the `unshare` system call, that allows a process to detach part of its execution context, associating it to other namespaces.

In our implementation, the `orted-restore` executes the following C call:

```
unshare(CLONE_NEWNS | CLONE_NEWPID)
```

where the `CLONE_NEWNS` flag detaches the mount namespace and the `CLONE_NEWPID` flag detaches the PID namespace; hence, `orted-restore` and its children share the new private namespaces.

The `orted-restore` daemon becomes then the `init` process (PID=1) in the new empty PID namespace of the destination node and can restart the application processes with the original PIDs from the source node. The isolated mount namespace is necessary to remount the `/proc` directory in order to match the new process identifier configuration. At this point, the *ORTE daemon* instance and its children can be safely restarted.

The full stack of system calls required to restart the checkpointed *ORTE daemon* is described in Figure 4.4.

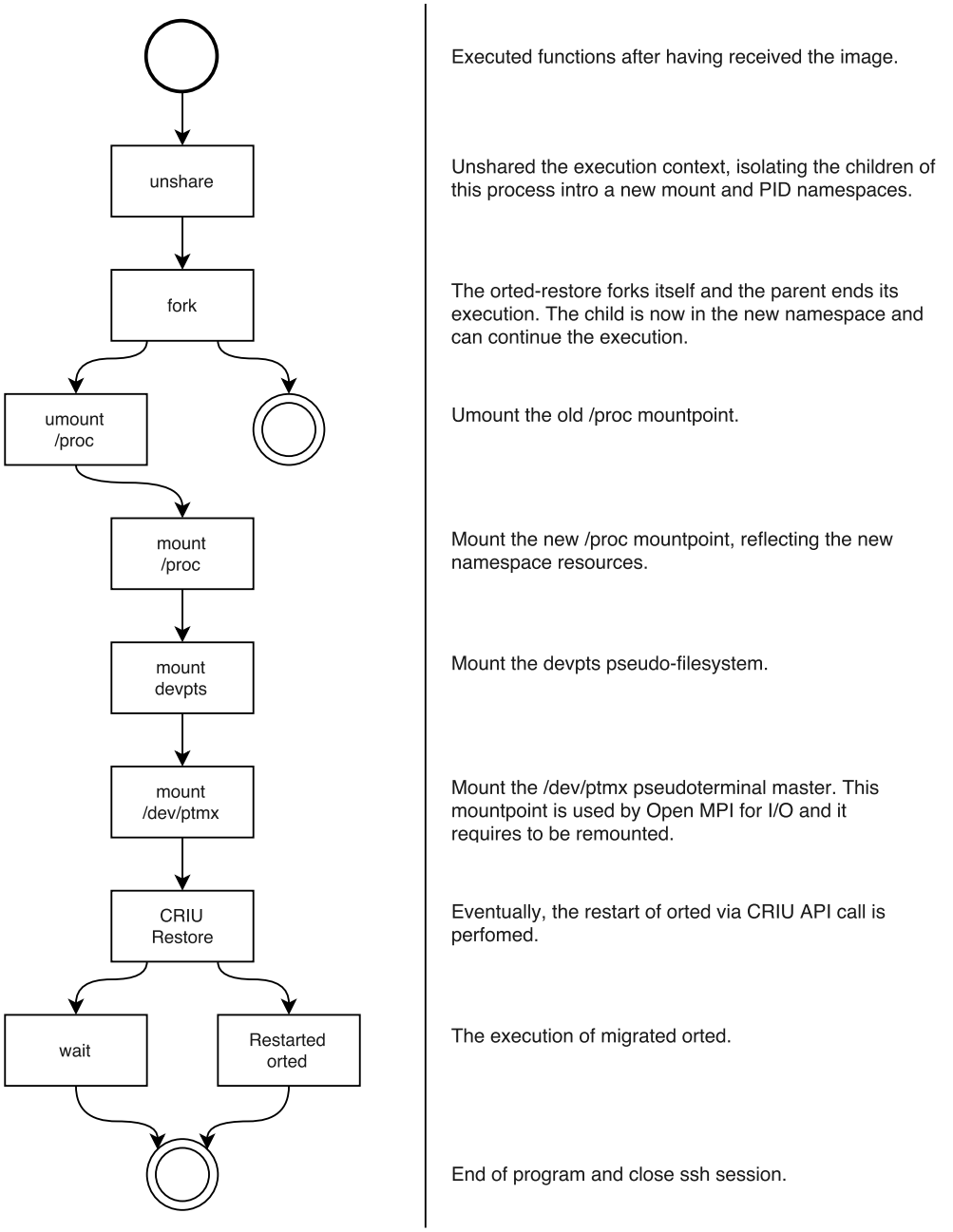

Figure 4.4: *Flow diagram of* `orted-restore` *execution.*

CHAPTER 5

Integration with the Barbeque Run-Time Resource Manager

In this chapter we present the integration of the `mig` framework with the Barbeque Run-Time Resource Manager [55], shorten BarbequeRTRM. The BarbequeRTRM is a framework to manage the resources of both heterogeneous and homogeneous systems at run-time. The extension of BarbequeRTRM that adds the capability to manage distributed systems is described. Subsequently, an example of exploitation of `mig` with a policy that takes into account resources located into remote systems is presented.

5.1 The BarbequeRTRM - Open MPI interface

Following the implementation of other resource managers in Open MPI, we decided to implement a socket client-server paradigm in order to interface the BarbequeRTRM to the Open MPI runtime and vice versa.

5.1.1 The `bbque-mpirun` launcher

As a first step, we introduced in the BarbequeRTRM a custom MPI application launcher, `bbque-mpirun`, acting as a wrapper of the well-known `mpirun` command commonly available in every MPI implementation. This is order to

enable the possibility of controlling the execution flow of MPI processes, by the BarbequeRTRM. The launcher exploits the already available BarbequeRTRM **Run-Time Library (RTLib)** API to communicate with the BarbequeRTRM.

When invoked very first step of the `bbque-mpirun` launcher is to open a socket listening onto a specific TCP port and the spawning an instance of underlying `mpirun` launcher. When the `ras` component of BarbequeRTRM is selected, recognizing a previously set environment variable and loaded, it connects to the open socket of `bbque-mpirun`. At this point `mpirun` sends the nodes request (specifically the number of *slots*, i.e. the total number of processes to be spawned in each node). `bbque-mpirun` receives the request and it sends through to the BarbequeRTRM. Subsequently, the resource manager can select an available set of computing nodes, according to a policy, and send it to the `bbque-mpirun` as reply, that in turn is forwarded to `mpirun`.

It is worth to specify that `bbque-mpirun` is multi-threaded piece of software implemented in C++, which operates as shown in the UML Sequence Diagram in Figure 5.1:

- The main thread follows the normal BarbequeRTRM applications execution, exchanging data like performance measurements and reconfiguration (not yet implemented). In case the BarbequeRTRM changes the allocation, it is in charge of notifying it to the `mpirun` instance;

- A second thread is waiting on the TCP socket connected with `mpirun`. Currently only diagnostic messages for migration are sent over that channel. This thread is implemented in the `CommandManager` class.

- A third thread is in charge to check the abnormal termination of `mpirun` process before the establishment of the socket connection. This thread is implemented in the `ProcessChecker` class.

As already introduced, `bbque-mpirun` may be considered as a wrapper of `mpirun` command, that establishes a link between BarbequeRTRM and the Open MPI framework. This approach is similar to the one adopted by *SLURM* resource manager.

5.2 Towards the BarbequeRTRM distributed version

BarbequeRTRM was designed to be a run-time resource manager targeting single node systems. In parallel with `mig` development, this work is the first step towards the development of a distributed version of the BarbequeRTRM.

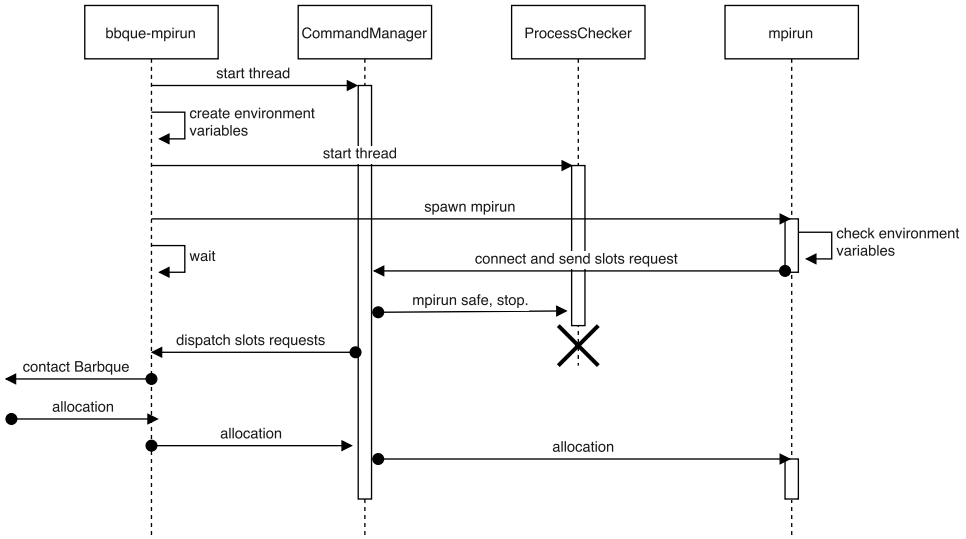

Figure 5.1: *The UML Sequence Diagram of the* `bbque-mpirun` *tool.*

In order to correctly manage MPI applications and to select an appropriate allocation over the distributed nodes, the BarbequeRTRM has to know how many nodes are in the systems and how they are characterized.

Calculate the optimal processes allocation over the nodes and future rescheduling – thus the evaluation of a favorable migration – are duties of an appropriate policy, which is introduced in the next section.

In BarbequeRTRM the available resource retrieval and the fulfillment of a scheduling are a task of the `PlatformProxy` component. The `PlatformProxy` interface was previously implemented by a system-specific subclass, e.g. the Linux and OpenCL implementations. We have redesigned the `PlatformProxy` in a more hierarchical fashion, adding a `LocalPlatformProxy` and a `RemotePlatformProxy` component. The former has the duty of providing a transparent layer to the resources, like local CPU cores and OpenCL devices, while the latter is in charge of communicating with available remote BarbequeRTRM instances, if any. The conceptual diagram of the BarbequeRTRM components that allows to work in a distributed environment is shown in Figure 5.2. As shown

Since we are in the early stages of the design of a distributed version of BarbequeRTRM, the described design may change in next months. Furthermore, the information needed by the scheduling policy – subsequently passed to `bbque-mpirun` – is currently retrieved via `RemotePlatformProxy` that in turn reads a static XML file, containing the information of remote nodes.

The platform manager is in charge of abstracting the resources to the upper-level components. The `PlatformProxy` interface is implemented by the

`Local` and `Remote` proxies, that in turn abstract the resource in the local machine and in the remote machines. The `LocalPlatformProxy` dispatches the function call to the appropriate `PlatformProxy` depending on which type of resource is involved. For example, in multi-GPU systems, the `OpenCLPlatformProxy` that in turn wrap the OpenCL runtime, is called to enforce the assignment of a specific GPU to the application. In Linux, the Control Groups (cgroups) framework is used to enforce the resource assignments related to the CPU cores and memory [56].

On the other hand, the `RemotePlatformProxy` contacts the underlying network components to communicate with remote BarbequeRTRM instances. At present, this is an in progress development.

Finally, the `DistributedManager` is in charge of managing and coordinating the cluster of BarbequeRTRM instances. Typical duties of this class are:

- Manage the system topology and hierarchy;
- Discover new available systems in the network;
- Periodically retrieve the status and the runtime statistics of the BarbequeRTRM instances;
- React to nodes failures and network partitions with suitable fault-tolerant mechanisms and protocols.

The `PlatformManager` class has been also added in order to abstract the previously described underlying classes with respect to other BarbequeRTRM components.

5.3 DistRib policy

The **DistRib** policy implemented in BarbequeRTRM is based on a **Integer Linear Programming (ILP)** solver. Since the main objective of this thesis is not to find an advanced policy, but to implement a first exploitation of the `mig` framework, a classical ILP is proposed. The proposed policy provides an optimal solution, but it does not scale efficiently.

5.3.1 Optimization problem

The goal of the ILP problem is to find a performance-optimal mapping between systems and applications. Please note that this mapping is not necessarily one-to-one: a system may have more than one applications running and an application

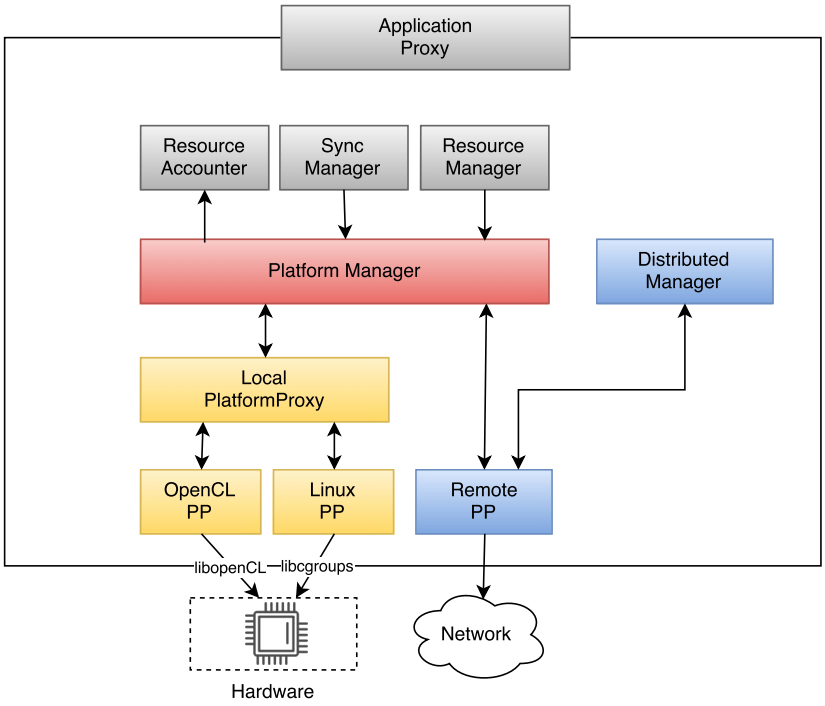

Figure 5.2: *The conceptual diagram of BarbequeRTRM distributed components.*

may run on a different systems. The latter situation is more typical due to the nature of HPC applications.

Mapping the application onto different systems introduce a significant overhead due to the network communications between different nodes. Even if InfiniBand is considered, the intra-host communication with shared memory is certainly faster [57]. In addition to this overhead, heterogeneous systems may have different performance that must be taken in account.

Another cost contribution that we can consider in the resource allocation or scheduling policy outcome is the **oversubscription**:

Definition 1. *An **oversubscribed** system is a system where the number of assigned MPI processes is greater than the number of available processing cores (or elements).*

This cost may have a strong impact on performance and Open MPI must be therefore aware of oversubscription.

According to Open MPI implementation, for each MPI process usually 100% of CPU core is reserved, even if not really used, in order to speed-up the communication performance. The idea is to avoid as much as possible the overhead due to the context switches performed by the operating system. However, this technique produces very bad effects in case of an oversubscribed system, especially if Open MPI is not aware of oversubscription. We do not delve deeper into

the analysis of oversubscription, since our resource manager does never oversubscribe systems.

Therefore, the cost of oversubscription is proposed in subsequent optimization problem for completeness, but at present not implemented in the policy.

The ILP formulation proposed is:

$$\min \sum_{a \in A} \sum_{s \in S} \left[p_a \cdot (C^{\text{sys}}(a,s) + C^{\text{dist}}(a,s) + C^{\text{os}}(a,s)) \right] \tag{5.1}$$

where A is the set of all applications to be scheduled and S is the set of all available systems.

Let for each application a:

- $\pi(a,s)$: an integer positive variable that identifies the number of processes of application a assigned to system s;
- $\phi(a,s)$: a binary variable that values 1 if the application a is assigned to system s, 0 otherwise;
- p_a: the priority of the application a;
- r_s: the penalty to use the system s (see later for details);
- K_d: a constant used as weight coefficient.

The variables are π and ϕ, the parameters are p_a and r_s and the only constant is K_d.

Obviously, π and ϕ are related and ϕ can be written as:

$$\phi(a,s) = \begin{cases} 1, & \text{if } \pi(a,s) > 0 \\ 0, & \text{else} \end{cases}$$

$$C^{\text{sys}}(a,s) = r_s \cdot \pi(a,s) \tag{5.2}$$

$$C^{\text{dist}}(a,s) = K_d \cdot \phi(a,s) \tag{5.3}$$

The r_s penalty is useful when the systems are heterogeneous from the point of view of the delivered performance: slower systems may have a high value of b_s, faster system a lower one. If the systems are homogeneous, r_s can be set to 0 and consequently $C^{\text{sys}} = 0 \quad \forall a, s$.

In our implementation, the term C^{os} is not currently present. This limitation is not really significant in HPC environments, since a system oversubscription generally leads to a dramatically drop of performance.

The formulation of the optimization problem in GNU MathProg is presented in Appendix B.

5.3.2 The selection algorithm

The ILP problem may not be feasible, so the solver would not be able to find a valid solution. The typical case is when the number of cores requested by all applications is greater than the available ones. To overcome to this problem, this algorithm is implemented:

1. Fill the system and the application vectors with all available resources and ready applications;
2. Try to solve the ILP problem;
3. If OK, enforces the solution;
4. If FAIL, remove the application with less priority and return to (2).

Obviously, the **starvation** of applications with less priorities may occur. In order to address this issue we implemented an *aging scheduling*. Every time that an application is deselected from the scheduling vector its priority is increased. Therefore, next time the policy runs the non-scheduled applications have higher probability to be scheduled.

CHAPTER 6

Experimental Evaluation

This chapter presents the results of a set of experimental tests performed to evaluate the overheads introduced by the exploitation of the proposed process migration mechanism.

The overheads may be categorized in two types: performance loss due to the execution of multiple *ORTE daemons* on the same node and the time required to actually perform the process migration.

For each performance test, in this Chapter we provide the description of the software used, the hardware setup, the methodology applied to perform the measures and finally the results obtained.

6.1 Introduction

As we have discussed in Subsection 4.1.1, the execution of multiple *ORTE daemons* on the same node enables the migration of a subset of processes instead of migrating the entire population of processes running on the same node. This ensures higher flexibility both for the system resource management and the wide range of use cases that can benefit from a task migration mechanism. The advantages have been previously described in Subsection 4.1.1. Unfortunately, there are also drawbacks, in terms of overheads impact on the application execu-

tion. Splitting the control of the application processes lifecycle under different *ORTE daemons* in facts, implies that some processes cannot rely anymore on shared memory to communicate with each other, even if they are running on the same node. In such cases, we need to use TCP/IP connections for inter-process communication, which have been shown to be less performing than shared memory [57].

In Section 6.3 instead, the migration overhead is presented. It represents the time lost to complete all the migration procedure described in Chapter 4. Therefore, opposite to the multiple *ORTE daemons* overhead, the migration overhead is not a persistent overhead. It impacts on the application execution time only if the migration is triggered.

Finally, in Section 6.4 we show the validation of the DistRib policy of the BarbequeRTRM discussed in Section 5.3.

6.1.1 The NAS Parallel Benchmark Suite

The **NAS Parallel Benchmark Suite (NPB)** [58] is a benchmark suite developed by the *NASA Advanced Supercomputing Division* since 90s. This set of applications is designed to evaluate the performance of supercomputers that exploits the parallel computation. The benchmarks are implemented with several technologies, such as *MPI*, *OpenMP*, *Java*, *HPF*[1], Globus. However, only the *MPI* and *OpenMP* implementations are continuously developed and updated. They do not depend on a particular implementation of *MPI* and *OpenMP*.

The NPB suite is widespread acknowledged in research, therefore we used it for testing both the multiple *ORTE daemons* overhead and the migration overhead. We selected the original eight benchmarks specified in NPB version 1, composed of five kernels and three pseudo applications. NPB version 2 and 3 introduced also other special benchmarks:

- NPB-MZ: the multi-zone version of the original pseudo-applications;
- The unstructured computation benchmarks;
- The parallel I/O benchmarks for specific techniques;
- GridNPB to test the performance of computational grids.

Since these specific benchmarks are not significant for our migration mechanism, we decided to use only a subset of the original benchmarks.

[1] High Performance Fortran is an extension of Fortran 90 for parallel applications.

The original five kernels are focused on testing the performance on particular metrics, instead the pseudo-applications are common abstractions of real problems solvable with MPI frameworks:

- **Kernels**
  - *Integer Sort* (`IS`): parallel sort of uniform distributed integer keys. It provides a balanced benchmark on integer computation, random memory and network communication speeds.
  - *Embarrassingly Parallel* (`EP`): generation of pairs of Gaussian deviates. It provides an estimation of floating-point performance upper-limit. The inter-process communication is minimal and non-significant.
  - *Conjugate Gradient* (`CG`): application of the Inverse Power Method to find the largest eigenvalue of a sparse randomized matrix. It tests the performance of irregular memory accesses and irregular long-distance communications.
  - *Multi-Grid* (`MG`): an ad-hoc multigrid algorithm to solve the Poisson Problem $\nabla^2 u = v$. It is a good representation of the performance of long- and short-distance communication and memory intensive operations.
  - *Fourier Transform* (`FT`): solving a Partial Differential Equation using forward and inverse Fast Fourier Transformations. It is a rigorous test of long-distance all-to-all communication performance.
- **Pseudo-Applications**
  - *Block Tri-diagonal solver* (`BT`)
  - *Scalar Penta-diagonal solver* (`SP`)
  - *Lower-Upper Gauss-Seidel solver* (`LU`)

Every kernels have integrated verification tests. These tests may be *partial verification* – if they are conducted on partial results – or *full verification* – if they are conducted after the combination of all partial results. As a consequence, we can also verify if the migration procedure has corrupted the application execution.

The benchmarks use a dedicated Pseudorandom Number Generator in order to generate uniform distributed pseudorandom numbers. This algorithm ensures sufficient randomicity, independently on the system used for executing the suite, provided some system constraints are met. However, most of the systems in use today meet these requirements.

	Class						
Bench.	**S**	**W**	**A**	**B**	**C**	**D**	**E**
IS	512 KB	8 MB	64 MB	256 MB	1 GB	16.4 GB	N.A.
EP	128 MB	256 MB	2 GB	8 GB	32 GB	512 GB	8 TB
CG	76.6 KB	437 KB	1.2 MB	7.4 MB	172 MB	240 MB	1.7 GB
MG	256 KB	16 MB	128 MB	128 MB	1 GB	8.1 GB	64 GB
FT	2 MB	4 MB	64 MB	256 MB	1 GB	16 GB	128 GB
BT	13.5 KB	108 KB	2 MB	8 MB	32 MB	518 MB	8 GB
LU	13.5 KB	280 KB	2 MB	8 MB	32 MB	518 MB	8 GB
SP	13.5 KB	364 KB	2 MB	8 MB	32 MB	518 MB	8 GB

Table 6.1: *Problem data sizes for each class and benchmark in NPB suite [59].*

Furthermore, each problem has multiple **classes** to be selected, where a class represents the input data size, summarized in Table 6.1.

6.2 `mig`: ORTE daemons granularity overhead

Hardware setup

We used a computing node equipped with two Intel Xeon E5-2640 octa-core hyper-threaded CPUs, with 128GB of RAM per CPU. The system is based on a NUMA architecture.

As common in HPC environments, we disabled Hyper-Threading, remaining with 16 cores at our disposal. The running operating system was CentOS 6.7 with updated Linux kernel version 3.18.

Methodology

We selected `IS`, `MG` kernels and `BT`, `SP`, `LU` pseudo-applications. We discarded the other kernels since their execution would not be affected significantly by a run-time migration of some processes:

- `EP` is not a good candidate for testing communication, since it does not almost send anything.
- `CG` tests long distance communication and our overhead is present only in short distance (even same machine).
- `FT` as for `CG` it tests only long distance and not short.

The kernels were executed specifying input classes `B`, `C`, `D`. For the pseudo-applications, instead, we did not consider class `D` since the completion of the execution would require too much time using the systems at our disposal. Finally,

we excluded classes `S`, `W`, `A` because the problem size would have been too small, leading to very short executions. Vice versa, `E` class would be too expensive in terms of memory and execution time.

We executed the pairs `(benchmark, class)` spawning 16 processes for each benchmark execution. We selected the number of *ORTE daemons* controlling the MPI processes according to different granularities: 1, 2, 4, 8, 16. Each *ORTE daemon* instance had therefore to manage 16, 8, 4, 2 or 1 MPI processes respectively. In particular, the case of single *ORTE daemon* instance is a standard execution of unmodified Open MPI and we took it as a reference result in the overhead evaluation.

We measured the execution time of each tuple `(benchmark, class, granularity)`, starting after the `MPI_Init` call and stopping before the `MPI_Finalize` call. The time needed to spawn the *ORTE daemons* and the processes are therefore not considered. We repeated the test 20 times to obtain a significant statistics. It turned out that we experienced an average standard deviation below 1% of the total execution time.

Results

The overall results are shown in Figure 6.1. The global trend is that the overhead increases sub-linearly with respect to the number of *ORTE daemon* instances, while it decreases as the problem size increases. The sub-linear increase of the overhead can be explained by the fact that, once there are at least two *ORTE daemon* instances, the TCP/IP communication between MPI processes on different instances becomes the bottleneck for communication latencies. Adding more *ORTE daemon* instances to the existing ones does not tend to further degrade performance.

Conversely, the decrease of the overhead in case of increasing problem size is due to the fact that increasing problem size means that more time is spent on computing data; therefore the time spent in communication – which is where the overhead applies – decreases in percentage.

Looking at the data summarized in Table 6.2, we can state that the *ORTE daemons* granularity poorly affects the application execution time. Considering all the test cases, we can observe indeed that the percentage of time loss remains in the $0 - 5.x\%$ range.

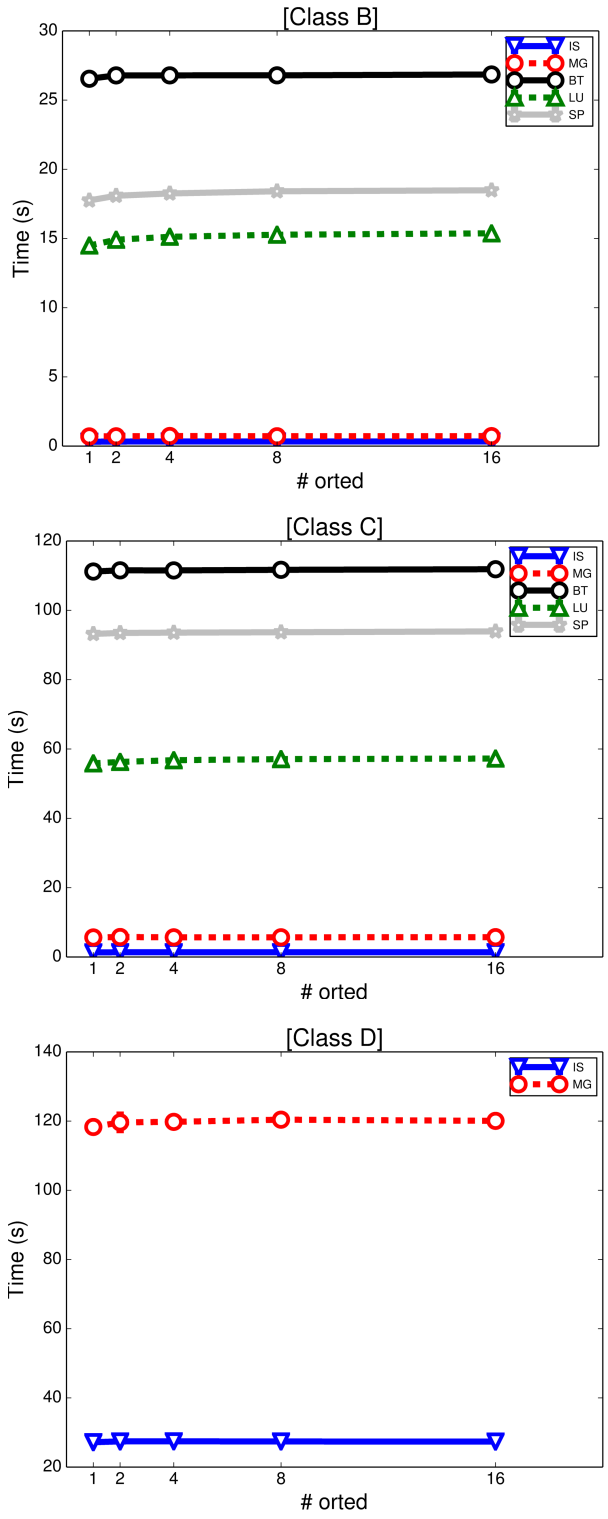

Figure 6.1: *Execution time of each benchmark when running 16 processes using a number of* ORTE daemons *that ranges from 1 to 16.*

Benchmark	Class	# orted	Overhead %
IS	B	2	4.68
		4	5.18
		8	4.85
		16	4.85
	C	2	2.21
		4	2.68
		8	2.64
		16	2.64
	D	2	0.87
		4	0.96
		8	0.79
		16	0.64
MG	B	2	1.28
		4	4.36
		8	1.73
		16	3.24
	C	2	2.25
		4	0.62
		8	0.88
		16	1.57
	D	2	1.13
		4	1.25
		8	1.79
		16	1.50
BT	B	2	0.92
		4	0.93
		8	0.93
		16	1.17
	C	2	0.31
		4	0.29
		8	0.45
		16	0.60
SP	B	2	1.90
		4	2.83
		8	3.72
		16	4.12
	C	2	0.31
		4	0.40
		8	0.52
		16	0.79
LU	B	2	2.92
		4	4.37
		8	5.42
		16	6.10
	C	2	0.73
		4	1.63
		8	2.19
		16	2.48

Table 6.2: *Static overhead of* `IS`*,* `MG`*,* `BT`*,* `SP`*, and* `LU` *with increasing migration granularity, i.e. increasing number of* ORTE daemons*, compared with single* ORTE daemon *case.*

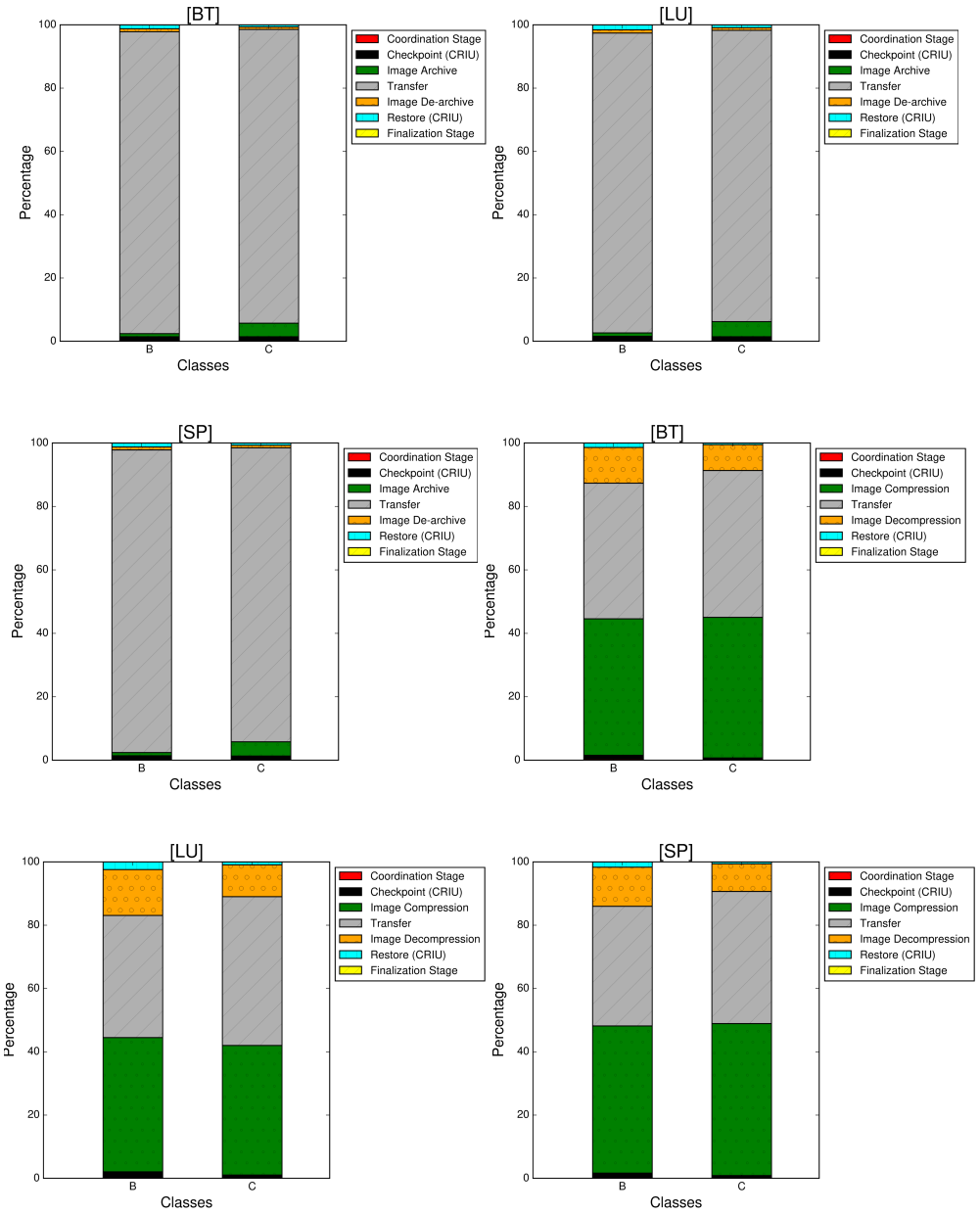

Figure 6.2: *Process migration time composition with respect to the input problem size. Top images: migrations without image compression. Bottom figures: migration with image compression.*

6.3 mig: ORTE daemons migration overhead

Hardware setup

We characterized the migration overhead by running the benchmarks on two nodes connected via Gigabit Ethernet, both equipped with two Intel Xeon E5-2640 CPU, 128GB of RAM per CPU (NUMA) and keeping the Hyper-Threading disabled.

Methodology

In the experimental migration scenario, we launched two *ORTE daemon* instances per node, each one managing 4 out of the 16 application processes. The resource manager triggered the migration requests after 25 seconds of execution.

To better observe the composition of the migration overhead, we divided the migration time, isolating seven contributions. We considered the time required by each of the five migration phases previously described in 4.3, plus two additional contributions:

1. Time required to encapsulate the CRIU process dump into the archive;
2. Time to extract the dump from the archive after the migration.

With the exception of the *Coordination* and the *Finalization* phases, the other contributions are expected to be strongly dependent on the problem size, in particular on the image transfer time. In this regard, we evaluated the possibility of introducing the compression of the image generated by the CRIU checkpoint before proceeding with the transfer. In such a case, a decompression step is obviously required on the destination node, before resuming the execution of the processes. To this purpose, we used the GZIP compression algorithm [60].

Results

In Figure 6.3, we can see how the compression is effective in reducing the size of the checkpoint image to transfer. This because of the shared memory implementation. Open MPI in facts allocates over 100MB of unused shared memory as ghost files initialized as zeros. The consequence is that compression is very effective in such cases, resulting in image sizes scaled down to $22 - 38\%$ with respect to the original sizes. In case of bigger datasets instead – like the C class – this phenomenon is less evident, with compressed image sizes resulting the $45 - 65\%$ of the original sizes.

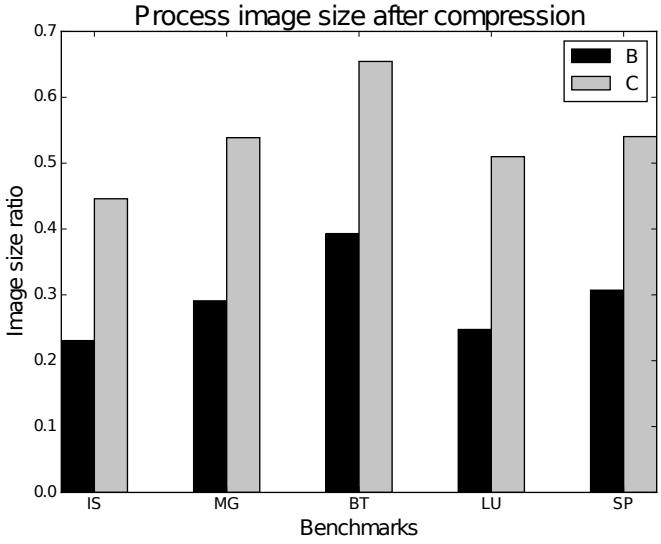

Figure 6.3: *Process checkpoint image after GZIP compression for each benchmark, with respect to the input data class*

The composition of the migration overhead is highlighted in Figure 6.2. If the compression is not applied (top figures), the time required to transfer the process image over the network dominates the whole migration process ($80 - 90\%$ of the time) independently of the benchmark and of the class of input data. The contributions due to the synchronization stages (*Coordination* and *Finalization*) and the time needed for doing checkpoint/restore with CRIU are instead negligible.

Conversely, when the compression is applied (bottom figures), the transfer time is reduced, but a new overhead contribution is introduced. The time spent to perform image compression/decompression is not negligible. We can observe indeed an overall percentage value comparable to the transfer time ($40 - 50\%$) for the compression, plus a $10 - 15\%$ for the decompression.

Figure 6.4 provides an overview of the measured migration times, comparing the cases with image compression against cases where no compression is applied. For the `IS` and `MG` benchmarks, where hundreds of MB of data must be moved, the compression represents a penalty. Conversely, for `BT`, `LU` and `SP`, where data size is a few MB, break-even points can be found. Generally, we can state that resource manager should be in charge of choosing whether apply compression or not, taking into account application properties, input data size, node capabilities and network parameters (e.g., topology, bandwidth, etc...). The `mig` framework is then driven accordingly.

Comments

Please note that these tests were performed on two computational nodes connected via Gigabit Ethernet. Most recent HPC systems connect nodes using InfiniBand, which is much faster than Ethernet. It follows that, in the average case, we do not expect compression to be needed, because transfer time will usually be in the range of seconds rather than of tens of seconds. In our experimental set-up, as shown in the results, the time required to interrupt the execution of a set of processes, to migrate them to another node and to resume their execution is not negligible and may require tens of seconds.

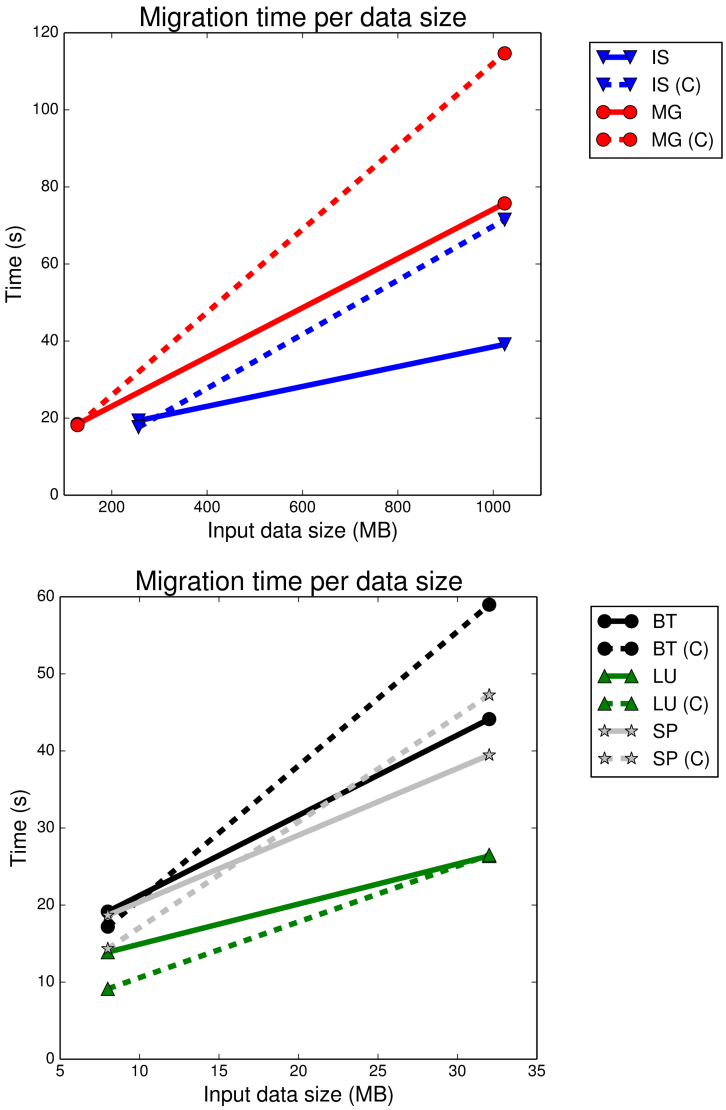

Figure 6.4: *`IS`, `MG` and `BT`, `SP`, `LU` migration overhead with respect to the input problem size. Migrations using image compression (dashed lines (C)) can be compared to migrations without image compression.*

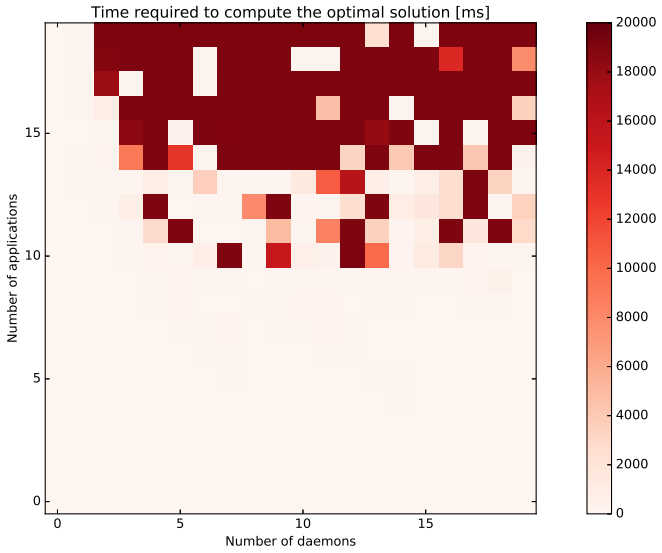

Figure 6.5: *Performance results of ILP problem with homogeneous systems (no per-system penalty).*

6.4 DistRib validation

6.4.1 The GLPK solver

In order to solve the ILP problem the **GNU Linear Programming Kit (GLPK)** [61] is used. GLPK implements Linear Programming (LP) and Mixed Integer Programming (MIP) solvers. The latter is a generalized case of ILP, since it allows the mix between integer and real variables. Since the textual formulation presented in Appendix B is adequate only for the first prototype of the ILP formulation, the actual code uses the API provided by GLPK itself, linking the relative shared library.

The algorithms used

GLPK implements several algorithms for the resolution. The policy uses the standard *Simplex algorithm* for solving the LP relaxation and then the *branch-and-cut* method [62] to obtain the integer solutions. The *branch-and-cut* method is a hybrid of classical *branch-and-bound* and *cutting plane* methods that assure an optimal solution with a good probability. GLPK allows the production of a sensitivity report in order to analyze the goodness of the solution, however at present this is not used in our software.

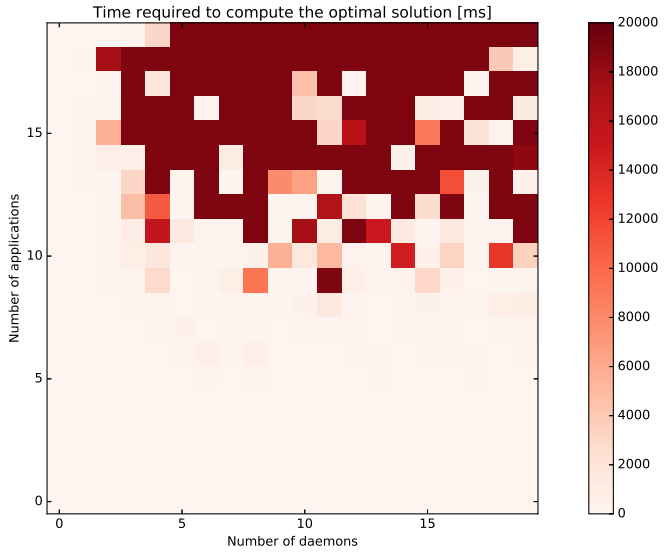

Figure 6.6: *Performance results of ILP problem with heterogeneous systems (per-system penalty presents).*

6.4.2 Performance

Integer Linear Programming is proved to be *NP-complete* [63], thus no polynomial-time algorithm is known.

We performed the tests timing GLPK solver. The instances are selected from 1 to 20 systems and from 1 to 20 applications, for a total of 400 instances. The number of cores per system and the number of cores per application are uniformly randomized. Each instances are executed 20 times and the resulting execution times have been averaged. A timeout at 20 seconds is placed, after that the execution is killed.

The first batch of tests are executed with $r_s = 0 \; \forall s \in S$ (homogeneous systems) and the results presented in Figure 6.5. The second batch of tests are executed with r_s as a random value from 1 to 10 uniformly distributed. This test implements the case of heterogeneous systems and its results presented in Figure 6.6. The two cases do not show any significant difference adding the addend of per-system penalty to the objective function.

However, GLPK performs well with our ILP programming for number of applications less than 10, and it seems not very sensitive to the number of systems. Isolated cases are very fast even with a high number of systems and applications. For example, *(15 applications, 17 systems)* is very fast case, but *(14 applications, 17 systems)* and *(16 applications, 17 systems)* are very slow. This behavior may be explained analyzing the optimization performed by GLPK, that can solve rapidly some special cases.

In a real HPC scenario with thousands of nodes and several applications a

greedy policy or other non optimal algorithm is required. The policy has to be sufficiently fast in order to do not introduce more overhead.

CHAPTER 7

Future Works and Conclusions

This final chapter provides an overview of the possible future works regarding process migration in HPC systems. Then, possible future developments of the `mig` framework are suggested, including the extension to iterative migration and InfiniBand. Finally, the current limitations of the proposed approach are discussed along with the conclusions of the overall work presented in this thesis.

7.1 Enhancing process migration in HPC

As depicted in Figure 7.1 the research interest in Process Migration targeting HPC systems is still high and it has a positive trend. This, in conjunction with the problematics described in Chapter 1, leads us to suppose that in the next years the *process migration* will become an hot research topic.

7.1.1 Next challenges

The CRIU project is rapidly evolving thanks to the very active development community. Since the project is relatively young, it is still unstable, especially for what concerns the advanced features. In the next years we expect this project to reach a good level of maturity, such it could be used also in production environments rather than only in research.

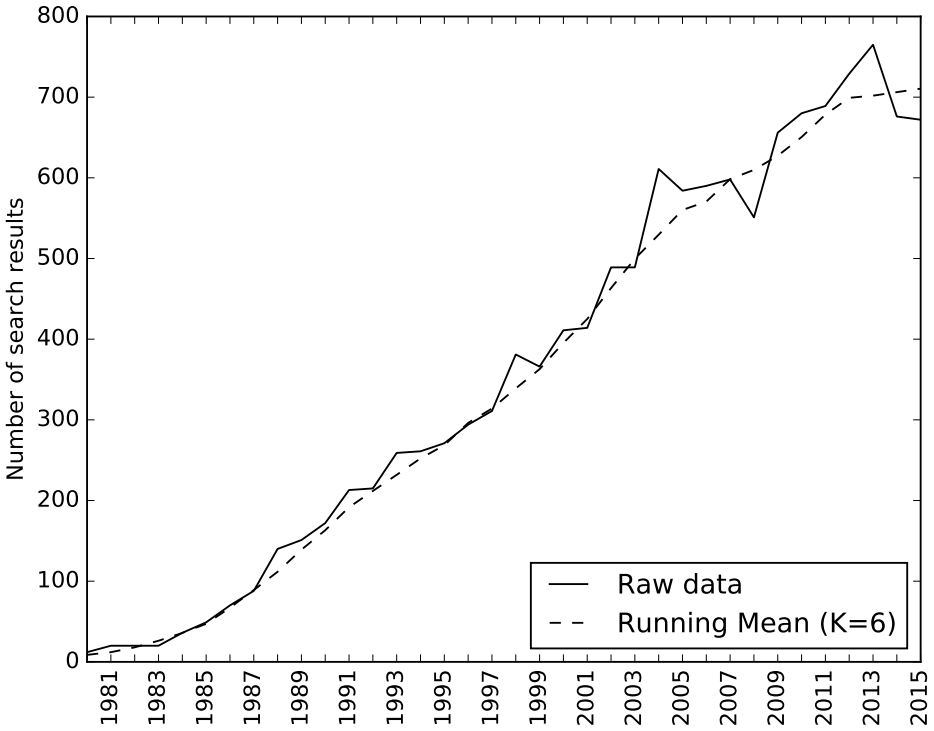

Figure 7.1: *Number of search results in Google Scholar with keyword "process migration"*

In parallel, new research challenges may arise and old issues may be addressed. An example of such a case is the possibility managing process migration over **heterogeneous processors**. The explosion of several heterogeneous architectures, from the Graphical Processing Units to specialized accelerators, requires the adoption of dedicated techniques in resource management and application frameworks. Therefore, low-level layer capable of transparently move processes between processors or nodes with different architectures can become an essential feature. Unfortunately, this type of migration is complex and implementing a working solution requires the involvement of several computer engineering branches, from operating systems to compilers. Previous research described in Chapter 2 provided tools with strong limitations and that work only for specific scenarios.

The first attempt that can be evaluated for an inclusion in CRIU is the migration between general purpose architectures with different ISA (Instruction Set Architecture), for instance to migrate a process between `x86_64` and `AArch64` architectures. The Checkpoint/Restart is already supported by CRIU for both architectures, but migrating processing between them is currently not possible.

This because, such a type of migration requires the translation of all the processors registers, the memory addresses and other machine-dependent information. Moreover, since there is no one-to-one correspondence between instructions with different ISA, some sort of checkpoint barriers must be implemented in the binary code. To understand the last problem consider the `MNEG` instruction of `ARMv8` (`AArch64`) having this semantic: $R_d = -R_n \cdot R_m$. This sort of specialized instruction does not exist in `x86_64` processors and must be split into a multiplication and a subtraction. Consequently, a migration mechanism cannot checkpoint the application between multiplication and subtraction, thus it should consider an atomic operation both of them. Managing this atomicity for different types of architecture seems not a straightforward task and it necessarily requires additional support at compiler-level.

The previously described possible improvement in process migration may solve also the problem of migrating processes between different Linux kernel versions or different libraries, obviously provided that they expose the same API. As already described in Chapter 4, CRIU requires perfectly identical environments.

7.1.2 Future developments

The `mig` framework is not currently ready for production environments. A sufficiently stable version has to be developed and possibly integrated in the mainline repository of Open MPI. The stabilization of the `mig` framework and related components requires extensive tests over different environments and system setups.

Iterative Migration

As we have seen, the transfer time dominates the migration time. This time becomes an overhead in the overall application execution time, since the processes cannot progress during the image migration. However, CRIU allows the **Iterative Migration** of processes: the processes are checkpointed without killing them and, during the processes execution, the image is transfered. As a result, the execution and image transfer time are overlapped, this operation is often called **Pre-Copy**. Obviously, the transfered image represents the status of the application at the checkpoint time. Therefore, it does not represent the current status of the migrating process and it must be updated with the process memory and the context changes (register, etc.).

In HPC environment, the *iterative migration* shows good performance results, significantly lowering the migration overhead [37]. Therefore it can be

considered a possible application improvement to introduce in the `mig` framework, in order to limit the impact on the wall-time.

InfiniBand

The InfiniBand support is the prioritized task to be accomplished in the development of `mig` framework. In HPC the InfiniBand networks are very frequent, because their advantages compared to Ethernet. InfiniBand presents lower overhead over Ethernet, because the applications are able to directly communicate with the network adapter without the necessity of operating system calls. Even considering the 10Gbit and 40Gbit Ethernet, InfiniBand FDR presents higher throughput and lower latency compared with Ethernet protocols [64]. Furthermore, in 2014 the new InfiniBand EDR doubles the theoretical throughput with respect to FDR.

Implementing the InfiniBand support should be a relatively easy task thanks to the high modularity of Open MPI. The functions offered by the TCP `btl` component have to be replicated in the InfiniBand `btl` component. The features provided by other modules – like `mig` – are independents from the network protocol used.

Removing the ORTE daemon granularity

A further reduction of the *ORTE daemon* overhead could be obtained by moving the migration granularity to process-level, directly migrating MPI application processes, instead of the underlying *ORTE daemon*. This requires to change the active BTL components of the migrating process. Whether *process A* and *process B* are in the same node communicating via *shared memory* and process B migrates towards another node, they have to change the active `btl` component to a remote one (e.g. TCP or InfinBand) in order to resume the connection to the other process. This change requires to address several synchronization issues.

7.1.3 Limitations

The major limits of the proposed process migration mechanism are in similar to those of the other C/R based systems: the nodes of HPC systems must be homogeneous, i.e. the operating system (with kernel version), libraries version and application binaries must be perfectly identical. Therefore, currently the `mig` framework is not exploitable in heterogeneous environments.

Performing the checkpoint with CRIU requires administration level permissions (`root` user in Linux) in all the nodes. This limitation is a partial requirement of CRIU that is in progress to be dropped by the CRIU development

team. However, as previously described, process migration requires the use of `unshare` system call, that in turn requires the `CAP_SYS_ADMIN` capability in Linux. This permission is usually granted only to the `root` user. Grant this permission to non-administrative users may introduce security problems that should be carefully evaluated and addressed.

In fault tolerant exploitation one of the most important issue to be addressed is the possibility of migrating also the *ORTE daemon* and consequently the MPI processes from the HNP. Otherwise, the HNP becomes the bottleneck in terms of fault-tolerance: the MTTF of the overall system is reduced to the MTTF of the HNP. Since the `mpirun` command is actually an *ORTE daemon*, it would not be difficult to implement the migration of HNP. We think the only needed change is the adaption of the coordination protocol between *ORTE daemons* and MPI processes.

7.2 Conclusion

In this thesis we introduced a novel approach to support process migration in the Open MPI framework. The approach is based on handling the execution of multiple *ORTE daemon* instances, which can be thought of as the smallest migratable unit. This is performed transparently to the application and the non-involved Open MPI frameworks and components.

Compared to other state-of-the-art solutions, one of the major advantages of our approach is the *maintainability*. The extension introduced in the Open MPI runtime in fact has a minimal impact on the other Open MPI frameworks. Furthermore, on the application side the framework does not introduce any additional API. Therefore its exploitation it does not require any change on the applications code.

Moreover, the `mig` framework does not rely on any virtualization layer. This constitutes a gain in terms of performance with respect to approaches based on virtual machine allocation. In this regard, our proposal allows us to perform fine-grained migrations, since the resource manager can decide between migrate an entire application or a subset of its processes. This feature also increases the controllability of the workload execution.

The integration with Barbeque Run-Time Resource Manager has been implemented in order to exploit both Open MPI and the `mig` migration mechanism in conjunction with a resource manager. In this regard, a simple policy that solves an Integer Linear Programming problem has been implemented as a BarbequeRTRM plugin. This policy provides an optimal solution in most cases but it is usable only with a small number of systems and applications. Therefore, in

the future a greedy policy must be implemented to work with large system, in particular considering the Exascale computing horizon.

Through experimental tests, we shown how the overhead due to grouping the application processes on top of several *ORTE daemons* can be considered negligible. Conversely, stopping and resuming the processes execution on different nodes, introduce an overhead dependent on the specific application, its input data size, the network and the node capabilities. As a consequence, a resource manager can play a key role to evaluate when a migration is worth to be performed.

Overall, we can state that the work presented in this thesis is the first process-level migration feature developed for Open MPI whose control is kept at system-level (resource manager) and that does not require the code of applications to be changed.

From the MPI communication standpoint, the lack of *InfiniBand* support is currently the most important missing feature. However the development of this component is currently ongoing.

In next years, we can expect an increasing interest in process migration and in general in C/R techniques. Therefore, the next steps of this work is to sufficiently debug the code in order to add it to the Open MPI mainline repository. In this way the `mig` framework may follow the rapid developments of Open MPI and CRIU and it can be available for other resource managers.

APPENDIX $\mathcal{A}$

Open MPI extra commands

This appendix summarizes the auxiliary commands of Open MPI not included in the MPI standard. The list is updated with commands present in version 1.10 of Open MPI.

The list of available commands as presented in the documentation **User Commands** (*man page* 1) are presented in Table A.1

Command	Description
`ompi-clean` `orte-clean`	It cleans up old files and processes left by previous Open MPI jobs in the local node. It kills the processes spawned by a previous application and remove all temporary files.
`ompi-ps` `orte-ps`	It shows the information about the active MPI jobs and processes. It works universally, i.e. it displays the information even if it is not the HNP.
`ompi-server` `orte-server`	It activates a server that can be contacted via the MPI calls *Publish_name*/*Lookup_name*.
`ompi-top` `orte-top`	It displays the information similar to `top` Linux utility.
`ompi-info` `orte-info`	It provides the information about the compilation and the installation of Open MPI. This utility is very useful for both Open MPI developers and users, in order to check how Open MPI is configurated and which modules are available.
`opal_wrapper`	Should not be called directly. A wrapper executable for `mpicc`, `mpic++`, etc.
`orte-dvm`	It starts an `orted` for each node before submit a job. The jobs may be submitted next via the `orte-submit` command. It is useful for launching numerous short applications.
`orte-submit`	It submits a job into distributed daemons spawned with `orte-dvm`.
`orted`	The daemon spawned on each node.

Table A.1: *Open MPI User commands.*

APPENDIX B

DistRib ILP formulation

This appendix provides a formulation in **GNU MathProg** of the proposed Integer Linear Programming problem used in the DistRib policy.

GNU MathProg is a high-level language to write mathematical models, in particular optimization problems. It is specific to GLPK, but compatible with the well-known **A Mathematical Programming Language (AMPL)** [65]. To be precise, the GNU MathProf contains a subset of instruction of the AMPL language.

The proposed formulation is implemented in BarbequeRTRM using the GLPK libraries even if less intuitive than the GNU MathProg language. However, the overhead drastically reduces, since a syntax parser of the model is not required.

The GNU MathProg formulation is presented in Listing B.1.

The proposed ILP has two variables, the `proc_assigned` and the `is_assigned` variables respectively correspond to the π and ϕ of the mathematical model presented in Chapter 5.

The parameters used in the formulation are:

- `num_pe`: the vector containing the number of available processor elements per core.
- `num_proc`: the vector containing the number of processes requested per

application.

- `priority`: the vector parameter that represents p_a, i.e. the priority for each application.
- `sys_penalty`: the vector parameter that represents r_s, i.e. an empirical number for per-system penalty.
- `K_dist`: the constant weight to the cost related to distribution
- `max_pe_per_system`: an arbitrarily number sufficient big to guarantee the constraints between π and ϕ (details later).

After the definition of the cost objective function, there are four constraints:

- `full_assignment`: it assures that all the processes request of each application is respected;
- `resource_availability`: it assures that the number of processes assigned to each systems does not exceed the available processing elements;
- `var_association`: it bound the π and ϕ variables in sense that if $\phi = 1$ then $\pi \geq 1$.
- `var_association_rev`: the reverse bound between π and ϕ, if $\pi \geq 1$ then $\phi = 1$.

```
set SYSTEMS;
set APPLICATIONS;

/*
 * The only two necessary variables. The first one is not used in the
     cost
 * function but it's required for proper resource assignment.
 */
var proc_assigned{i in APPLICATIONS, j in SYSTEMS}, >=0, integer;
var is_assigned  {i in APPLICATIONS, j in SYSTEMS}, >=0, binary;

/*
 * The various constant parameters.
 */
param num_pe     {j in SYSTEMS};
param num_proc   {i in APPLICATIONS};
param priority   {i in APPLICATIONS};
param sys_penalty{j in SYSTEMS};

param K_dist;
param max_pe_per_system; /* required for the constaints of binary
    variable,
                            it may be set to 1000000 or similar */

/*
 * The minimization of the cost function.
 */
minimize cost: sum{i in APPLICATIONS} sum{j in SYSTEMS}
                    ( priority[i] *
                    (
                      proc_assigned[i,j] * sys_penalty[j]  +
                      K_dist * is_assigned[i,j]
                    ));

/*
 * Constraints.
 */
s.t. full_assignment{i in APPLICATIONS}:
    sum{j in SYSTEMS} proc_assigned[i,j] >= num_proc[i,j];

s.t. resource_availability{j in SYSTEMS}:
    sum{i in APPLICATIONS} proc_assigned[i,j] <= num_pe[j];

s.t. var_association{i in APPLICATIONS, j in SYSTEMS}:
    is_assigned[i,j] <= proc_assigned[i,j];

s.t. var_association_rev{i in APPLICATIONS, j in SYSTEMS}:
    max_pe_per_system * is_assigned[i,j] >= proc_assigned[i,j];
```

Listing B.1: *The MathProg formulation of DistRib ILP solver.*

Bibliography

[1] Thierry Van der Pyl. The European HPC strategy and actions in Horizon 2020. Technical report, European Commission, February 2014.

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[3] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[4] Oreste Villa, Daniel R. Johnson, Mike O'Connor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, Stephen W. Keckler, and William J. Dally. Scaling the power wall: A path to exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 830–841, Piscataway, NJ, USA, 2014. IEEE Press.

[5] Marc Snir, William D Gropp, and Peter Kogge. Exascale research: preparing for the post-moore era. 2011.

[6] ASCAC Subcommittee Report. Top Ten Exascale Research Challenges. Technical report, Department of Energy, February 2014.

[7] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015.

[8] Y. Arakawa, T. Nakamura, Y. Urino, and T. Fujita. Silicon photonics for next generation system integration platform. *IEEE Communications Magazine*, 51(3):72–77, March 2013.

[9] X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Transactions on Computers*, 61(6):767–779, June 2012.

[10] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[11] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[12] Intel xeon processor e7 family reaches reliability parity with risc/unix, delivers 99.999Technical report, INFORMATION TECHNOLOGY INTELLIGENCE CONSULTING, East Lansing, Michigan, 2013.

[13] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, page 1094342014522573, 2014.

[14] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[15] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, July 2015.

[16] Miao Chen, Fang Dong, and Junzhou Luo. Dynamic resource management in a hpc and cloud hybrid environment. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 206–215. Springer, 2013.

[17] José María Cela, Philippe OA Navaux, Alvaro LGA Coutinho, and Rafael Mayo-García. Fostering collaboration in energy research and technological developments applying new exascale hpc techniques. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Proceedings*, pages 701–706. Institute of Electrical and Electronics Engineers (IEEE), 2016.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, July 2015.

[19] Nicholas T Karonis, Brian Toonen, and Ian Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[20] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[21] F. Reghenzani, G. Pozzi, G. Massari, S. Libutti, and W. Fornaciari. The mig framework: Enabling transparent process migration in Open MPI. *ACM International Conference Proceedings Series*, 2016.

[22] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[23] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[24] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[25] Criu - checkpoint/restore in userspace. `https://criu.org/`. Accessed: 2016-08-08.

[26] Wei-Jih Li and Jyh-Jong Tsay. Checkpointing message-passing interface (MPI) parallel programs. In *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, pages 147–152. IEEE, 1997.

[27] Joshua Hursey, Timothy I Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in Open MPI. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 49–58. ACM, 2009.

[28] 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.

[29] Joshua Hursey, Chris January, Mark O'Connor, Paul H Hargrove, David Lecomber, Jeffrey M Squyres, and Andrew Lumsdaine. Checkpoint/restart-enabled parallel debugging. In *European MPI Users' Group Meeting*, pages 219–228. Springer, 2010.

[30] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[31] Bryan Mills, Ryan E Grant, Kurt B Ferreira, and Rolf Riesen. Evaluating energy savings for checkpoint/restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 6. ACM, 2013.

[32] 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 *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[33] Wei Huang, Matthew J Koop, Qi Gao, and Dhabaleswar K Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 9. ACM, 2007.

[34] Simon Pickartz, Jens Breitbart, and Stefan Lankes. Impacts of virtualization on intra-host communication. 2016.

[35] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji, Nasir Ghani, et al. A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39(11):709–736, 2013.

[36] Georg Stellner. Cocheck: Checkpointing and process migration for MPI. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 526–531. IEEE, 1996.

[37] 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.

[38] Peter Smith and Norman C Hutchinson. Heterogeneous process migration: The tui system. *Software-Practice and Experience*, 28(6):611–640, 1998.

[39] Uriel Cabello, José Rodríguez, Amilcar Meneses, Sonia Mendoza, and Dominique Decouchant. Fault tolerance in heterogeneous multi-cluster systems through a task migration mechanism. In *Electrical Engineering, Computing Science and Automatic Control (CCE), 2014 11th International Conference on*, pages 1–7. IEEE, 2014.

[40] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146. IEEE, 1998.

[41] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[42] Ian Foster and Nicholas T Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society, 1998.

[43] Ke Wang, Xiaobing Zhou, Hao Chen, Michael Lang, and Ioan Raicu. Next generation job management systems for extreme-scale ensemble computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 111–114. ACM, 2014.

[44] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012.

[45] Michael Karo, Richard Lagerstrom, Marlys Kohnke, and Carl Albing. The application level placement scheduler. 2006.

[46] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[47] Songnian Zhou. Lsf: Load sharing in large heterogeneous distributed systems. In *I Workshop on Cluster Computing*, volume 136, 1992.

[48] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[49] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[50] Niels Provos and Nick Mathewson. libevent-an event notification library, 2003.

[51] Chris Rapier and Benjamin Bennett. High speed bulk data transfer using the ssh protocol. In *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*, page 11. ACM, 2008.

[52] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

[53] Namespaces - linux programmer's manual. `http://man7.org/linux/man-pages/man7/namespaces.7.html`. Accessed: 2016-08-22.

[54] Cgroups kernel documentation. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Accessed: 2016-09-12.

[55] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8. IEEE, 2012.

[56] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):39, 2015.

[57] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[58] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[59] Problem sizes and parameters in NAS parallel benchmarks. `http://www.nas.nasa.gov/publications/npb_problem_sizes.html`. Accessed: 2016-08-30.

[60] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[61] Andrew Makhorin. Glpk (gnu linear programming kit), 2008.

[62] John E Mitchell. Branch and cut. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[63] Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981.

[64] Jerome Vienne, Jitong Chen, Md Wasi-Ur-Rahman, Nusrat S Islam, Hari Subramoni, and Dhabaleswar K Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 48–55. IEEE, 2012.

[65] Robert Fourer, David M Gay, and Brian W Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.