High-throughput Computation through Efficient Resource Management

PhD Dissertation

Author: Sergio Iserte Agut

Advisors: Rafael Mayo Gual

Antonio José Peña Monferrer

Castelló de la Plana, Spain

November 2018

A Ana, per haver cregut en mi des del principi.
Gràcies, sense tu açò no haguera sigut possible.

Abstract

This proposal addresses, from two different approaches, the improvement of data centers productivity through an efficient resource management.

On the one hand, the combination of GPU remote virtualization technologies with workload managers in HPC clusters demonstrated an interesting increase in throughput, in terms of completed jobs per unit of time, during the research conducted in the predoctoral period. The dissertation begins with an extended study on its impact not only in productivity, but also in resource utilization and energy consumption. Hence, an efficient management of the access to these accelerators is crucial in order to obtain a higher number of completed jobs per unit of time rate. On the same basis, cloud computing environments (public or private) also deal with GPUs, since virtual machines can be equipped with these devices. As detailed in this document, the adoption of a GPU remote virtualization technology together with a resource manager introduces new working modes aimed to the global throughput improvement.

On the other hand, the second approach involves job reconfigurations in terms of varying its number of processes during the execution (commonly referred as MPI malleability) in order to increase the system throughput. Currently, MPI jobs suppose a high percentage of the total load in an HPC facility. In an effort to ease the adoption of malleability in scientific applications, this manuscript presents two solutions, from an OmpSs-like programming model approach and from a MPI-friendly syntax, which provide the necessary tools for *easily* converting an application into malleable. Performance evaluations reveal a non-negligible improvement not only in the throughput, but also in the job waiting time and in the energy consumption.

Resumen

Esta propuesta aborda, desde dos enfoques distintos, la mejora de la productividad de centros de procesamientos de datos mediante una gestión eficiente de los recursos.

Por un lado, la combinación de tecnologías de virtualización remotas de GPUs junto con gestores de cargas de trabajos en clústeres HPC, demostró en la investigación llevada a cabo durante el periodo predoctoral un interesante incremento de productividad, en terminos de trabajos completados por unidad de tiempo. La disertación comienza con un estudio extendido de su impacto no sólo en la productivad, sino también en utilización de recursos y consumo energético. Así pues, una gestión eficiente del acceso a estos aceleradores es crucial para obtener un mayor ratio de trabajos completados por unidad de tiempo. Del mismo modeo, entornos de *cloud computing* (públicos o privados) también gestionan GPUs, ya que las máquinas virtuales pueden ir equipadas con estos dispositivos. Tal y como se detalla en este documento, la adopción de una tecnología de virtualización de GPUs junto con un gestor de recursos, introduce nuevos modos de trabajo dirigidos al incremento de la productividad global.

Por el otro lado, el segundo enfoque involucra reconfiguración de trabajos en términos de modificar el número de procesos durante la ejecución (comúnmente referido como malleabilidad MPI) para incrementar la productividad del sistema. Actualmente, los trabajos MPI suponen un alto porcentaje del total de la carga en una instalación HPC. En el esfuerzo de facilitar la adopción de la maleabilidad en aplicaciones científicas, este manuscrito presenta dos soluciones, desde un enfoque del modelo de programación OmpSs y desde una sintaxis familiar a MPI, las cuales proveen de las herramientas necesarias para convertir *fácilmente* una aplicación en maleable. La evaluación de prestaciones revela un significativo incremento no sólo en la productividad, sino también en el tiempo de espera de los trabajos y del consumo energético.

Acknowledgements

This work would not have been possible without the constant support of my advisors Dr. Rafael Mayo and Dr. Antonio J. Peña. They not only have guided the dissertation, but also inspired me by sharing their experience and giving me enough freedom to develop my own ideas. I owe my deepest gratitude to Prof. Enrique S. Quintana-Ortí for his magnificent management of the HPC&A research group at Universitat Jaume I, where I learned invaluable things for my career and I developed my passion for research and HPC. Regarding the research group, I am deeply grateful to the current, or former, members who have contributed in the established pleasant working atmosphere, as well as, technical and administrative staff.

I would like to show my gratitude to Dr. Federico Silla and his group at the Universitat Politècnica de València for collaborating and allowing me to use their HPC infrastructure. I also want to thank Dr. Raúl Peña-Ortíz and his colleagues at the Universitat de València for their interest in extending our work with their ideas.

It is a pleasure to thank those who made easy to be away from home during my internships, people from the Computer Science Department at Barcelona Supercomputing Center and people in the High Performance and Distributed Computing at Queen's University of Belfast. In addition, I am also grateful with the Student Volunteer program at SC Conference Series and their organizers. After taking part in it for 4 times, I have been able to broaden my horizons, not only professionally but also personally, as never before I would have imagined.

Last by not least, special thanks to my family and my beloved, supportive, patient, inspiring..., everything, Ana. A vosaltes, Ana, al meu avi Joaquín, als meus pares Javier i Dina, i al meu germà Jorge, no puc deixar d'agrair-vos el vostre suport i haver estat ahí sempre que vos he necessitat.

Contents

List of Figures

List of Tables

Listings

Acronyms

ACK acknowledgement.

AMPI adaptive MPI.

API application programming interface.

AWS Amazon web service.

C/R checkpoint/restart.

CG conjugate gradient.

cGPU cloudified GPU.

CPU central processing unit.

DDR3 SDRAM double data rate type 3 synchronous dynamic random-access memory.

DMR dynamic management of resources.

DMR API dynamic management of resources application programming interface.

EasyGrid AMS EasyGrid application management system.

FDR fourteen data rate.

FS flexible sleep.

FT Fourier transform.

GB gigabyte.

Gb gigabit.

Gb/s gigabits per second.

GHz gigahertz.

GPGPU general-purpose computing on graphics processing units.

GPGPUMS general purpose GPU management system.

GPU graphic processing unit.

GSaaS GPU scheduling as a service.

HPC high-performance computing.

HTC high-throughput computing.

IaaS infrastructure as a service.

IB InfiniBand.

IS integer sort.

IT information technology.

LU lower-upper Gauss-Seidel solver.

MD molecular dynamics.

MIC many integrated core.

MPI message passing interface.

NPB NAS parallel benchmarks.

OS operating system.

PCI peripheral component interconnect.

PCM process checkpointing and migration.

PDU power distribution units.

PhD Philosophiae Doctor (doctor of philosophy).

rCUDA remote CUDA.

rGPU remote GPU.

RMS resource manager system.

RODVR resource-oriented distributed virtual routing.

SCR scalable checkpoint/restart.

SDK software development kit.

SPMD single program multiple data.

TCP/IP transmission control protocol/internet protocol.

ULFM user level failure migration.

vCPU virtual CPU.

VM virtual machine.

Part I

Background

CHAPTER 1

Introduction

This chapter starts with the motivation of this Philosophiae Doctor (doctor of philosophy) (PhD) dissertation. The objectives of this thesis are introduced next, followed by a statement of how the different contents are structured along this document.

1.1 Motivation

The unstoppable increase of computing cores in high-performance computing (HPC) facilities needs of specific programming paradigms that harness the underlying infrastructure. Nowadays, the vast majority of applications are ready to run on multiple computational units working together in the same problem in order to accelerate the computation. To leverage parallelism further, paradigms that benefit from concurrent computation on several nodes, such as the well-known message passing interface (MPI), are utilized. Besides, more recently, the use of graphic processing units (GPUs) as general-purpose accelerators has been extended thanks to their high number of computational units, which heavily exploit parallelism.

On the one hand, providing a cluster with GPUs presents several handicaps, such as its high acquisition cost, but also its high power consumption even in an idle state. GPU virtualization techniques allow sharing devices among applications, provided that sufficient enough memory is available in the GPU, especially if GPU-enabled applications may use a remote GPU from another node. This means that when having a non-accelerated application allocating all the cores of a node, the GPU in that node can be still leveraged. All these insights have a remarkable positive impact in the GPU utilization, what is immediately translated into a higher throughput and a lower energy consumption. In order to leverage GPU virtualization in a cluster, rCUDA [58][59] has been proven to be a very reliable tool for HPC. Nevertheless, the current resource manager systems (RMSs) are not ready for coping with virtual GPUs.

On the other hand, MPI applications usually follow the single program multiple data (SPMD) programming model, where all the processes execute the same code, but aiming to different data. This homogeneous workflow has permitted that ideas such as "malleability" have been adopted in this field. MPI malleability can change on-the-fly the number of processes of a given job allowing it to continue its execution with the new process layout. This reconfiguration also assumes data-redistribution among processes, what involves a higher effort from users willing to leverage

malleability. For this reason, resource management tasks have become crucial when refereeing to resource utilization and global throughput. In order to be aware of the cluster status, Slurm, one of the most popular RMS in HPC, is able to handle detailed information of nodes and jobs making easier to implement reconfiguration policies for malleability. However, we still need a communication layer between both entities, the runtime system and the RMS. After their cooperation, a fixed workload that cannot be fitted to the system requirements turns into an adaptive workload ready for being reconfigured for the good of the productivity.

1.2 Objectives

As detailed in the motivation, *our aim is to design deploy and analyze a set of tools to increase the throughput of a production HPC cluster*. We are focussed on optimizing the resource utilization in the HPC facility from two different perspectives: GPU, used as accelerators and processors in the compute nodes.

Remote GPU management

First, we perform an exhaustive evaluation of the GPU virtualization adoption in a cluster processing realistic workloads of scientific applications. This analysis arises from the pre-doctoral development of an Slurm version with support for remote GPUs enabled by rCUDA.

GPU applications do not usually make a continuos stressing use of the devices, which present a low utilization rate. Apart from that, in a production cluster jobs are assigned with the requested resources. However, when in need for GPUs, jobs are restricted to be only assigned to nodes hosting GPUs. GPU virtualization technologies may be leveraged to mitigate those side effects, by sharing and offshoring the GPUs.

Dynamic Management of Resources

The second approach for high throughput computing in our research deals with MPI applications and involves the development of an application programming interface (API) to help users turning their applications into malleables.

For this purpose, we have implemented the internal communication between the RMS and the runtime to perform support and easy reconfiguration actions targeting global throughput. The RMS evaluates the reconfiguration requests triggered by the jobs, and the runtime handles processes and data redistribution.

The reconfiguration request includes a series of user-defined parameters (minimum, maximum and preferred) that tune the scheduling. With those parameters and the information of the system, the reconfiguration policy decides if expand or shrink a job.

When an action is taken the RMS conveys the guidelines of the reconfiguration to the runtime, which must perform the following operations: i) to allocate a new set of resources (only when expanding); ii) to spawn the new processes in a new MPI communicator; iii) to perform the data redistribution among processes in both communicators; iv) to define the exact point of the execution where the new processes are going to continue; v) to terminate the processes of the initial communicator; and vi) to deallocate the resources (only when shrinking).

We foster the development of malleable applications through two points of view:

- An API integrated in the Nanos++ runtime which presents an OmpSs-like syntax with automatic data redistribution.

- An external library linked to Nanos++ and Slurm which offers an MPI-like syntax allowing user-driven data redistribution.

1.3 Structure of the Document

This manuscript is divided into four parts. Part I introduces background information helpful to understand the rest of the document. This part is composed of three chapters. Chapter 1 is the current chapter and includes the motivation, the objectives pursued and the structure overview for this PhD dissertation. Chapter 2 describes the technology related with this thesis. Chapter 3 presents the state of the art of GPU virtualization and MPI malleability.

The two following parts present the conducted research in efficient resource management for the two different approaches included in this dissertation:

- Part II describes the effect of using remote GPU (rGPU) management in HPC clusters and cloud infrastructures. Chapter 4 studies the benefits of adopting a workload manager remote GPU (rGPU)-enabled in a supercomputer. Chapter 5 explores the impact of rGPU management in public and private clouds, where we present GPU scheduling as a service (GSaaS).

- Part III includes the design and implementation details as well as the correspondent performance evaluation of dynamic management of resources (DMR) malleability solutions. Specifically, Chapter 6 describes the initial study of a malleability API with an OmpSs-like syntax. While Chapter 7, redefines the API in order to offer a more usable malleability solution in the shape of a library based on an MPI-like syntax.

Part IV concludes this work with a summary, future work and the conclusions of this PhD dissertation. Appendix A displays the list of publications related to this thesis. Appendix B contains a reproducibility artifact for DMRlib experiments.

CHAPTER 2

Related Technology

This chapter presents the most relevant software components that this PhD dissertation builds upon. Apart from describing each technology, we remark the extensions or plugins leveraged in order to provide the necessary background for subsequent chapters. The Slurm workload manager is introduced first, since it is leveraged during all the research performed in this thesis. Next, OpenStack and rCUDA, employed in Part II, are presented. Finally, we describe the OmpSs programming model, in which PartIII is based on.

2.1 Slurm

Slurm is an open-source, fault-tolerant, and highly scalable cluster manager and job scheduling system for Linux clusters.

Figure 2.1 depicts a basic deployment of Slurm that consists of a daemon that runs on each computing node (*slurmd*), a central daemon that runs on the management node (*slurmctld*), and several command line utilities (*srun, scancel, sinfo, squeue, and scontrol*). The daemons manage nodes, the basic compute resource in Slurm; partitions, which group nodes into logical disjoint sets; jobs or allocations of resources assigned to a user for a specified amount of time; and job steps, which are sets of (possibly parallel) tasks within a job. Available nodes within a partition are assigned to jobs in the priority queue.

Slurm can be extended with plugins for:

- Accounting Storage: Primarily Used to store historical data about jobs. When used with SlurmDBD (Slurm Database Daemon), it can also supply a limits based system along with historical system status.
- Account Gather Energy: Gather energy consumption data per job or nodes in the system. This plugin is integrated with the Accounting Storage and Job Account Gather plugins.
- Authentication of communications: Provides authentication mechanism between various components of Slurm.
- Checkpoint: Interface to various checkpoint mechanisms.

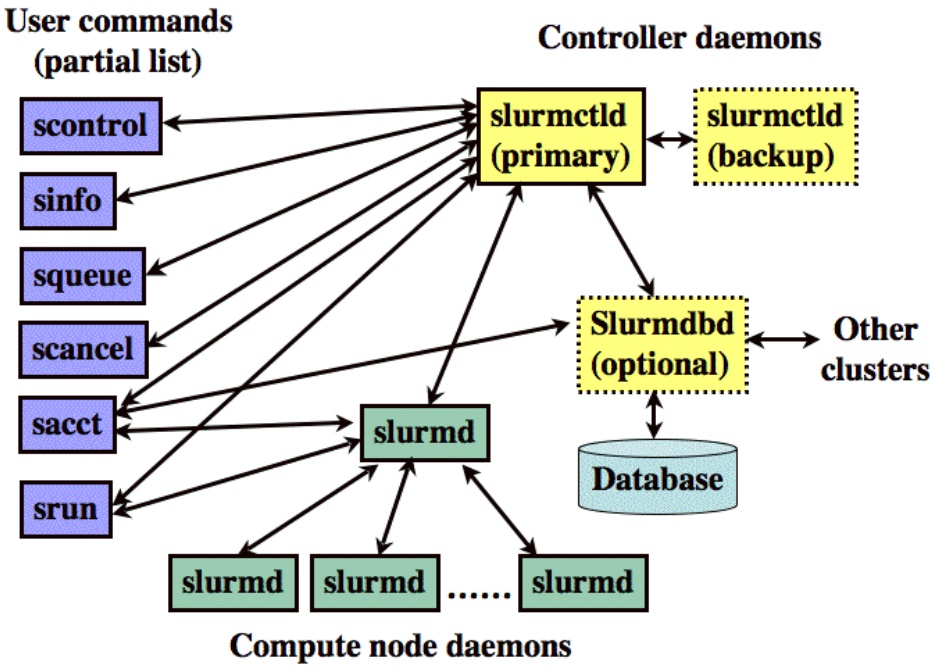


Figure 2.1: Slurm basic components interactions diagram.

- Cryptography (Digital Signature Generation): Mechanism used to generate a digital signature, which is used to validate that job step is authorized to execute on specific nodes. This is distinct from the plugin used for Authentication since the job step request is sent from the user's srun command rather than directly from the slurmctld daemon, which generates the job step credential and its digital signature.
- Generic Resources: Provide interface to control generic resources like GPUs and Intel many integrated core (MIC) processors.
- Job Submit: Custom plugin to allow site specific control over job requirements at submission and update.
- Job Accounting Gather: Gather job step resource utilization data.
- Job Completion Logging: Log a job's termination data. This is typically a subset of data stored by an Accounting Storage Plugin.
- Launchers: Controls the mechanism used by the *srun* command to launch the tasks.
- MPI: Provides different hooks for the various MPI implementations. For example, this can set MPI specific environment variables.
- Preempt: Determines which jobs can preempt other jobs and the preemption mechanism to be used.
- Priority: Assigns priorities to jobs upon submission and on an ongoing basis (e.g. as they age).

- Process tracking (for signaling): Provides a mechanism for identifying the processes associated with each job. Used for job accounting and signaling.
- Scheduler: Plugin determines how and when Slurm schedules jobs.
- Node selection: Plugin used to determine the resources used for a job allocation.
- Switch or interconnect: Plugin to interface with a switch or interconnect. For most systems (Ethernet or InfiniBand) this is not needed.
- Task Affinity: Provides mechanism to bind a job and it's individual tasks to specific processors.
- Network Topology: Optimizes resource selection based upon the network topology. Used for both job allocations and advanced reservation.

2.1.1 Consumable Generic Resources

The Slurm extension for using rGPUs in an HPC cluster was used in the study performed out in Chapter 4. This extension provides two new plugins:

- `gres/rgpu`: a *generic resource* plugin which introduces a new type of resource in the cluster, the rGPU.
- `select/cons_rgpu`: a *node selection* plugin that detaches the rGPUs from the node configuration of Slurm and make the rGPUs available from any node and shareable among jobs.

Without these plugins, consumable resources as GPUs were not assignable to jobs allocated in nodes different from the GPU host. Besides, if two jobs were allocated in the same host, only one could have the GPU assigned.

2.1.2 Resource Reallocation

Part III at this dissertation discusses how jobs are resized with the collaboration of Nanos++ (BSC's OmpSs runtime system) and Slurm. Slurm is in charge of managing the resource allocation of the jobs when a resize is performed, supporting changes in the job allocation of nodes. Natively, Slurm offers a job resize mechanism described below:

- Job A has to be expanded
  1. Submit a new job B with a dependency on the initial job A. Job B requests the number of nodes (NB) to be added to job A.
  2. Update job B setting its number of nodes to 0. This produces a set of NB allocated nodes which are not attached to any job.
  3. Cancel job B.
  4. Update job A and set its number of nodes to NA+NB.
- Job A has to be shrunk
  1. Update job A setting its number of nodes to the desired size (NA is updated).

After these steps, Slurm's environment variables for job A are updated. These commands have no effect on the status of the running job, and the user remains responsible for any malleability process and data redistribution.

2.2 rCUDA

rCUDA is a framework that provides transparent access to any GPU installed in a cluster, independently of the location of the application requesting GPGPU services. Thus, rCUDA is useful in a number of scenarios: i) in a cluster equipped with rCUDA, the designers can reduce the total number of GPUs in the system, improving the utilization rate of the power-hungry hardware accelerators; ii) rCUDA can also be leveraged to significantly accelerate the data-parallel computations of a conventional cluster, by adding only a reduced pool of accelerators to the system, much smaller than the total number of nodes; and iii) rCUDA increases the number of GPUs that can be accessed by an application, from only the local accelerators to all GPUs available in the cluster. In summary, in many practical cases, in exchange for a slight increase of the execution time of GPU-enabled applications, considerable savings can be achieved in energy consumption, maintenance, space, and cooling with rCUDA.

rCUDA is organized as a client-server distributed architecture (Figure 2.2). The client middleware runs in the same cluster node as the application demanding GPU acceleration services, while the server middleware runs in the cluster node where the physical GPU resides.

- The client middleware consists of a collection of wrappers that replace the NVIDIA CUDA Runtime (provided by NVIDIA as a shared library) in the client (GPU-less) node, and some accelerated libraries such as cuBLAS, cuFFT and cuSPARSE. These wrappers are in charge of forwarding the API calls from the applications requesting acceleration services to the server middleware, and retrieving the results, providing applications with the illusion of a direct access to a local GPU.

- The server middleware runs as a service on one or more cluster nodes equipped with one or more GPUs each. This middleware receives, interprets, and executes the API calls from the clients on a real GPU, employing a different process to serve each remote execution over an independent GPU context, thus enabling GPU multiplexing.

rCUDA accommodates several underlying client-server communication technologies, thanks to its modular, layered architecture, which supports runtime-loadable network-specific communication libraries. This software currently provides communication modules for Ethernet and InfiniBand based networks. Furthermore, regardless of the specific communication technology, data transfers between rCUDA clients and servers are pipelined for performance, using pre-allocated buffers of pinned memory.

2.3 OpenStack

OpenStack is a cloud operating system (OS) that provides infrastructure as a service (IaaS). OpenStack controls large pools of compute, storage, and networking resources throughout a data center. All these resources are managed through a dashboard or an API that provides administrators control while empowering their users to provision resources through a web interface or a command-line interface. OpenStack supports most recent hypervisors and handles provisioning and life-cycle management of VMs. The OpenStack architecture offers flexibility to create a custom cloud, with no proprietary hardware or software requirements, and the ability to integrate with legacy systems and third-party technologies. From the HPC perspective, OpenStack offers high performance virtual machine configurations with different hardware architectures. OpenStack is composed of several projects, each one responsible for a service of the OS, as it is shown in Figure 2.3.

Among the 46 available projects, the most popular are:

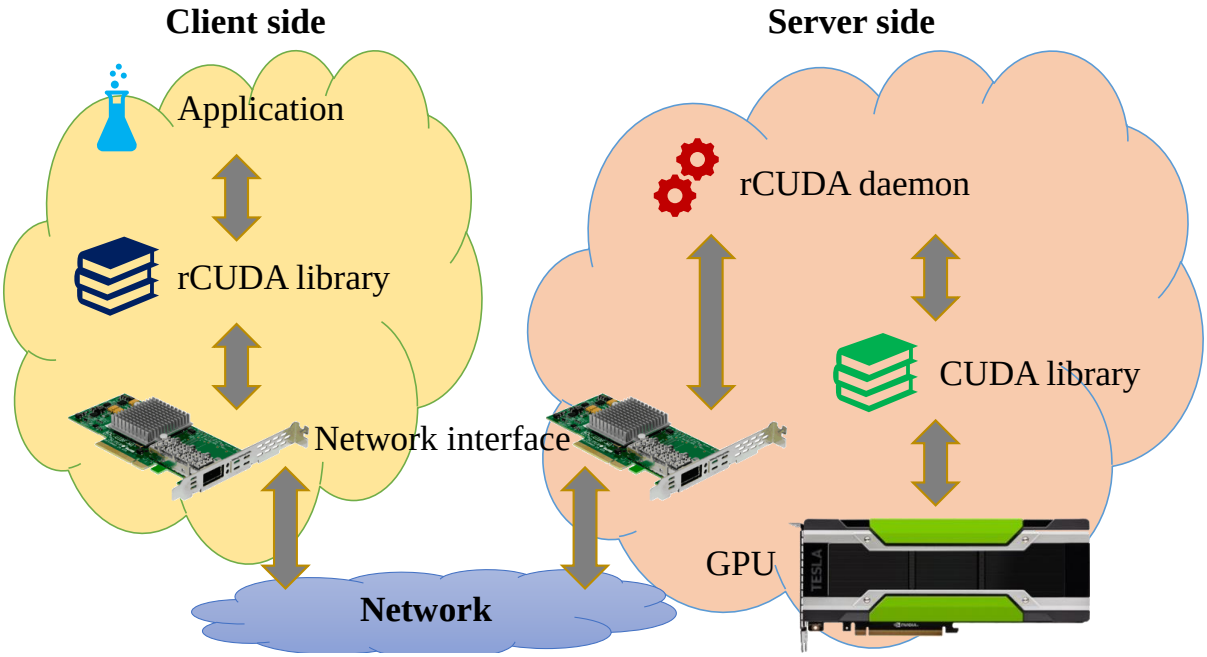


Figure 2.2: rCUDA scheme for the client and server components.

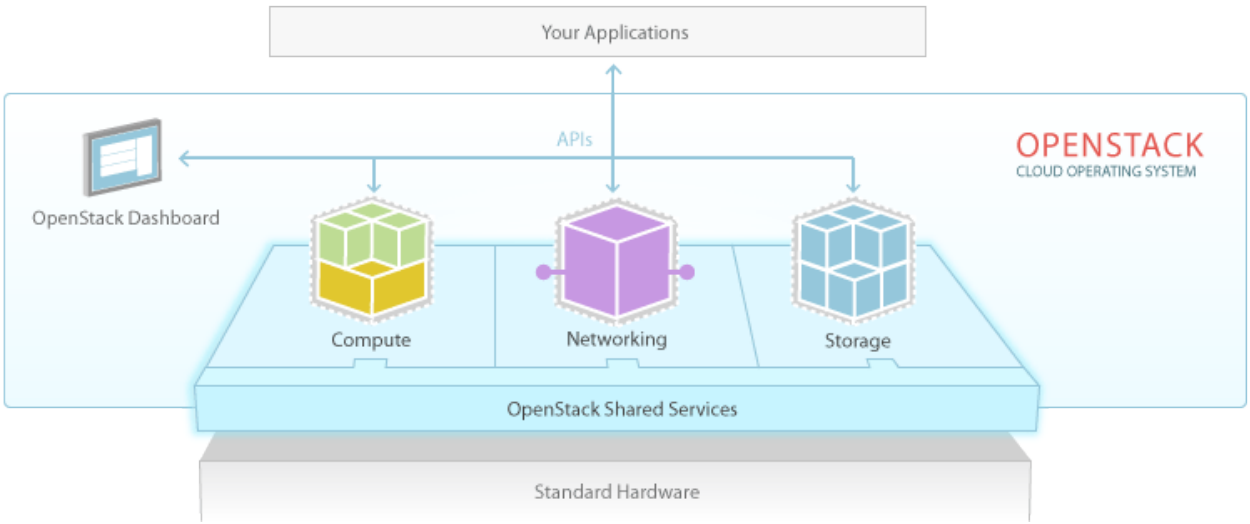


Figure 2.3: OpenStack shared services integration scheme.

- NOVA: compute service.
- NEUTRON: networking.
- SWIFT: object storage
- GLANCE: image service.
- KEYSTONE: identity service.
- CINDER: block storage.

2.4 OmpSs

The OmpSs programming model extends OpenMP with new directives to support asynchronous parallelism and heterogeneity. OmpSs is a task-based model that can define tasks from functions and structured blocks. By setting data dependencies among these tasks, OmpSs is capable of implementing the asynchronous parallelism. OmpSs is implemented by BSC's Mercurium compiler and Nanos++ runtime system.

Mercurium is a source-to-source compiler aimed at fast prototyping. The compiler refactors the original code replacing the synchronous calls annotated by the user, with asynchronous calls to the Nanos++ external API.

The Nanos++ runtime provides services to support task parallelism using synchronization based on data dependencies. Its main purpose is to be used in research of parallel programming environments. Thanks to its modularity, it can be extended with plugins for:

- Task scheduling policy.
- Thread barrier.
- Device support.
- Instrumentation formats.
- Dependencies approach.
- Throttling policies.

Part III is based on the Nanos++ *device support for cluster* plugin, which enables working with distributed-memory systems. This *support for clusters* allows the user to integrate the MPI paradigm into their OmpSs code. Furthermore, this plugin implements an offloading feature that performs dynamic offloads of tasks among MPI processes [67]. Leveraging this support, a task can be migrated from one node to another. The process operates as follows:

- The job is launched and the initial MPI processes are created.
- When the task offloading is triggered, a new set of MPI processes is created in a new communicator, using `MPI_COMM_SPAWN_MULTIPLE`.
- The initial processes pack and send the arguments and the data to the target processes in the new communicator. Notice that the packet also contains the user code that has to be executed by the new processes.

- The new processes receive all the data and execute the expected code.
- Finally, the processes send a message back to signal the end of the offloaded tasks and terminate.

As we describe in Chapter 6, OmpSs functionality has been extended to leverage job malleability.

CHAPTER 3

State of the Art

This chapter presents the state of art relevant to this dissertation, reviewing other research and/or engineering efforts and stating the difference with our contributions. The chapter is divided in two sections. The first section describes the current solutions for GPU virtualization management in HPC. The second section undertakes a review of the current state of malleability and the different available solutions to deal with it.

3.1 Remote GPUs Management

One of the issues for high-throughput computing (HTC) addressed in this dissertation involves GPGPU virtualization, specially when it is aimed to increase a cluster productivity. For this reason, in this section we describe the most remarkable technologies and projects related to this topic.

3.1.1 GPGPU Virtualization Technologies

GPUs have remarkably evolved during the last few years, from being just graphics coprocessors to become powerful general-purpose accelerators, profusely adopted in HPC systems. In addition to the favorable performance/cost ratio of GPUs, this evolution has been further stimulated by considerable advances in GPU programmability. On the other hand, the deployment of GPUs is hampered by their high acquisition and maintenance (including energy) costs, as well as the limited amount of (GPU-appealing) data-parallelism for many applications. In this sense, GPGPU virtualization offers an alluring means to increase utilization of the GPUs in an HPC facility, which can potentially yield a faster amortization of the total costs of ownership (TCO) for this type of equipment. Concretely, GPU virtualization logically decouples the GPUs in the cluster from the nodes they are located in, thus opening a path to share the accelerators among all the applications that request GPGPU services, independently of whether the node(s) these applications are mapped to are equipped with a GPU. In consequence, the GPUs can be accessed from any application running in the cluster, the amount of these accelerators can be reduced, and their utilization rate can be significantly improved.

Frameworks such as CUDA [55] assist programmers in using GPUs for general-purpose computing. Several remote GPU virtualization solutions exist for this API, such as GridCuda [43],

DS-CUDA [56], gVirtuS [17], vCUDA [69], GViM [20], Shadowfax [53], gCloud [8], Vgris [63] and rCUDA [58][59]. These middleware systems make a GPU accessible from remote nodes, that is, from nodes where the physical device is not hosted. In this way, these tools provide applications with virtual instances of the real device, which can therefore be concurrently shared. In general, CUDA-based virtualization solutions aim to offer the same API as the NVIDIA CUDA Runtime API does, enabling seamless remote access.

CUDA-based GPU virtualization solutions may be classified into 2 types:

- those devised as general-purpose virtualization solutions, to be used in native domains, and
- those intended to be used in the context of cloud computing.

3.1.1.1 Remote GPUs management in clusters

Among the virtualization solutions that provide general purpose GPU virtualization, one can find rCUDA, GridCuda and DS-CUDA. rCUDA, described in 2.2, features CUDA 7.5 and provides specific communication support for transmission control protocol/internet protocol (TCP/IP) compatible networks [10, 64] as well as for InfiniBand (IB) fabrics [58, 59, 9]. GridCuda [42] also offers access to rGPUs in a cluster but supports the old CUDA version 2.3. Regarding DS-CUDA, it integrates a more recent version of CUDA (v4.1) and includes specific communication support for IB by making use of the IB Verbs API. However, DS-CUDA presents several strong limitations, such as not allowing data transfers with pinned memory. In the those frameworks, applications invoking CUDA kernels are not aware that their requests are intercepted by the corresponding GPU virtualization middleware and redirected to a real GPU, which is generally located in a remote node of the cluster.

Although remote GPU virtualization has demonstrated very low overhead with respect to a configuration with a local GPU, due to its novelty, this technology was not supported by job schedulers that are commonly encountered in production clusters (e.g., SLURM[1], PBSPro[2], MOAB[3], TORQUE[4], LSF[5], OAR[6], MAUI[7], LoadLever[8], Condor[9], and Oracle Grid Engine[10]). In particular, a common job scheduler in production only dealt with real GPUs so that, when a job requested a number of nodes equipped with one (or more) GPU(s), the scheduler tried to map that job to nodes that actually owned the requested number of GPUs, thus impairing the benefits of GPU virtualization.

In the master's thesis in [22] we presented an extension to Slurm that support remote GPU virtualization. With this extension Slurm becomes aware of the fact that the assignment would no longer be constrained by the GPU kernels having to be executed in the same node where the invoking application is mapped to. The goal was thus to create a GPU virtualization-aware job scheduler which in turn allowed applications to leverage all the cluster GPUs, independently of their location. However, the master's thesis lacks of a thorough analysis of the Slurm's extension in day-to-day HPC scenarios.

[1] https://slurm.schedmd.com
[2] https://pbsworks.com
[3] http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/
[4] http://www.adaptivecomputing.com/products/open-source/torque/
[5] https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_welcome.html
[6] https://oar.imag.fr
[7] http://www.adaptivecomputing.com/products/open-source/maui/
[8] https://www-03.ibm.com/systems/power/software/loadleveler/
[9] https://research.cs.wisc.edu/htcondor/
[10] http://www.oracle.com/technetwork/oem/grid-engine-166852.html

3.1.1.2 Remote GPUs management in clouds

General-purpose computing on graphics processing units (GPGPU) in the cloud has been a topic massively addressed in the last years. From the beginning, peripheral component interconnect (PCI) passthrough [78] is the most common solution for providing GPU-enabled virtual machines (VMs), where a VM is configured with exclusive access to the PCI port of the accelerator. Despite the rigidness of this solution, it is used by cloud providers like Amazon web service (AWS)[11], which is currently one of the flagships in IaaS. AWS offers GPU-capable VMs with support to CUDA and OpenCL through Amazon Elastic Compute Cloud EC2[12]. Authors in [21] discuss about how heterogeneous computing with GPUs can benefit the Cloud and give their perspective on the need for a paradigm shift.

More flexible approaches have been proposed to provide GPU access to non-GPU-enabled clients, contributing to the current state-of-art. In a desktop virtualization setting we find the NVIDIA GRID GPU[13] and the Intel KVMGT[14] technology which implement complete GPU virtualization. However, our focus is on data centers.

Several solutions have been developed to be specifically used within VMs, such as, for example, vCUDA, GViM, gVirtuS, Shadowfax, gCloud and Vgris. The vCUDA technology, intended for Xen[15] VMs, only supports an old CUDA version (v3.2) and implements an unspecified subset of the CUDA Runtime API. GViM, targeting Xen environments, is based on the obsolete CUDA version 1.1 and, in principle, does not implement the entire CUDA Runtime API. gVirtuS is based on an old CUDA version (being more than 6 years without any update)[16] and implements only a small portion of its API. Despite being designed for VMs, it also provides TCP/IP communications for remote GPU virtualization, thus allowing applications in a non-virtualized environment to access GPUs located in other nodes. Shadowfax allows Xen VMs to access any GPUs in the cluster; however it only supports the obsolete CUDA version . gCloud is a similar solution, but it is not yet integrated in a cloud computing manager, and the application source code has to be adapted to run in the virtual environment. Finally, Vgris [63] has not been tested on cloud infrastructures and only allows local access to GPUs.

Furthermore, we can also find in the literature efforts to integrate GPGPU virtualization technologies in cloud environments. For instance, authors in [37] combined OpenStack, KVM[17], and rCUDA to enable scalable use of GPUs among virtual machines.

Although some of these projects are showing a high rate of matureness, they have neglected their integration in real cloud platforms, and their management of the GPU resources.

3.2 Malleability

This section addresses the other issue for HTC included in this manuscript: job malleability.

Applications are submitted in the shape of jobs to the workload manager. Jobs can be classified in four different types depending on who and when their number of processes (job size) have been defined [46]. Hence, the classification (see Table 3.1) takes into account if the size of a job has been determined by the user or by the system and if it has been decided at submittal or during the

[11] Amazon web services: `http://aws.amazon.com`.
[12] Amazon EC2: `http://aws.amazon.com/ec2`.
[13] `www.nvidia.com/object/grid-technology.html`
[14] `https://01.org/igvt-g/blogs/wangbo85/2017/intel-gvt-g-kvmgt-public-release-q22017`
[15] `https://www.xenproject.org`
[16] `https://www.openhub.net/p/gvirtus`
[17] `https://www.linux-kvm.org/page/Main_Page`

Table 3.1: D. Feitelson and L. Rudolph job classification depending on how their size is determined.

Who	**When**	
	Submission	Execution
User	Rigid	Evolving
System	Moldable	Malleable

execution of the job.

The most common type of jobs in current HPC facilities are the rigid jobs; however, many RMSs provide commands that allow moldable jobs. On the contrary, jobs that change their size during the execution time are not so usual, since they not only require a resizing framework, but also the effort from the user who is the responsible for implementing the reconfiguration of the application.

3.2.1 Job Reconfiguration

In 1996, D. Feitelson and L.Rudolph established a classification for jobs [15] considering who and when their size is determined (see Table 3.1).

From that classification, many studies in job reconfiguration arose. For instance, the first steps toward malleability in shared-memory systems, which exploited the flexibility of applications. In [57] authors present a series of preemptive policies that interrupt active jobs in order to redistribute processors among the pending jobs.

More interesting is the adoption of malleability in distributed-memory systems. Here, depending on how the application data is redistributed during a job reconfiguration, we distinguish two groups: data-on-disk and data-in-memory. Furthermore, some malleability solutions have been designed to collaborate with an RMS. Thus, the RMS, aware of the system status, decides *when* and *where* the reconfiguration has to occur, while the runtime, with this information, performs the necessary operations for the reconfiguration of the job.

3.2.1.1 On-disk Reconfiguration

On-disk reconfiguration is based on the principle of saving the state of a job in a non-volatile memory device, in order to load it when required. checkpoint/restart (C/R) is the most popular example of that type of mechanism. C/R saves the state of an application at a given point of its execution and reloads it at a future time. Traditionally, it is used for preventing data loss in the exceptional case of a system fault. However, C/R has also been utilized in job malleability with the methodology of halting the execution in order to resume it with a different number of processes (see Figure 3.1).

Authors in [11] present an extension of the process checkpointing and migration (PCM) MPI library [13] in order to automate the processes reconfiguration task. Their work explores how malleability can be used in C/R applications [12]. The checkpoint–and–reconfigure mechanism is leveraged to restart applications with a different number of processes from data stored in checkpoint files.

Charm++ [38] is a parallel programming system that virtualizes the processors and bases its paradigm in migratable objects called *chares*. Their implicit synchronization mechanisms allow jobs to be reconfigured [1]. The reconfiguration process is based on the native Charm++ C/R feature, so that, after saving the state of the application, the *chares* are redistributed among the new processes to resume the execution. Leveraging Charm++ we find Adaptive MPI (AMPI) [19], which is an

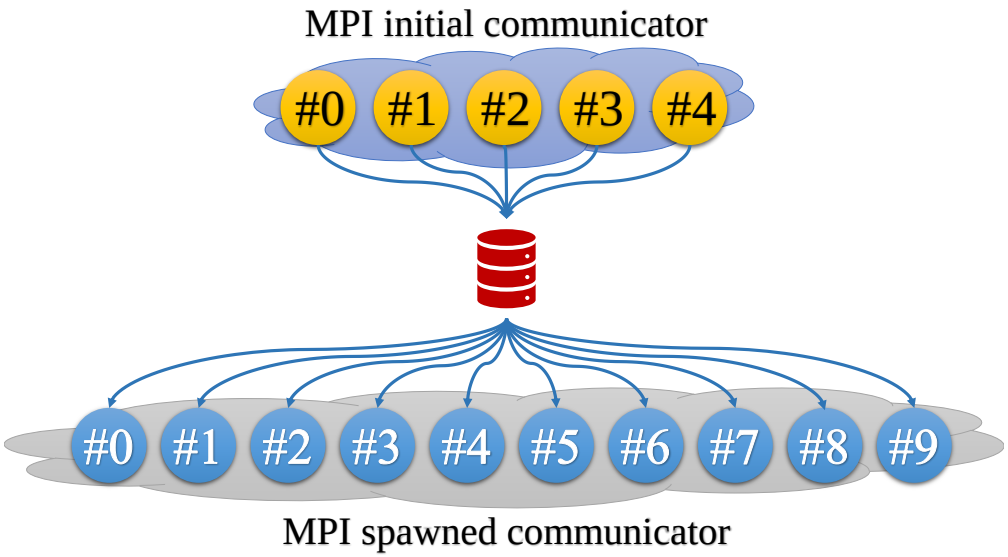


Figure 3.1: Scheme of on-disk reconfiguration.

implementation of MPI on top of Charm++'s adaptive runtime system.

Authors in [41] extent the scalable checkpoint/restart (SCR) MPI library [54], which only fetches the checkpoint file matching the MPI rank which saved the state. The extension enables job reconfiguration by allowing each rank to request any information on demand. However, the user is expected to use SCR API functions to orchestrate malleability.

3.2.1.2 In-memory Reconfiguration

Traditional on-disk C/R solutions show a low performance because of the costly disk access when writing and reading. Although there are in-memory C/R solutions [79], they are not yet standardized in production environments. Dynamic data redistribution mechanisms distribute the data, point-to-point or collectively, among processes without accessing the disk (see Figure 3.2). The data is always stored in the volatile memory of the node, what accelerates its manipulation.

The EasyGrid application management system (EasyGrid AMS) library [65] is aimed at adjusting automatically the scale of a running application. For this reason, the library provides a new set of functions enables developers to: determine reconfiguration points; calculate the new grade of parallelism, depending on the data gathered during the execution, and redistribute the data; and trigger reconfigurations.

Authors in [48] present an extension of MPI (named Flex-MPI) which integrates three new features: monitoring, load-balancing and data redistribution. Their performance-aware approach enforces to follow the next steps for reconfiguration:

- Get information about processes and environment.
- Register the data structures managed by the runtime.
- Enable the application performance monitoring engine.
- Reconfigure the job, if needed.

In [41], the MPI user level failure migration (ULFM) library is leveraged for malleability. Although ULFM is not in the MPI standard implementations, authors in this work combine the fault

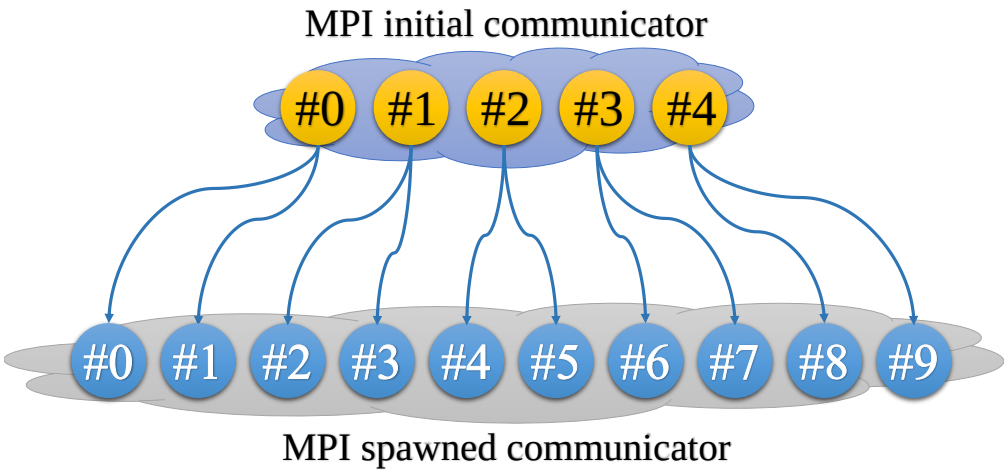


Figure 3.2: Scheme of in-memory reconfiguration.

tolerance mechanism in ULFM `MPI_Comm_shrink` with the MPI-2 standard routine `MPI_Comm_spawn` to support dynamic reconfiguration. When the application is requested to expand, the user is responsible for creating the new processes with `MPI_Comm_spawn` and redistribute the data among them. On the contrary, if it the application is expected to shrink, the user is in charge of redistributing the data, killing the required MPI processes, and using `MPI_Comm_shrink` to get the correct MPI communicator.

3.2.1.3 System-aware Reconfiguration

So far, we have reviewed a set of libraries and runtimes capable of reconfiguring applications using different approaches. However, some authors have implemented a simple ad-hoc scheduler, or a simulator, in order to evaluate their solutions. Production environments rely on complex RMSs which offer a wide range of working options. These RMSs manage the underlying resources in the cluster, as well as, the queue of jobs. That is why, these are the perfect candidates to orchestrate the reconfigurations of a set of malleable jobs; in other words, an adaptive workload. An adaptive workload does not need all its jobs to be malleable; however, the higher malleability rate, the higher adaptivity to the system status. We next describe, some of the most relevant efforts in adaptive workloads management.

ReSHAPE [75] is a coupled solution for adaptive workloads that includes from the reconfiguration libraries to the scheduler, through the runtime. This strong integration forces ReSHAPE users to specifically develop applications for this exclusive system.

The Power Aware Resource Manager (PARM) [68] uses over-provisioning, power capping, and job malleability to maximize job throughput under a strict power budget in over-provisioned facilities. Regarding the malleability, it relies on the Charm++ runtime support, which dynamically redistributes compute objects to processors.

Authors in [61] combined adaptive MPI (AMPI) with the workload manager Torque/Maui. Apart from extending the RMS to deal with malleable jobs, they provided a communication layer between the Charm++ runtime and the Torque/Maui scheduler.

Elastic MPI [7] is presented as an infrastructure and API extensions for malleable execution of MPI applications based on Slurm an the MPI implementation MPICH[18]. In this work, Slurm and

[18] `http://www.mpich.org`

MPICH are extended with new functionalities in order to deal with job reconfiguration. The close integration of both systems allows Slurm to get the control of creating and removing MPI processes while handling the resource allocations. MPICH has been provided with a set of new functions than substitute and complement the current standard implementation. With these functions, the application is initially defined as malleable and, periodically, the processes of the application check if Slurm initiated a reconfiguration.

3.2.2 Malleable Applications

Some of the previous malleability solutions have been used to develop malleable versions of full applications. In this section we review some of the most relevant efforts in turning applications into malleable. Although we can also find synthetic applications emulating behaviors of production software, they are out of the scope of our analysis.

In [73] the authors use ReSHAPE to bring malleability to their jobs, and they base their evaluation on NAS parallel benchmarks (NPB)[19]: integer sort (IS), conjugate gradient (CG), Fourier transform (FT) and lower-upper Gauss-Seidel solver (LU). While the last one is defined as a pseudo-application, the rest are kernels. In later studies [74] the authors move to synthetic workloads.

We can also find the CG method[20] working on top of other resize tools such as Flex-MPI [48]. Besides, the authors of Flex-MPI also target Jacobi[21] and Epigraph[22], with the latter being a more complex HPC application. Again, in subsequent studies [47], the authors move to synthetic workloads.

A higher level of complexity is found in [61], with the malleable version of LeanMD[23]. This is a molecular dynamics (MD) mini-application which implements a simplified version of the force calculations of NAMD[24]. The rest of the programs used by these authors synthetically emulate other applications.

In [76] authors developed a malleable version of LAMMPS[25]. One of the most complex HPC application developed so far in the scope of malleability. This is a classical molecular dynamics code, implemented using its C/R capabilities, with ReSHAPE driving the reconfigurations.

So far, the presented applications follow a regular pattern, where all the processes are expected to execute the same operations over different data and to constantly swap their data. Authors in [7] implement an application with data independence and a *master-worker* scheme, using their MPICH extension for malleability. These features are inherently suitable for dynamic reconfiguration of processes, because *workers* do their job independently and can leave and join the working group at any time.

3.2.3 Usability Study

This section presents examples of how each malleability solution addresses malleability programmatically. The study is limited to the implementation in: PCM API, AMPI, Flex-MPI and the Elastic MPI, since the projects EasyGrid AMS, SCR extension, ULFM and ReSHAPE do not provide any example on how to implement malleability.

[19] `http://www.nas.nasa.gov/Software/NPB`
[20] `https://en.wikipedia.org/wiki/Conjugate_gradient_method`
[21] `https://en.wikipedia.org/wiki/Jacobi_method`
[22] `http://epigraph.mpi-inf.mpg.de/WebGRAPH`
[23] `http://charm.cs.illinois.edu/research/leanmd`
[24] `http://www.ks.uiuc.edu/Research/namd`
[25] `http://lammps.sandia.gov`

```
1  void main(int argc, char **argv) {
2    MPI_Init(&argc, &argv);
3    MPI_Comm_get_parent(&parentComm);
4    if (parentComm == MPI_COMM_NULL) {
5      step = 0;
6      /* Initialization */
7    } else {
8      MPI_Recv(&dataSize, myRank, parentComm);
9      MPI_Recv(data, myRank, parentComm);
10     MPI_Recv(&step, myRank, parentComm);
11   }
12   compute(data, dataSize, step);
13 }
14 void compute(double *data, int dataSize, int step) {
15   for (t = step; t < TIMESTEPS; t++) {
16     nodeList = get_new_nodelist_somehow();
17     if (nodelist != NULL) {
18       MPI_Comm_spawn(myapp.bin, nodeList, &newComm);
19       MPI_Send(dataSize, myRank, newComm);
20       MPI_Send(data, myRank, newComm);
21       MPI_Send(t, myRank, newComm);
22       exit(0);
23     }
24     /* Computation */
25   }
26 }
```

Listing 3.1: Pseudo-code of job reconfiguration using bare MPI.

Malleability traditionally targets at iterative applications, in which their main loop represents an ideal synchronization point for redistributing data among processes. In order to implement malleability using the different solutions, we have designed an iterative application which performs calculations over a single data array called *data* whose size is determined in *dataSize*. Using this dummy application, we will perform a migration (it is the most simple case of reconfiguration because the number of processes does not change) using MPI pseudo-code and the tools provided by each malleability solution. Of course, a reconfiguration that expands or shrinks a job is more complex programmatically, but a migration is sufficient to understand how each solution works.

In Listing 3.1 we find the skeleton of that application from a purely MPI approach. All the examples across this section share the same skeleton. In the *main* function the data is initialized. After that, the *compute* function is invoked. In that function we find the main loop, where the real computation occurs and where malleability is implemented. Specifically, in this code we check whether there is a parent communicator (line 4), which would state that the processes are in the middle of a reconfiguration. If it is not the case, the execution continues with the calculation stage (line 12). However, if there is a parent communicator, processes have to receive the data from the processes in the initial communicator (lines 8-10) in order to continue the execution in the line 12.

In the computational stage (line 14), for each step we would use a function that emulates a resource request to a scheduler (line 16). The function, theoretically, would somehow return the

```
1  void main(int argc, char **argv) {
2    PCM_MPI_Init(&argc, &argv);
3    PCM_COMM_WORLD = MPI_COMM_WORLD;
4    PCM_Init(PCM_COMM_WORLD);
5    PCM_Status status = PCM_Process_status;
6    if (status == PCM_STARTED) {
7      step = 0;
8      /* Initialization */
9    } else {
10     PCM_Load(myRank, "step", &step);
11     PCM_Load(myRank, "dataSize", &dataSize);
12     PCM_Load(myRank, "data", data);
13   }
14   compute(data, dataSize, step);
15 }
16 void compute(double *data, int dataSize, int step) {
17   for (t = step; t < TIMESTEPS; t++) {
18     pcm_status = PCM_Status(PCM_COMM_WORLD);
19     if (pcm_status == PCM_MIGRATE) {
20       PCM_Store(myRank, "step", &t, PCM_INT, 1);
21       PCM_Store(myRank, "dataSize", &dataSize, PCM_INT, 1);
22       PCM_Store(myRank, "data", data, PCM_DOUBLE, dataSize);
23       PCM_COMM_WORLD = PCM_Reconfigure(PCM_COMM_WORLD, argv[0]);
24     }
25     /* Computation */
26   }
27 }
```

Listing 3.2: Pseudo-code of job reconfiguration using the PCM API.

node list where the processes should be spawned.

If the list is null, the computation continues normally (line 24). Nevertheless, a non-null return means that a reconfiguration has to take place. On this basis, the new processes are spawned (line 18) and the data is sent to them (lines 19-21). Once the data is received, the initial processes terminates their execution (line 22), letting the new ones continue with the execution.

3.2.3.1 PCM API

Following the example presented in [12], we have adopted malleability in our skeleton code using the PCM API (Listing 3.2). PCM wraps many of the MPI functions/variables, but the workflow is practically the same that we saw in Listing 3.1. In line 6, we check the *status*, so if there is a migration in progress, the data is loaded (lines 10-12)

The *compute* function (line 16) has also a lot of similarities with the pure MPI implementation. We use the API function "PCM_Status" in line 18, in order to know which reconfiguration action has been scheduled. Although in this case we are only considering the migration, the PCM API provides more reconfiguration actions. For this reason, data is "stored" and the reconfiguration is triggered (lines 20-23), which correspond to the spawning of processes and the data sending.

```
1  void main(int argc, char **argv) {
2    step = 0;
3    /* Initialization */
4    compute(data, dataSize, step);
5  }
6  void compute(double *data, int dataSize, int step) {
7    MPI_Info_create(&hints);
8    MPI_Info_set(hints, "ampi_load_balance", "sync");
9    for (t = step; t < TIMESTEPS; t++) {
10     /* Computation */
11     AMPI_Migrate(hints);
12   }
13 }
```

Listing 3.3: Pseudo-code of job reconfiguration using AMPI.

3.2.3.2 AMPI

AMPI, through Charm++, provides fully automated support for migrating MPI ranks among nodes in a system without any application-specific code at all[26]. In the example of Listing 3.3, we have used "isomalloc"[27], which allows every worker thread in the system to allocate slices of virtual memory for all user-level threads, enabling transparent migration of pointers into memory. For other ways of allocation, AMPI provides registration data mechanisms, as well as tools for data pack/unpack in order to perform the data redistribution among ranks.

Since we are assuming the implicit registration of data provided by "isomalloc" in the initialization (line 3), the *main* function initiates the computation in line 4. There, the reconfiguration is parametrized using an "MPI_Info" object (lines 7-8) and the iterations are initiated. For each iteration, the data is computed (line 10) and the migration mechanism is invoked (line 11).

3.2.3.3 Flex-MPI

The authors in [47] present an example code that we have ported to our skeleton. Flex-MPI is not only a malleability solution, but also a performance-aware framework able to monitor the execution performance in each iteration, in order to schedule the most appropriate reconfiguration action. This is why the resulting code presents a higher level of instrumentation.

In Listing 3.4 we show the malleable implementation of our sample code using Flex-MPI. At the beginning of the program, data is initialized and registered before the computation stage (lines 4-9). Then, in each computational step (line 13), the performance monitor is initiated and the computation performed (lines 14-15). With the gathered information during the step execution, a reconfiguration action is taken (line 16). Notice that after a reconfiguration, unlike the previous malleability solutions, the execution flow does not return to the *main* function, but it remains in *compute* for the rest of the run. Although, this study is focused on the migration action, Flex-MPI implements a simple procedure for removing processes in case of a shrink. Lines 17-18 check each process and terminate processes selected by the runtime.

[26] http://charm.cs.illinois.edu/manuals/html/ampi/manual.html
[27] http://charm.cs.illinois.edu/manuals/html/tcharm/manual-1p.html

```
1  void main(int argc, char **argv) {
2    MPI_Init(&argc, &argv);
3    step = 0;
4    /* Initialization */
5    XMPI_Get_wsize();
6    XMPI_Register(dataSize);
7    XMPI_Register(data);
8    XMPI_Register(step);
9    XMPI_Get_Shared_data();
10   compute(data, dataSize, step);
11 }
12 void compute(double *data, int dataSize, int step) {
13   for (t = step; t < TIMESTEPS; t++) {
14     XMPI_Monitor_init();
15     /* Computation */
16     XMPI_Eval_reconfiguration();
17     status = XMPI_Get_process_status();
18     if (status == EMPI_REMOVED)
19       break;
20   }
21 }
```

Listing 3.4: Pseudo-code of job reconfiguration using Flex-MPI.

3.2.3.4 Elastic MPI

A malleable application following a producer–consumer scheme using MPICH extensions was illustrated in the original paper presenting this approach [7]. Although the current project does not support a complete migration of processes (the node where `srun` is executed cannot be substituted because of an intrinsic limitation of the model), our pseudo-code in Listing 3.5 assumes this limitation has been removed.

The program starts with a new version of the original `MPI_Init`, which features a new parameter for malleability (line 2). This new parameter indicates if the process has been created by Slurm (line 3). If this is the case, the process probes if it has to be adapted to a new process layout (line 4) in order to carry out the reconfiguration and the data redistribution (line 6-8). The authors do not provide sufficient information about how to perform the data redistribution; however we know where it occurs (line 7 and 21). When the process is created by the launcher (line 11), the application performs the original initialization of variables .

Once the program is initiated, the execution continues in the computation stage (line 18). For each iteration, the processes probe if an adaptation is occurring (line 20), and if that is the case, the reconfiguration and the redistribution are leveraged (lines 22-24). Finally, the processes execute their operations in line 26.

Notwithstanding the variety of methods presented to adopt malleability in parallel scientific applications, none of them combines the features that we consider crucial in order to gain popularity among developers: i) automatic support for data transfers in job reconfigurations; and ii) a friendly syntax imported from parallel programming models such as OpenMP or MPI. Furthermore, for MPI-like syntax, the solutions must be based on the MPI standard without a dependency to any

```
void main(int argc, char **argv) {
  MPI_Init_adapt(&argc, &argv, &local_status);
  if (local_status == JOINING) {
    MPI_Probe_adapt(&adapt);
    if (adapt == ADAPT_TRUE) {
      MPI_Comm_adapt_begin();
      /* Data redistribution code */
      MPI_Comm_adapt_commit();
    }
  } else {
    if (local_status == NEW) {
      step = 0;
      /* Initialization */
    }
  }
  compute(data, dataSize, step, local_status);
}
void compute(double *data, int dataSize, int step, local_status) {
  for (t = step; t < TIMESTEPS; t++) {
    MPI_Probe_adapt(&adapt);
    if ((local_status == JOINING) || (adapt == ADAPT_TRUE)) {
      MPI_Comm_adapt_begin();
      /* Data redistribution code */
      MPI_Comm_adapt_commit();
    }
    /* Computation */
  }
}
```

Listing 3.5: Pseudo-code of job reconfiguration using Elastic MPI.

particular MPI library.

Part II

Remote GPUs Management

CHAPTER 4

rGPUs in an HPC Cluster

GPUs are currently used in data centers to reduce the execution time of compute-intensive applications. However, the use of GPUs presents several side effects, such as increased acquisition costs as well as larger space requirements. Furthermore, GPUs require a non-negligible amount of energy even while idle. Additionally, GPU utilization is usually low for most applications.

In this regard, the remote GPU virtualization mechanism could be leveraged to share the GPUs present in the computing facility among the nodes of the cluster. This can increase overall GPU utilization, thus reducing the negative impact of the increased costs mentioned before. Reducing the amount of GPUs installed in the cluster could also be possible.

In this chapter we analyze the performance attained by a cluster using the rCUDA remote GPU virtualization middleware together with the Slurm version with remote GPU support, from two points of view: (i) cluster throughput and GPU utilization are increased at the same time that energy consumption is reduced; (ii) cluster upgrades are made easier and cheaper just by attaching GPU-enabled servers to a non-GPU infrastructure. In addition, a series of remote GPU resource selection policies have been developed and are evaluated at the end of the chapter.

4.1 Increasing the Cluster Throughput

In this section we study the impact that using the remote GPU virtualization mechanism poses on the performance of a data center. To that end, we have executed several workloads in a cluster by submitting a series of randomly selected job requests to the Slurm queues. After job submission we have measured several parameters such as total execution time of the workloads, energy required to execute them, GPU utilization, etc. We have considered two different scenarios for workload execution. In the first, the cluster uses original CUDA libraries and therefore applications can only use those GPUs installed in the same node where the application is being executed. In this scenario, an unmodified version of Slurm has been used. In the second scenario we have made use of rCUDA and therefore an application being executed in a given node can use any of the GPUs available in the cluster. The modified version of Slurm [22] has been used so that it is possible to schedule the use of remote GPUs. These two scenarios will allow to compare the performance of a cluster using CUDA with that of a cluster using rCUDA.

In order to present the performance analysis, we first present the cluster configuration and the

Table 4.1: Configuration details for each application

Application	Configuration	Execution time	GPU Memory
GPU-Blast	1 6-thread process in 1 node	21 s	1599 MB
LAMMPS	4 single-thread processes in 4 different nodes	15 s	876 MB
mCUDA-MEME	4 single-thread processes in 4 different nodes	165 s	151 MB
GROMACS	2 12-thread processes in 2 different nodes	167 s	-
BarraCUDA	1 single-thread process in 1 node	763 s	3319 MB
MUMmerGPU	1 single-thread process in 1 node	353 s	2104 MB
GPU-LIBSVM	1 single-thread process in 1 node	343 s	145 MB
NAMD	4 12-thread processes in 4 different nodes	241 s	-

workloads used in the experiments.

4.1.1 Cluster Configuration

The testbed used in this study is based on the use of a cluster composed of 16 1027GR-TRF Supermicro servers. Each of the 16 servers includes two Intel Xeon E5-2620 v2 processors (six cores with Ivy Bridge architecture) operating at 2.1 gigahertzs (GHzs) and 32 gigabytes (GBs) of double data rate type 3 synchronous dynamic random-access memory (DDR3 SDRAM) at 1.6 GHzs. They also feature a Mellanox ConnectX-3 VPI single-port fourteen data rate (FDR) InfiniBand adapter connected to a Mellanox Switch SX6025 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. An NVIDIA Tesla K20 GPU is installed at each node. One additional node without GPUs has been leveraged to execute the central Slurm daemon responsible for scheduling jobs.

All the evaluation was performed using the rGPU selection policy "first remote", which prioritizes the selection of remote GPUs before local. This policy arranges all the rGPUs in an array, having the rGPUs hosted in the assigned nodes (if any) at the end of the array. The policy iterates the array checking if the current rGPU meets the requirements of GPU memory. As much as the policy finds resources, those are assigned to the job until fulfilling the request.

4.1.2 Workloads

The workloads generated aim to provide a representative range of results, for this reason we have leveraged these applications: GPU-Blast [77], LAMMPS [3], mCUDA-MEME [45], GROMACS [62], BarraCUDA [39], MUMmerGPU [40], GPU-LIBSVM [5] and NAMD [60]. Table 4.1 provides additional information about the applications used in this work, such as the exact execution configuration used for each of the applications, their execution time, and the GPU memory required by each application. For the multi-thread/multi-process applications, the amount of GPU memory depicted in Table 4.1 refers to the individual needs of each particular thread or process. Notice that the amount of GPU memory is not specified for the GROMACS and NAMD applications because we are using non-accelerated versions of these applications. The reason for this choice is simply to increase the heterogeneity degree of the workloads by using some CPU-only applications, as it could be the case in many data centers in production. The previous applications have been combined in order to create three different workloads as shown in Table 4.2.

As can be seen, the eight applications used present different characteristics, not only in the number of processes and threads used by each of them and their execution time but also in the

Table 4.2: Workloads composition (number of jobs per application)

Application	Workload		
	Set 1	Set 2	Set 1+2
GPU-Blast	112	-	57
LAMMPS	88	-	52
mCUDA-MEME	99	-	55
GROMACS	101	-	47
BarraCUDA	-	112	51
MUMmerGPU	-	88	52
GPU-LIBSVM	-	99	37
NAMD	-	101	49
Total	400	400	400

Table 4.3: Slurm launching parameters

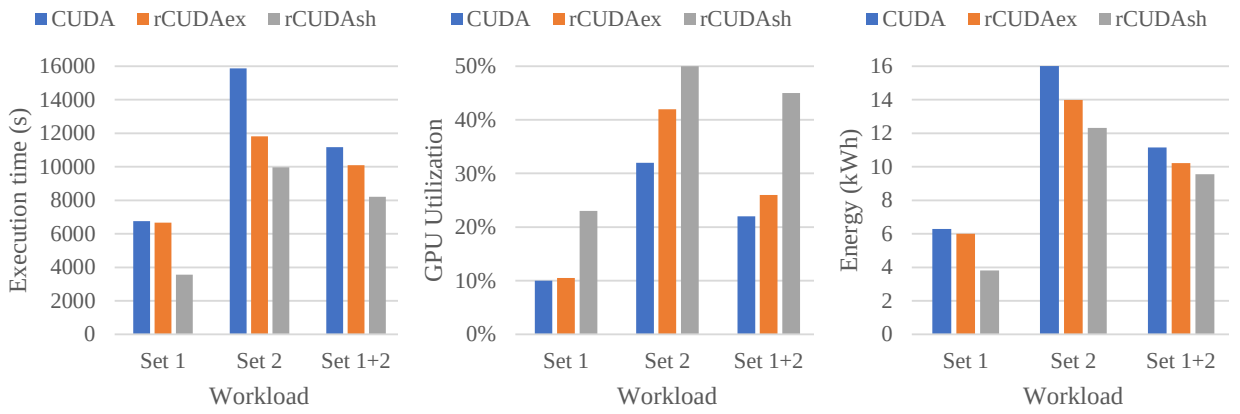
Application	Launch with CUDA	Launch with rCUDA exclusive	Launch with rCUDA shared
GPU-Blast	-N1 -n1 -c6 –gres=gpu:1	-n1 -c6 –rcuda-mode=excl –gres=rgpu:1	-n1 -c6 –rcuda-mode=shar –gres=rgpu:1:1599M
LAMMPS	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –rcuda-mode=excl –gres=rgpu:4	-n4 -c1 –rcuda-mode=shar –gres=rgpu:4:876M
mCUDA-MEME	-N4 -n4 -c1 –gres=gpu:1	-n4 -c1 –rcuda-mode=excl –gres=rgpu:4	-n4 -c1 –rcuda-mode=shar –gres=rgpu:4:151M
GROMACS	-N2 -n2 -c12	-N2 -n2 -c12	-N2 -n2 -c12
BarraCUDA	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –rcuda-mode=excl –gres=rgpu:1	-n1 -c1 –rcuda-mode=shar –gres=rgpu:1:3319M
MUMmerGPU	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –rcuda-mode=excl –gres=rgpu:1	-n1 -c1 –rcuda-mode=shar –gres=rgpu:1:2104M
GPU-LIBSVM	-N1 -n1 -c1 –gres=gpu:1	-n1 -c1 –rcuda-mode=excl –gres=rgpu:1	-n1 -c1 –rcuda-mode=shar –gres=rgpu:1:145M
NAMD	-N4 -n48 -c1	-N4 -n48 -c1	-N4 -n48 -c1

GPU usage patterns, what includes both memory copies to/from GPUs and also kernel executions. Therefore, although the set of applications considered is finite, it should provide a representative sample of a workload typically found in current data centers. We focus on the amount of resources required by each application and the time that those resources are kept busy.

Table 4.3 displays the Slurm parameters used for launching each of the applications. The use of real and virtual GPUs has been considered in the table. Notice that once Slurm has been enhanced, its users are able to submit jobs to the system queues in three different modes: (1) CUDA: no change is required to the original way of launching jobs; (2) rCUDA exclusive: the job will use the new remote virtual GPUs but it will not share them with other jobs; and (3) rCUDA shared: the job will use remote virtual GPUs, which will be shared with other jobs. In the first case, CUDA will be used (column labeled "Launch with CUDA"). In the second and third cases, rCUDA will be leveraged. In the second approach, the column labeled as "Launch with rCUDA exclusive" shows that no GPU memory is explicitly requested because the GPU assigned to a given job will not be shared with other jobs. In the third case, the column labeled as "Launch with rCUDA shared" shows that the amount of memory required at each GPU must be specified in the submission command.

4.1.3 Analysis of Cluster Performance

Figure 4.1 shows, for each of the workloads depicted in Table 4.2, the performance when CUDA is used along with the original Slurm job scheduler (results labeled as "CUDA") as well as the performance when rCUDA is used in combination with the modified version of Slurm. In this



(a) Workload completion time. (b) Average GPU utilization. (c) Total energy consumed.

Figure 4.1: Performance results from the 16-node 16-GPU cluster.

case, label "rCUDAex" refers to the results when remote GPUs are used in an exclusive way by applications whereas label "rCUDAsh" refers to the case when remote GPUs can be shared among several applications. Among both rCUDA uses, the shared variant is the most interesting because its flexibility. Figure 4.1a shows total execution time for each of the workloads. Figure 4.1b depicts the averaged GPU utilization for all the 16 GPUs in the cluster. Data for GPU utilization has been gathered by polling each of the GPUs in the cluster once every second and afterward averaging all the samples after completing workload execution. The `nvidia-smi` command was used for polling the GPUs and extracting the data. In a similar way, Figure 4.1c shows the total energy required for completing the workload execution. Energy has been measured by polling once every second the power distribution unitss (PDUs) present the cluster[1] After workload completion, the energy required by all servers was aggregated to provide the measurements in Figure 4.1c.

As can be seen in Figure 4.1a, workload "Set 1" presents the smallest execution time, given that it is composed of the applications requiring the smallest execution times. Furthermore, sharing the accelerators (rCUDAsh) is translated in a reduction in execution time for the three workloads, since more jobs can be concurrently executed. In this regard, execution time is reduced by 48%, 37%, and 27% for workloads "Set 1", "Set 2", and "Set 1+2", respectively. Notice also that the use of remote GPUs in an exclusive way also reduces execution time. In the case for "Set 2" this reduction is more noticeable because when CUDA is used the NAMD application (with 101 instances in the workload) spans over 4 complete nodes thus blocking the GPUs in those nodes, which cannot be used by any accelerated application during the entire execution time of NAMD (241 seconds). On the contrary, when "rCUDAex" is leveraged, the GPUs in those four nodes are accessible from other nodes and therefore they can be used by other applications being executed at other nodes. Regarding GPU utilization, Figure 4.1b shows that the use of remote GPUs helps to increase overall GPU utilization. Actually, when "rCUDAsh" is used with "Set 1" and "Set 1+2", average GPU utilization is doubled with respect to the use of CUDA. Finally, total energy consumption is reduced accordingly, as shown in Figure 4.1c, by 40%, 25%, and 15% for workloads "Set 1", "Set 2", and "Set 1+2", respectively.

Several are the reasons for the benefits obtained when GPUs are shared across the cluster. First, as already mentioned, the execution of the non-accelerated applications makes that GPUs in the nodes executing them remain idle when CUDA is used. On the contrary, when rCUDA is leveraged,

[1] Used units are APC AP8653 PDUs, which provide individual energy measurements for each of the servers connected to them.

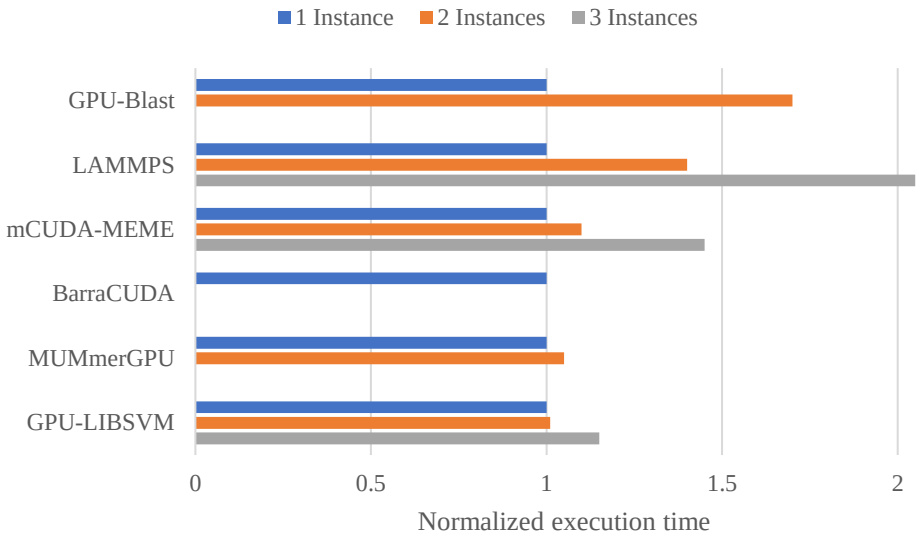


Figure 4.2: Normalized execution time when several instances run concurrently.

these GPUs can be used by applications being executed in other nodes of the cluster. Notice that this remote usage of GPUs belonging to nodes with busy central processing units (CPUs) will be more frequent as cluster size increases because more GPUs will be blocked by non-accelerated applications (also depending on the exact workload). Another example is the execution of LAMMPS and mCUDA-MEME, which require 4 nodes with one GPU. While these applications are being executed with CUDA, those 4 nodes cannot be used by any other application from Table 4.1: on the one hand, the other accelerated applications cannot access the GPUs in those nodes because they are busy; on the other hand, the non-GPU applications (GROMACS and NAMD) cannot use those nodes because they require all the CPU cores and LAMMPS and mCUDA-MEME already took one core. However, when GPUs are shared among several applications, GPUs assigned to LAMMPS and mCUDA-MEME can also be assigned to other applications that will run in any available CPU in the cluster, thus increasing overall throughput.

The second reason for the improvements shown in Figure 4.1 is related to the usage that applications make of GPUs. As Table 4.1 showed, some applications do not completely exhaust GPU memory resources. For instance, applications mCUDA-MEME and GPU-LIBSVM only use about 3% of the memory present in the NVIDIA Tesla K20 GPU. However, the unmodified version of Slurm (combined with CUDA) will allocate the entire GPU for executing each of these applications, thus causing that almost 100% of the GPU memory is wasted during application execution. This concern is also present for other applications in Table 4.1. Moreover, if NVIDIA Tesla K40 GPUs were used instead of the NVIDIA Tesla K20 devices employed in this study, then this memory underutilization would be worse because the K40 model features 12 GB of memory instead of the 5 GB of the Tesla K20 devices. On the contrary, when rCUDA is used in a shared way, GPUs can be shared among several applications provided that there is sufficient memory for all of them. Obviously, GPU cores will have to be multiplexed among all those applications, what will cause that all of them execute slower.

In this regard, Figure 4.2 presents the execution times for the GPU-accelerated applications in Table 4.1 when several instances of the same application are concurrently executed in a GPU. Although, it is also possible to analyze concurrent executions when the applications concurrently using the GPU are different, using several instances of the same application generates a higher pressure on the system because all the instances will try to synchronously perform the same operations.

Executions in Figure 4.2 have been manually constrained to a single node using CUDA. For some of the applications, only 2 concurrent instances were executed because of their larger memory requirements. In a similar way, BarraCUDA does not allow the concurrent execution of other instances because of its high memory requirements. As shown, executing several instances of the same application reports a speed-up for all of them: LAMMPS achieves the smallest one whereas GPU-LIBSVM obtains significant benefits. In summary, sharing a GPU among several applications reduces the total execution time. This reduction makes that combining rCUDA with the modified version of Slurm results in important reductions in the time required to complete workload execution.

Another possible point of view related to sharing GPUs among applications is that all the applications sharing the GPU run slower because they have to share the GPU cores. However, despite of the slower execution of each individual application, the entire workload is completed earlier, as shown in Figure 4.1. This means (1) that the time spent by applications waiting in the Slurm queues is reduced and (2) the execution of each individual application is completed earlier.

4.2 Upgrading a non-GPU Cluster

The use of GPUs in a cluster usually poses several burdens on the physical configuration of the nodes. For instance, nodes owning a GPU need to include larger power supplies able to provide the energy required by the accelerators. Also, GPUs are not small devices, and therefore, they require a non-negligible amount of space in the nodes where they are installed. These requirements make that installing GPUs in a cluster that did not initially include them may be expensive (power supplies need to be upgraded) or simply impossible (nodes do not have sufficient physical space for the GPUs). However, the workload in some data centers may evolve towards the use of GPUs. At that point, the concern is how to address the introduction of GPUs in a computing facility that did not include accelerators at acquisition time.

One possible solution to the above concern is acquiring some servers populated with GPUs and divert the execution of accelerated applications to those nodes. The Slurm RMS would automatically dispatch the GPU-accelerated applications to the new servers. However, although this approach is feasible, it presents the limitation that GPU jobs will probably have to wait for long until one of the GPU-enabled servers is available even though GPU utilization is usually low. Another concern is that accelerated MPI applications will only be able to span to as many nodes as GPU-enabled servers were acquired. Given these concerns, a better approach would be to acquire some servers populated with GPUs and use rCUDA to execute accelerated applications at any of the nodes in the cluster while using the GPUs in the new servers. This solution would not only increase overall GPU utilization with respect to the use of CUDA in the previous scenario but will also allow MPI applications to span to as many nodes as required because MPI processes would be able to remotely access GPUs thanks to rCUDA. In summary, the remote GPU virtualization mechanism allows clusters that did not initially include GPUs to be easily and cost-efficiently updated for using GPUs by attaching to them one or more computers containing GPUs. In this way, the original nodes will make use of the GPUs installed in the new nodes, which will become GPU servers. The modified version of Slurm would be used to schedule the use of the GPUs in the new servers.

To analyze the performance of these 2 possible solutions, we have substituted 1 of the nodes in the testbed cluster by a node containing 4 GPUs. This node is based on the Supermicro SYS7047GR-TRF server, populated with 4 NVIDIA Tesla K20 GPUs and 1 FDR InfiniBand network adapter. To additionally consider the use of parallel applications to be able to increase the heterogeneity of the workloads, we have modified the workloads used in the previous experiments by modeling distributed-memory applications with 2 and 4 threads, each one requiring a GPU. To that end,

Table 4.4: Composition of two additional workloads (number of jobs per application)

Application	Workload	
	WL1	WL2
GPU-Blast	41	48
LAMMPS short	39	46
LAMMPS long 2p	20	10
LAMMPS long 4p	20	10
mCUDA-MEME short	39	46
mCUDA-MEME long 2p	20	10
mCUDA-MEME long 4p	20	10
GROMACS	40	40
BarraCUDA	40	47
MUMmerGPU	41	47
GPU-LIBSVM	40	46
NAMD	40	40
Total	400	400

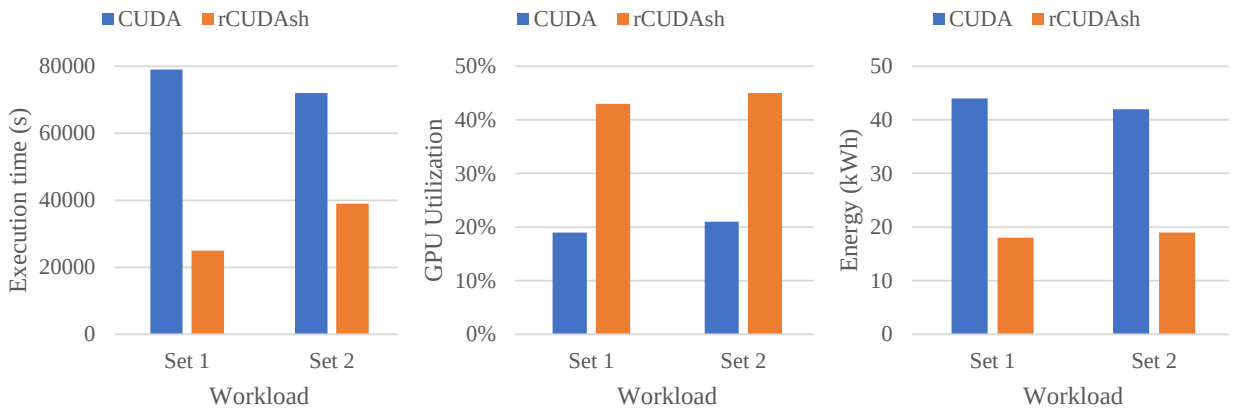
2 different flavors of the LAMMPS and mCUDA-MEME applications have been used, as shown in Table 4.4: 1) "LAMMPS long 2p" and "mCUDA-MEME long 2p" consist of 2 single-threaded processes that are forced to be executed in the same node. 2) "LAMMPS long4p" and "mCUDA-MEMElong4p" consist of 4 single-threaded processes that will be forced to execute in the same node. One additional flavor of these applications will model single-thread shared-memory applications. This additional flavor is composed by the "LAMMPS short" and "mCUDA-MEME short" cases shown in Table 4.4, which make use of 1 single-threaded process. Furthermore, small input data sets are used for the "LAMMPS short" and "mCUDA-MEME short" cases whereas the multi-threaded flavors use a large input data set to lengthen their execution time.

Figure 4.3 shows the performance results when a server with 4 GPUs has been attached to a cluster without GPUs. The original cluster is composed of 15 nodes (same node configuration as in the previous subsections, but GPUs have been removed). Results show that decoupling GPUs from nodes with rCUDA allows applications to make a much more flexible usage of the resources in the cluster (doubling their utilization), execution time is reduced up to 70%, and the energy consumption is halved.

4.3 Remote GPU Resource Selection Policies

After having proved the remarkable benefits of using remote GPUs in an HPC cluster, in this section are introduced, described and evaluated 6 complementary rGPU resource selection policies developed for this PhD dissertation. The first 4 policies only take into account the GPU memory for the scheduling. The fifth considers the number of jobs hosted by each GPU, while the last not only selects rGPUs, but also the target nodes. The rGPU selection policies are:

- **First local**: in this policy all the rGPUs are listed in an array, having the rGPUs hosted in the assigned nodes (if any) at the beginning of the array. With this procedure, the policy prioritizes the selection of local GPUs. The policy iterates the array checking if the current



(a) Total execution time of workload. (b) Average GPU utilization. (c) Total energy consumed.

Figure 4.3: Performance results when a 4-GPU server is attached to a 15-node cluster without GPUs.

rGPU meets the requirements of GPU memory. As much as the policy finds resources, those are assigned to the job until fulfilling the request.

- **First remote**: described in 4.1.1, this a variation of the previous policy, where the rGPUs hosted in the assigned nodes (if any) are appended to the last positions of the array. With this procedure, the policy prioritizes the selection of remote GPUs.

- **Worst fit**: this is the first policy that inspects the status of all the registered rGPUs before assigning the resource to the job. Particularly, "worst fit" searches for the rGPU with the highest amount of free memory. The goal of this policy is to distribute an equivalent amount of the load among all the rGPU resources.

- **Best fit**: with a similar approach to "worst fit", "best fit" aims at compacting the load in the smallest number of rGPUs as possible. The algorithm searches for the most loaded GPUs (in terms of allocated memory) and tries to fit the new job in the available memory.

- **Fewer jobs GPU**: the algorithm of this policy iterates all the available rGPUs in the cluster, looking for the devices with the lower number of running jobs. In the case of parity, the "worst fit" policy is applied.

- **Fewer jobs node**: this policy not only considers the GPU status, but also the number of running jobs in the assigned nodes. The goal of this policy is to distribute the network traffic concentrated in a given node. This traffic could be generated by the remote access to the GPU, as well as MPI communications. Nodes are selected considering their number of assigned jobs. Hence, nodes handling a lower number of jobs will be selected by the policy. Furthermore, this policy is compatible with any of the previous rGPU selection policies.

All the described policies are based on the Slurm `select/cons_res` selection plug-in, so the non-rGPU resources are managed in the same way, expect for the last policy.

For the evaluation of the implemented policies we only used the GPU-enabled applications of Table 4.1, having as a result the workloads described in Table 4.5. Regarding the hardware, we leveraged the 16-node GPU-enabled platform described in Section 4.1.1.

Table 4.5: Workloads composition (number of jobs per application)

Application	Workload		
	Set 1	Set 2	Set 1+2
GPU-Blast	132	-	66
LAMMPS	138	-	71
mCUDA-MEME	130	-	68
BarraCUDA	-	142	62
MUMmerGPU	-	131	59
GPU-LIBSVM	-	127	74
Total	400	400	400

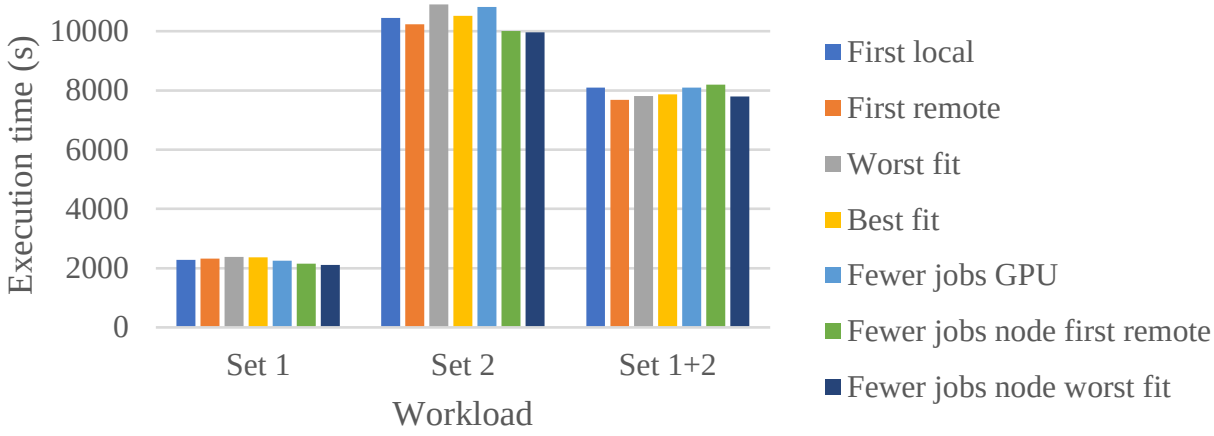


Figure 4.4: Performance results of the three workloads using different rGPU selection policies.

Figure 4.4 depicts the performance results of each selection policy grouped by workload set. The node-aware policy "fewer jobs node" was configured with two different rGPU selection plug-ins: "first remote" and "worst fit".

With a small variation in time, the results show an inconclusive evidence of a "winner" policy. In general, the node-aware policies "fewer job node" seem to perform better due to the fair distribution of jobs among the nodes, preventing the network interface saturation of the host that executes many jobs. Apart from that, "first remote" also shows a slightly time improvement in sets 2 and 1+2, since selecting remote devices lightens the burden of the host.

4.4 Conclusions

The chapter demonstrates the benefits of adopting a GPU virtualization technology, as it is the rCUDA middleware, into a cluster managed by a rGPU-aware RMS. Thus, we have carried out a thorough performance evaluation of a cluster using a modified version of Slurm which is able to manage the use of the rGPUs.

The improvements attained in execution time for a batch of jobs have been quantified. The associated reduction in energy consumption has also been presented. These features may be interesting in the context of exascale computing facilities given that one of the walls in this area is the

hard power consumption limitation.

Furthermore, it is also expected that as GPUs feature larger memory sizes, the benefits presented in this section will become more remarkable.

Regarding the policies, we conclude that there is no clearly outstanding policy that performs remarkably better for any kind of workloads. We assume this behavior to the simplicity of them and the lack of additional scheduling information.

CHAPTER 5

rGPUs in the Cloud

GPUs increment the computational power of a node thanks to their parallel architecture. This trend has led cloud service providers such as Amazon or cloud operating systems such as OpenStack to add VMs including this kind of accelerators to their facilities. To fulfill these needs, the guest hosts must be equipped with GPUs which, unfortunately, will be barely utilized if a non GPU-enabled VM is running in the host. The solution presented in this section is based on remote access to GPUs and how they can be shared in order to reach an equilibrium between service supply and the applications' demand of accelerators. Particularly, we propose to decouple real GPUs from the nodes by using the virtualization technology rCUDA. With this software configuration, GPUs can be accessed from any VM avoiding the need of placing a physical GPUs in each guest host.

For this purpose, we analyze from 2 perspectives the impact of adopting rGPUs in cloud environments:

- we study and deploy a rGPU-enabled cluster through a public cloud computing provider; and
- we design and develop the support and the logic to provide effective and secure access to the rGPUs in a private cloud infrastructure.

The results demonstrate this is a viable configuration which adds flexibility to current and well-known cloud solutions.

5.1 Public Cloud

Many cloud vendors have started offering VMs with GPUs in order to provide general-purpose computing on graphics processing units (GPGPU) computation services. A few relevant examples include AWS[1], Penguin Computing[2], Softlayer[3] and Microsoft Azure[4]. In the public scope, one of the most popular cloud vendors is AWS, which offers a wide range of pre-configured instances ready to be launched.

[1] `https://aws.amazon.com`
[2] `http://www.penguincomputing.com`
[3] `http://www.softlayer.com`
[4] `https://azure.microsoft.com`

Table 5.1: AWS HPC instances available in *US EAST (N. VIRGINIA)*, June 2015.

Name	virtual CPUs (vCPUs)	Memory	Network	GPUs	Price/h
c3.2xlarge	8	15 GiB	High	0	$ 0.42
c3.8xlarge	32	60 GiB	10 Gb	0	$ 1.68
g2.2xlarge	8	15 GiB	High	1	$ 0.65
g2.8xlarge	32	60 GiB	10 Gb	4	$ 2.6

Table 5.2: Network performance results using IPERF.

Server	Client	Network	Bandwidth
g2.8xlarge	c3.2xlarge	High	1 Gb/s
g2.8xlarge	c3.8xlarge	10 Gb	7.5 Gb/s
g2.8xlarge	g2.2xlarge	High	1 Gb/s
g2.8xlarge	g2.8xlarge	10 Gb	7.5 Gb/s

5.1.1 Amazon Web Services Features

AWS is a public cloud computing provider, offering several services, such as cloud-based computation, storage and other functionality. AWS enables organizations and/or individuals to deploy services and applications on demand. These services replace company-owned local information technology (IT) infrastructure and provide agility and instant elasticity matching perfectly with enterprise software requirements. From the point of view of HPC, AWS offers high-performance facilities via instances equipped with GPUs and high-performance network interconnections.

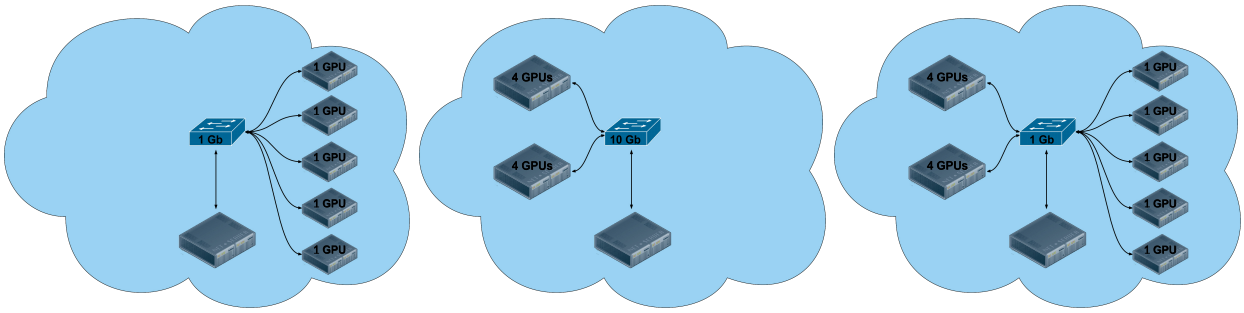
Instances An instance is a pre-configured VM focused on a specific target. Among the large list of instances offered by AWS, we can find specialized versions for general-purpose (T2, M4 and M3); computer science (C4 and C3); memory (R3); storage (I2 and D2) and GPU capable (G2). Each type of instance has its own purpose and cost (price). Moreover, each type offers a different number of CPUs as well as network interconnection, which can be: low, medium, high or 10 gigabit (Gb). For our study, we worked in the AWS availability zone *US EAST (N. VIRGINIA)*. The instances available in that case present the features reported in Table 5.1. For the following experiments, we select C3 family instances, which are not equipped with GPUs, as clients, whereas instances of the G2 family will act as GPU-enabled servers.

Networking Table 5.1 shows that each instance integrates a different network. Since the bandwidth is critical when GPU cloudification is applied, we first perform a simple test to verify the real network bandwidth. To evaluate the actual bandwidth, we executed the IPERF[5] tool among the instances described in Table 5.1, with the results shown in Table 5.2. From this experiment, we can derive that network "High" corresponds to a 1 Gb interconnect while "10 Gb" has a real bandwidth of 7.5 gigabits per second (Gb/s). Moreover, it seems that the bandwidth of the instances equipped with a "High" interconnection network is constrained by software to 1 Gb/s since the theoretical and real bandwidth match perfectly. The real gap between sustained and theoretical bandwidth can be observed with the 10 Gb interconnection, which reaches up to 7.5 Gb/s.

[5] `http://iperf.fr`

Table 5.3: Performance results of *bandwidthtest* transferring 32 MB with pageable memory in a local GPU.

Name	Data movement	Bandwidth
g2.2xlarge	Host to Device	3,004 MB/s
g2.2xlarge	Device to Host	2,809 MB/s
g2.8xlarge	Host to Device	2,814 MB/s
g2.8xlarge	Device to Host	3,182 MB/s



(a) 5 rGPUs over a 1 Gb network. (b) 8 rGPUs over a 10 Gb network. (c) 13 rGPUs over a 1 Gb network.

Figure 5.1: Schemes of the configured testbed scenarios using AWS instances.

GPUs An instance relies on a VM that runs on a real node with its own virtualized components. Therefore, AWS can leverage a virtualization framework to offer GPU services to all instances. Although the `nvidia-smi` command indicates that the GPUs installed are NVIDIA GRID K520, we need to verify that these are non-virtualized devices. For this purpose, we execute the NVIDIA software development kit (SDK) *bandwidthtest*. As shown in Table 5.3, the bandwidth achieved in this test is higher than the network bandwidth, which suggests that the accelerator is an actual GPU.

5.1.2 Testbed Scenarios

All scenarios are based on the maximum number of instances that a user can freely select without submitting a formal request. In particular, the maximum number for "g2.2xlarge" is 5; for "g2.8xlarge", it is 2. The instances operate the RHEL 7.1 64-bit OS and version 6.5 of CUDA. We design three configuration scenarios for our tests:

- Scenario A (Figure 5.1a) shows a common configuration in GPU-accelerated clusters, with each node populated with a single GPU. Here, a node can access 5 GPUs using the "High" network.

- Scenario B (Figure 5.1b) is composed of 2 server nodes, equipped with 4 GPUs each, and a GPU-less client. This scenario includes a 10 Gb network, and the client can execute the application using up to 8 GPUs.

- Scenario C (Figure 5.1c) combines scenarios A and B. A single client, using a 1 Gb network interconnection, can leverage 13 GPUs as if they were local.

Table 5.4: Performance results of *bandwidthtest* transferring 32 MB with pageable memory in a remote GPU.

Name	Data movement	Network	Bandwidth
A	Host to Device	High	127 MB/s
A	Device to Host	High	126 MB/s
B	Host to Device	10 Gb	858 MB/s
B	Device to Host	10 Gb	843 MB/s

Once the scenarios are configured, the rCUDA middleware needs to be installed in order to add the required flexibility to the system. The rCUDA server is executed in the GPU-enabled nodes and the rCUDA libraries are invoked from the node that acts as client. In order to evaluate the network bandwidth using a rGPU, we executed again the NVIDIA SDK *bandwidthtest*. Table 5.4 exposes that the bandwidth is limited by the network.

5.1.3 Experimental Results

The first application is `MonteCarloMultiGPU`, from the NVIDIA SDK, a code that is compute bound (its execution barely involves memory operations). This was launched with the default configuration, "scaling=weak", which adjusts the size of the problem depending on the number of accelerators. Figure 5.2 depicts the options per second calculated by the application running on the scenarios in Figure 5.1 as well as using local GPUs. For clarity, we have not represented the results observed in Scenario B, because they exactly match to those obtained in Scenario C up to 8 GPUs (Scenario C is an extension of Scenario B). In this particular case, rCUDA (using rGPUs) outperforms CUDA (local GPUs) because the former loads the libraries when the daemon is started [58]. With rCUDA we can observe differences in the results among both scenarios. Here, Scenario A can increase the throughput because the GPUs do not share the PCI bus with other devices because each node is only equipped with one GPU. On the other hand, when the 4-GPU instances ("g2.8xlarge") are added (Scenario C), the PCI bus constrains the communication bandwidth, hurting the scalability.

The second application, LAMMPS [3], needs at least one GPU to host its processes, but can benefit from the presence of multiple GPUs. Figure 5.3 shows that, for this application, the use of rGPUs does not offer any advantage over the original CUDA, because the four lines are almost overlapped. Furthermore, for the execution on remote GPUs, the difference among both networks is small, although the results observed with the "High" network are worse than those obtained with the "10 Gb" network. In the execution of LAMMPS on a larger problem (see Figure 5.4), CUDA still performs slightly better, but the interesting point is the execution time when using rGPUs. These are almost the same even with different networks, which indicates that the transfers turn the interconnection network into a bottleneck. For this type of application, enlarging the problem size compensates the negative effect of a slower network.

5.1.4 Conclusions

The previous experiments reveal that the AWS GPU-instances, at the moment of the evaluation (June 2015), are not appropriate for HPC because neither the network nor the accelerators are powerful enough to deliver high performance when running compute-intensive parallel applications. As [58] demonstrates, network and device types are critical factors to performance. In other words,

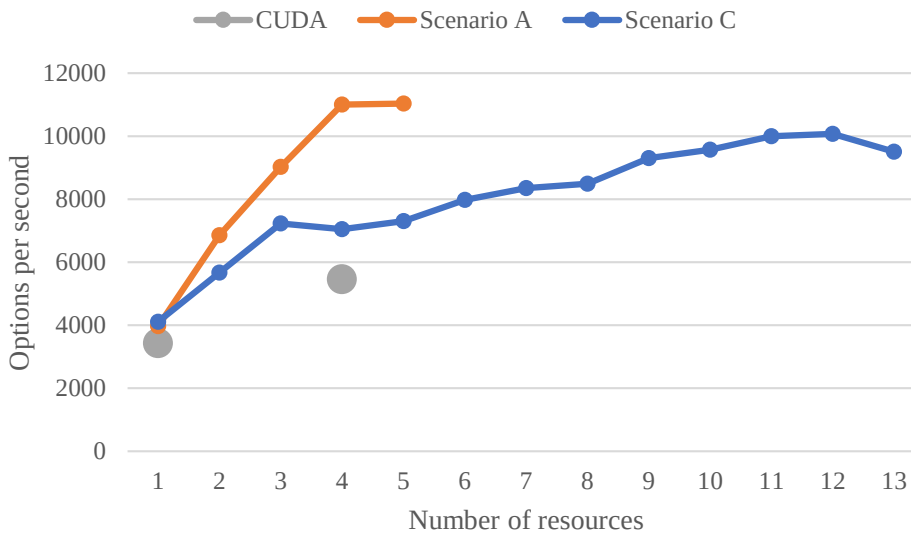


Figure 5.2: Throughput results of `MonteCarloMultiGPU` in terms of options per second.

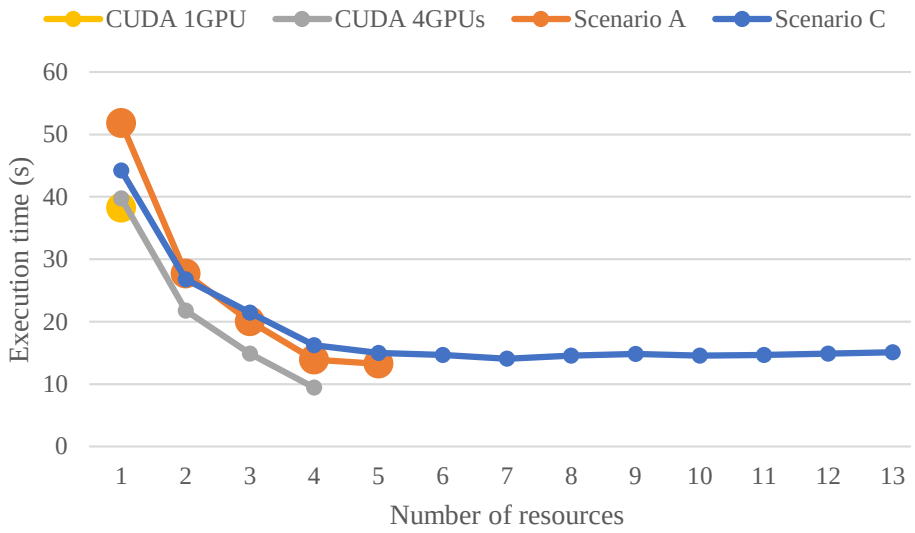


Figure 5.3: Performance results of LAMMPS with a *run size* of 100.

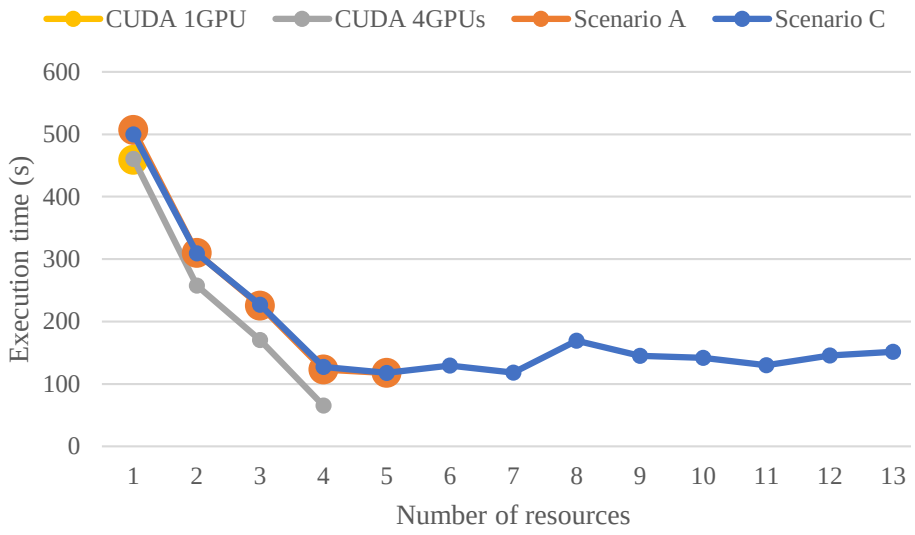


Figure 5.4: Performance results of LAMMPS with a *run size* of 2000.

AWS is more oriented toward offering a general-purpose service than to provide high performance. Also, AWS fails in offering flexibility, since it enforces the user to choose between a basic instance with a GPU and a powerful instance with 4 GPUs. Table 5.1 shows that the resources of the "g2.8xlarge" are quadrupled, but so it is the cost per hour. Therefore, in the case of having other necessities (instance types), using GPU virtualization technology we could attach an accelerator to any type of instance. Furthermore, reducing the budget spent in cloud services is possible by customizing the resources of the available instances. For example, we can work on an instance with 2 GPUs for only $1.3 by launching 2 "g2.2xlarge" and using rGPUs, avoiding to pay the double for features that we do not need in "g2.8xlarge". In terms of GPU-shareability, AWS reserves GPU-capable physical machines which will be waiting for a GPU-instance request. Investing in expensive accelerators to keep them in a standby state is counter-productive. It makes more sense to dedicate less devices, accessible from any machine, resulting in a higher utilization rate.

5.2 Private Cloud

The adoption of cloud computing in data centers offers new computational possibilities. From the end-user perspective, the cloud offers on-demand resources which can be easily provisioned and decommissioned on-the-fly. From the point of view of the cloud provider, virtualization, flexible resource availability, and shareability may lead to important economic competitiveness and a better exploitation of the infrastructure [51]. Although many types of resources have been adapted to this paradigm (i.e: CPUs, volatile/permanent memory devices, networks,...), others, such as GPUs, are not so flexible regarding cloud computing.

The main contribution of the work presented in this section is to provide a reliable service capable of deploying CUDA-enabled VMs through a secure and flexible access to the physical GPUs. This kind of access to the GPU is hereafter known as cloudified GPUs (cGPUs), since it has been specially designed for cloud platforms. The proposed solution combines two independent strategies: an architectural component to provide cGPUs, and a cGPU resource management.

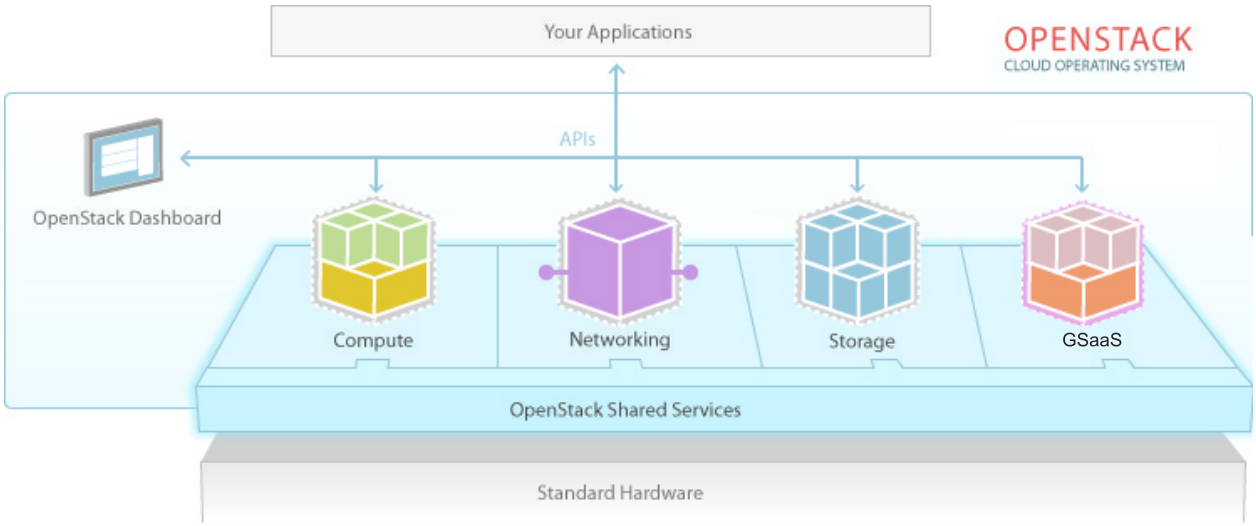


Figure 5.5: GSaaS: A service to cloudify and schedule GPU access in OpenStack.

5.2.1 GSaaS: A Service to Cloudify and Manage GPUs

GSaaS aims to provide a management layer between the GPUs and the cloud infrastructure. In this regard, it is completely integrated in the OpenStack architecture as a new shared service in charge of cloudifying and scheduling the access from VMs to physical GPUs, as depicted in Figure 5.5.

Besides the cloud platform and the GPGPU virtualization technology, we also need:

- a resource manager responsible for arbitrating the assignation of GPUs to the VMs, and
- a GPU masking mechanism which provides security when accessing remote GPUs.

Figure 5.6 shows the involved technologies in GSaaS. remote CUDA (rCUDA) enables the access to GPUs that are previously scheduled and assigned by the general purpose GPU management system (GPGPUMS). Additionally, the resource-oriented distributed virtual routing (RODVR) is introduced to cloudify GPUs and hide their location details from users, with the aim of providing a better integration in the cloud and avoiding unauthorized accesses to the devices.

The GSaaS components integration into the control plane of the cloud infrastructure is depicted in Figure 5.7. *Compute nodes* are standard OpenStack computation nodes that allocate tenant virtual machines. It is worth noting that these nodes do not include GPU devices, while *compute-GPU* nodes feature physical GPUs and may allocate VMs that use them locally as cGPUs. On the other hand, *compute-rGPU* nodes are connected to a *GPU network* that enables their hosted VMs to access remote cGPUs provided by the *compute-GPU* nodes. This dedicated GPU network avoids interference with the VM or management traffic in OpenStack. With RODVR, the access to the GPUs is secured and isolated from the rest of communication. With this purpose, an *RODVR Endpoint* is deployed in each node (*compute-GPU* or *compute-rGPU*) willing to host cGPU-enabled VMs, while rCUDA-server daemons are started only in *compute-GPU* nodes. The GSaaS service orchestrates the deployment of cGPU-enabled VMs by using the GPGPUMS module and interacting with the cloud infrastructure and with the *RODVR Endpoints* through the OpenStack management network.

While RODVR is responsible for the abstraction of the actual GPU location inside the VM configuration, GPGPUMS selects and assigns the most appropriate cGPUs according to a policy.

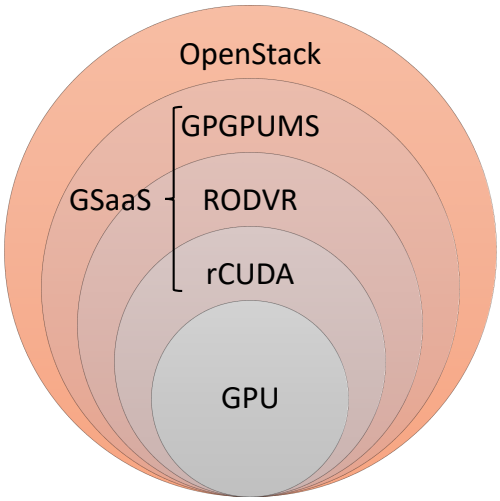


Figure 5.6: Technologies integration scheme.

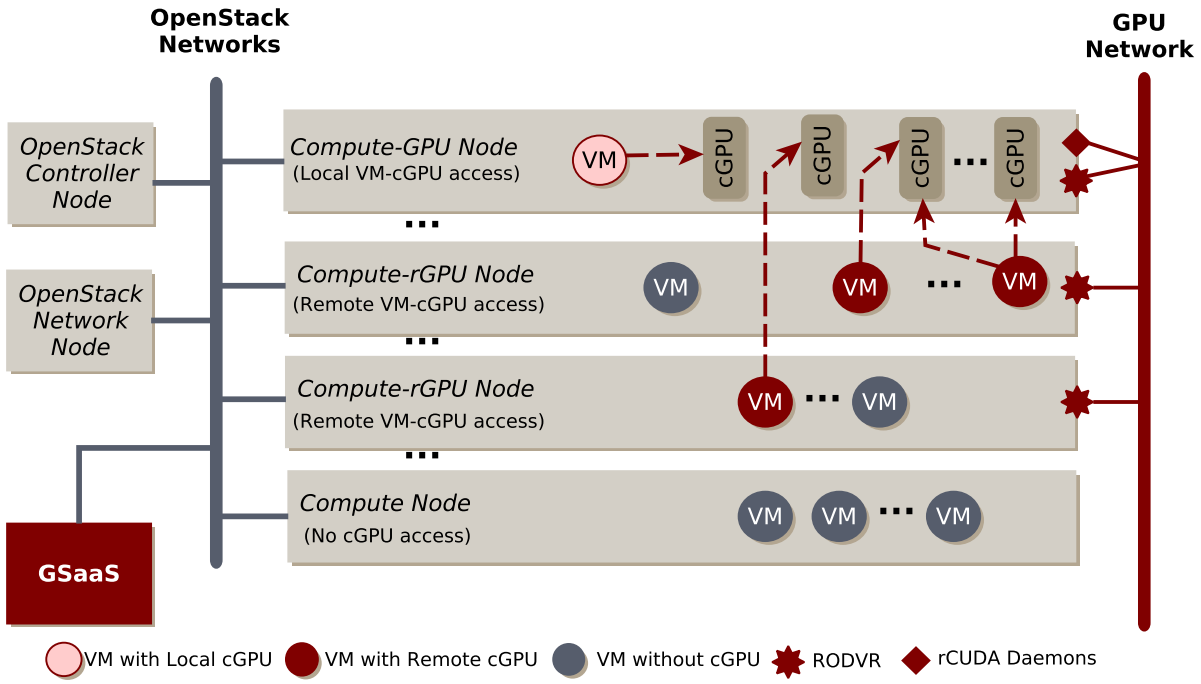


Figure 5.7: GSaaS components integration diagram, showing proposed node types and interconnection networks.

This configuration may also facilitate the migration of the VM to a different compute node, with (*compute-GPU*) or without (*compute-rGPU*) physical GPUs, transparently for the VM.

5.2.1.1 Booting a cGPU-enabled VM

Figure 5.8 depicts the activities performed when a request for a cGPU-enabled VM is submitted to GSaaS. This request provides information related to the VM instance (image, flavor, network interfaces, etc.) and related to the GPUs (number, memory, access mode, etc).

Initially, GPU requirements are checked by GPGPUMS and if these are satisfied, then a call to the OpenStack `Nova` service is performed to check if the VM requirements may be fulfilled. If the request is valid, the VM is created by providing a script to `cloud-init`, which is a multi-distribution package that handles early initialization of the cloud instance. This process is executed during the VM boot stage and sets rCUDA environment variables with the number of available cGPUs and their masked locations. These variables are needed to run the CUDA application in the VM.

Once the `Nova` agent starts the specified image (*active state*), the GSaaS service waits until the network is up in the machine (*deployment stage*). This ensures that the `Neutron` agent in the compute node has created all the networking infrastructure for the VM. Then, concurrently to the VM boot process, GSaaS invokes the GPGPUMS module to select the GPUs, and then, each GPU remote access is configured via an RODVR endpoint. The GPU information provided by GPGPUMS is used by RODVR to insert rules in the compute node with the aim of routing packets from the VM to the assigned GPUs. Due to these rules, packets that leave a virtual interface attached to the machine with a masked IP and port are rerouted to the real address of the host where the rCUDA daemon is running. When both stages (selection and configuration) have finished, the CUDA application in the VM is ready to be used.

Equation 5.1 defines the total boot time of a VM when assigning x cGPUs. The parameters of the equation correspond to the activities in Figure 5.8, where the *VM boot time* is the sum of both *checking times (GPU and VM)*, the *VM activation & creation times* and the higher time between the concurrent activities (*VM deployment* and *GPUs selection & RODVR configuration*).

$$\begin{aligned} &\textit{cGPU-enabled VM Boot Time}(x) = \\ &\textit{Check GPU \& VM Time} + \\ &\textit{VM Creation Time} + \textit{VM Activation Time} + \\ &\max(\textit{VM Deployment Time}, \\ &\textit{GPUs Selection Time}(x) + \textit{RODVR configuration Time}(x)) \end{aligned} \tag{5.1}$$

5.2.1.2 GPU Selection Policies

GPGPUMS includes a GPU selection policy to host cGPUs based on a *first fit* algorithm, which selects the first GPU in the pool of GPUs with sufficient available memory to fulfill the request. In cases of low need for GPU memory, this policy yields as a result a performance degradation, since GPUs are massively oversubscribed by the cGPUs. For this reason, we moved to other policies aimed to performance instead of energy-saving or resource-conservative, as it is *first fit*. Thanks to the modularity and the plugin-based architecture of GPGPUMS, we developed a *least load fit* policy that checks the available memory in the whole pool of GPUs and selects the device with the maximum amount of free memory. Algorithm 1 describes how decisions are made by our *least load fit* implementation. First of all, the GPUs are listed and sorted by their current available memory

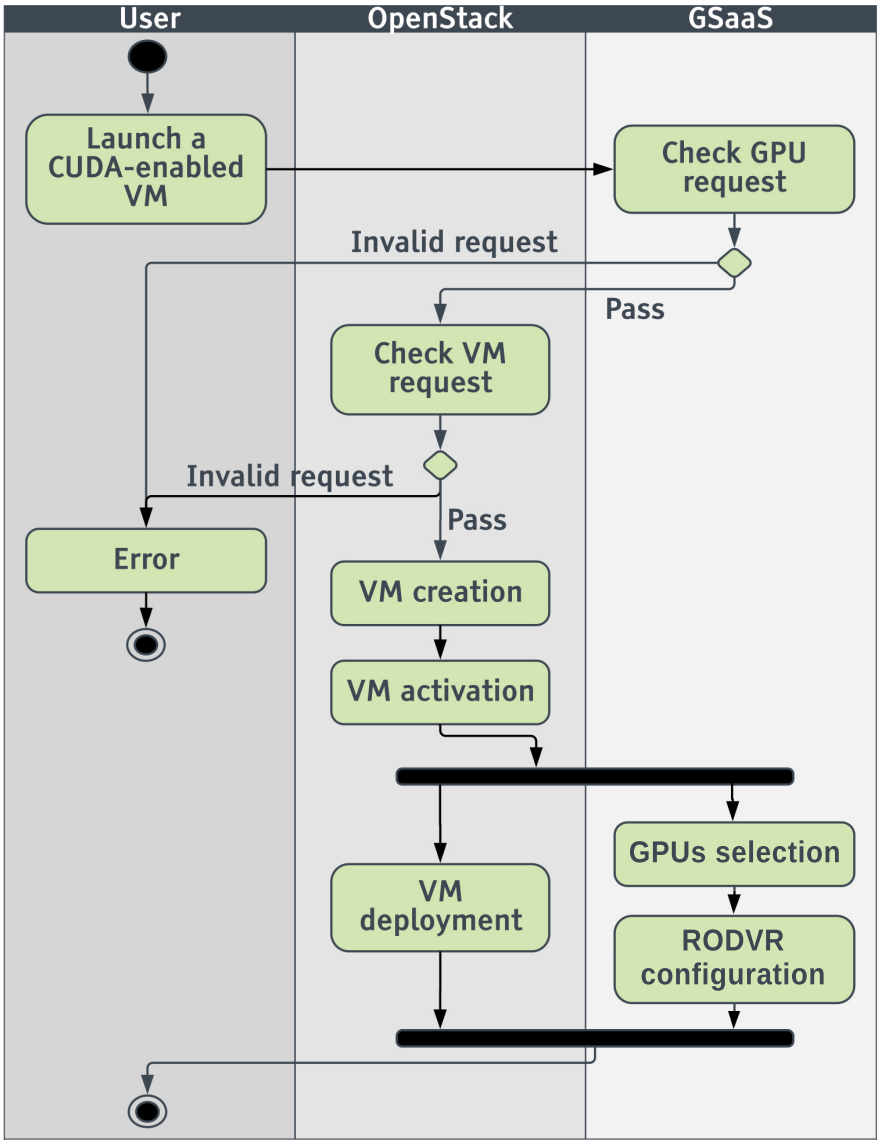


Figure 5.8: Activity diagram during the launch of a cGPU-enabled VM.

(line 2). Then, the list is iterated (line 3) in order to find a GPU with sufficient memory to host the cGPU (lines 4-7).

Algorithm 1 GPU selection policy based on a least load fit strategy.

```
1: function SELECT_GPU(job)
2:     gpu_sorted_list = GPUS.sort_desc_by(avail_mem)
3:     for each gpu in gpu_sorted_list do
4:         new_mem = gpu.avail_mem + job.req_mem
5:         if new_mem <= gpu.total_mem then
6:             gpu.update_status(job)
7:             return gpu
8:         end if
9:     end for
10:    return NULL
11: end function
```

5.2.1.3 Working Modes

GSaaS has been designed to implement the following working modes:

- **Remote-Exclusive**: A VM uses GPUs located in different nodes and monopolizes the use of the assigned GPUs.
- **Remote-Shared**: A VM uses GPUs located in different nodes, but these GPUs may be shared with other VMs.
- **Local-Shared**: A VM uses GPUs provided by the compute node in which it is allocated, but these GPUs may be shared with other VMs.
- **Local-Exclusive**: A VM uses GPUs provided by the compute node in which it is allocated, and it monopolizes the use of the assigned GPUs.

5.2.1.4 Capabilities and Usage Examples

Users are expected to use the command `gsaas` with different arguments in order to deploy their cGPU-enabled VMs. The following list of examples illustrates the usage of the command (information related to the VM is omitted for simplicity):

- `$ gsaas -l -ncgpus=2`: The basic invocation to launch a cGPU-enabled VM. The parameter `-l` stands for *launch* and `-ncgpus` determines the number of cGPUs associated to the VM. By default, cGPUs are only accessible by one VM (exclusive mode).
- `$ gsaas -l -ncgpus=2 -poolmem=2048 -mode=shared`: In this case, a user requests 2 cGPUs in shared mode (`-mode`) with a GPU-memory limitation of 2 GB (`-poolmem`). In other words, the scheduler (GPGPUMS) is in charge of managing the GPU memory in order to meet the request, by assigning 2 cGPUs of 2 GB to the VM.
- `$ gsaas -l -ncgpus=8 -poolmem=4096 -mode=shared -locality=local`: A user may also define the location of the VM with respect to the cGPUs. By default, VMs are deployed in compute hosts determined by the `Nova` service of OpenStack. However, if we have a special interest

in deploying a VM in the same compute node where the physical GPUs are hosted, we will use the argument `-locality`. In this example, a user is launching a VM with access to 8 cGPUs of 4 GB of GPU memory each one, in the same compute node where the accelerators are physically attached.

- `$ gsaas -l -ncgpus=2 -locality=local`: In this case, the VM is deployed in the host where the 2 GPUs are placed, having exclusive access to them.

- `$ gsaas -t -id=<VMid>`: With the argument `-t`, which stands for *terminate*, together with the identifier of a VM (`-id`), the user is able to stop and destroy an existent cGPU-enabled VM.

When requirements of launching arguments cannot be fulfilled, the deployment of the VM is aborted with an error (see *Check GPU request* activity in Figure 5.8).

5.2.2 Performance Evaluation

This section presents a set of experiments that evaluates and demonstrates the benefits provided by GSaaS. The experiments range from a detailed analysis of the VM deployment time to the scalability study of different type of applications like multi-GPU or distributed computation.

5.2.2.1 Experimental Setup

The experimental setup used to evaluate the features and the performance of our proposal is depicted in Figure 5.9. It is based on *OpenStack Ocata* with *Neutron* and the *ml2* plugin to define cloud networking[6], and adopts a hybrid cloud infrastructure approach [6]. Virtual and physical machines run an Ubuntu 16.04 operating system.

The master node (*master*) and the 3 *compute-rGPU* nodes (*compute[0–2]-rGPU*) features 2 Intel XEON E5-2630-v3 sockets (8 cores at 2.40 GHz each) with 32 GB of DDR3-2200 SDRAM and a 2 TB hard drive. Each processor provides 16 vCPUs due to the hyper-threading technology, and therefore 32 vCPUs are available per node.

The *master* node hosts the virtualized OpenStack controller and network nodes. Each virtualized node is configured with 8 vCPUs. The controller is equipped with 10 GB of RAM memory and the network node features 6 GB of RAM. The GSaaS elements have been deployed in a VM with 2 vCPUs and 2 GB of RAM as part of the hybrid infrastructure.

The *Compute-GPU* node has 2 Intel XEON E5-2603-v4 sockets (6 cores at 1.70 GHz) for a total of 12 cores with 32 GB of DDR-2133 SDRAM and a 1 TB hard drive. It is equipped with 4 NVIDIA Quadro M4000 GPUs featuring 8 GB of memory each. The *compute-GPU* node may allocate tenant VMs which make use of its local GPUs.

The 3 networks (*Management*, *VM*, and *GPU*) use 10 GbE network cards connected to 3 NetGear proSAFE Plus (XS708E) switches.

5.2.2.2 Performance Metrics

Metrics provided by the experimental configuration may be classified into three main groups according to their origin: metrics from tenant VMs, metrics from their hosts, and metrics from the GPUs (using the `nvidia-smi` tool). With the purpose of detecting overhead conditions, memory and processor utilization is measured for each physical and virtual machine. Moreover, the detailed deployment time when evaluating the booting process, or the execution time for each performance

[6] `https://docs.openstack.org/ocata/networking-guide/`

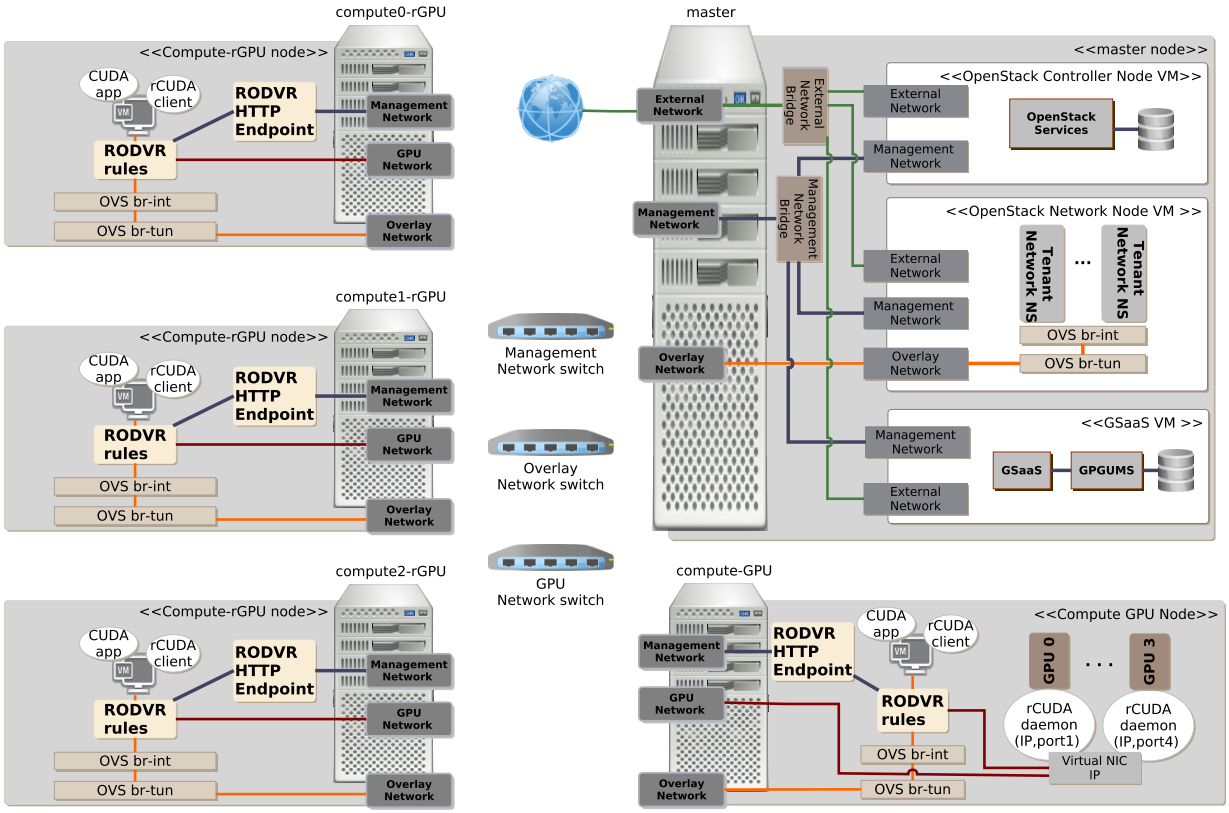


Figure 5.9: Experimental setup deployed in a hybrid cloud infrastructure based on OpenStack.

Table 5.5: Detailed time of GSaaS booting a VM with a different number of assigned cGPUs

Stage	Time in seconds			
cGPUs	0	25	50	75
GSaaS Checking	-	0.0047	0.0047	0.0046
VM Creation	0.6817	0.6740	0.7397	0.7515
VM Activation	1.2808	1.2626	1.2588	1.3121
VM Deployment	20.6710	20.6755	20.6769	20.6745
GSaaS Management	-	7.9014	15.6715	23.3833
Boot Time	22.6335	22.6368	22.7101	25.4815

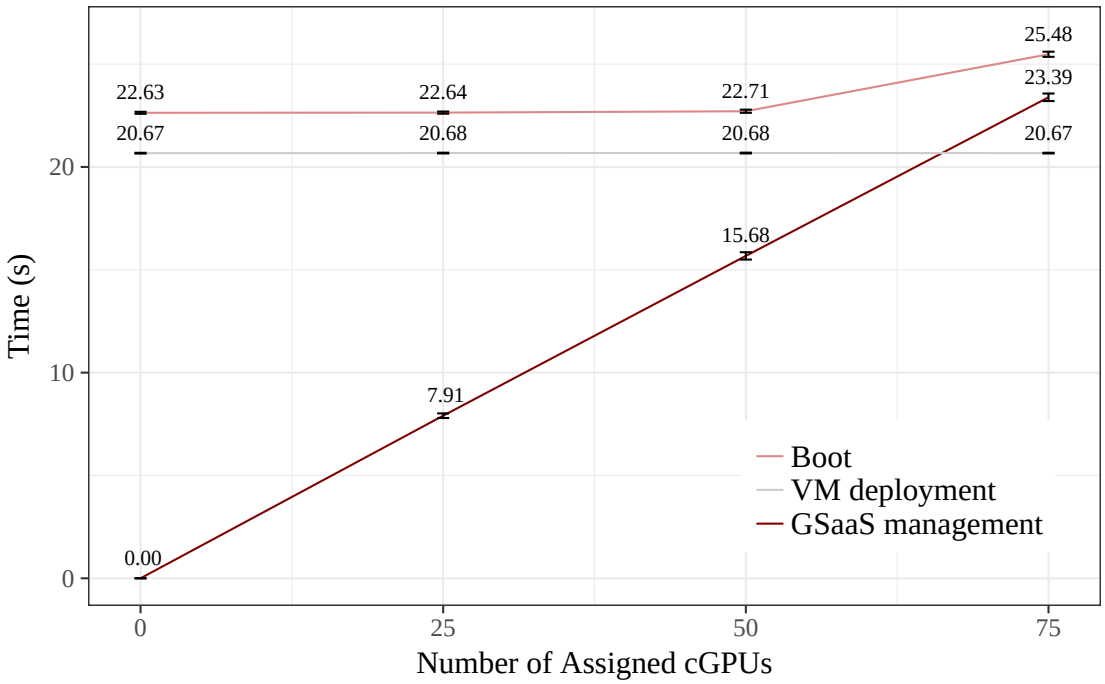


Figure 5.10: Boot time of a VM when increasing the number of assigned cGPU

test, are collected to characterize the user utilization pattern. The performance of each GPU is metered by the memory, the usage rate and the power consumption of the running processes. All the experiments were executed 50 times in order to obtain statistically significant measurements.

5.2.2.3 Infrastructure Deployment Evaluation

This section analyzes the scalability of deploying CUDA-enabled VMs from two different points of view: a VM with multiple cGPUs, and multiple single-cGPU VMs.

First, *cGPUs scalability* has been evaluated by assessing the detailed temporal cost of booting a VM depending on the number of cloudified GPUs assigned to it. Specifically, we have performed a series of experiments that measure each time considered in Equation 5.1 when booting a VM with 0 to 75 cGPUs in 25-cGPU steps. The VM is booted using a flavor of 1 vCPU, 768 MB RAM and each cGPU requiring 100 MB of memory. Results are shown in Table 5.5 and Figure 5.10.

The boot time of a cGPU-enabled VM is the elapsed time between the invocation of a `gsaas`

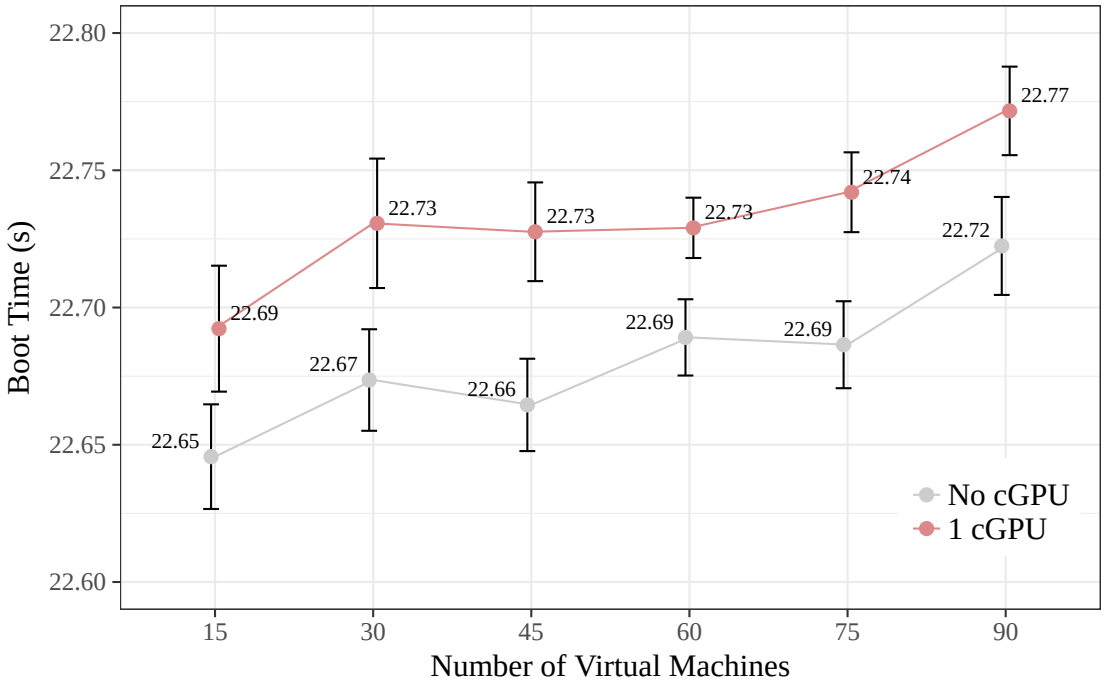


Figure 5.11: Boot time of non-GPU-enabled VMs compared to cGPU-enabled VMs.

command and the moment when the requested cGPUs are ready to be used. GSaaS checking times are not affected by the number of cGPUs, as shown in Table 5.5. VM creation and VM activation times experience a small increase of time with the number of cGPUs. The boot time is basically determined by the maximum between the time required by OpenStack to deploy the VM and the time required by GSaaS to select GPUs and configure the access to each cGPU in the compute node where the VM is allocated (named GSaaS management time). As can be seen in Figure 5.10, the GSaaS management time increases linearly with the number of cGPUs to configure. However, the VM deployment time is bigger than the GSaaS management time up to around 62 cGPUs. The overhead when assigning 50 cGPUs is almost negligible, since it represents only an increase of 0.2% in the boot time with respect to the base case of booting the VM without cGPU. In the case when 75 cGPUs are assigned, the GSaaS management time is bigger than the VM deployment time and the boot time experiences an overhead of 12%.

VMs scalability, on the other hand, has been evaluated by studying the deployment of multiple single-cGPU VMs (from 15 to 90) in 15-VM steps. Experiments use the three *compute-rGPU* nodes (*compute[0–2]-rGPU*) to fairly allocate the VMs, and they wait for 10 seconds among consecutive invocations to the `gsaas` command. Each VM is booted using the same flavour as in the previous experiment. Results are shown in Figure 5.11. The Y-axis represents a zoomed-in region of time, which indicates a negligible increment in the time, mainly due to the performance deterioration of the compute nodes that host all the VMs.

5.2.2.4 Experimental Results

Three different applications have been chosen in order to study different aspects of our proposal.

- First, the importance of the VM-GPU tenant locality and the GPU shareability is analyzed.

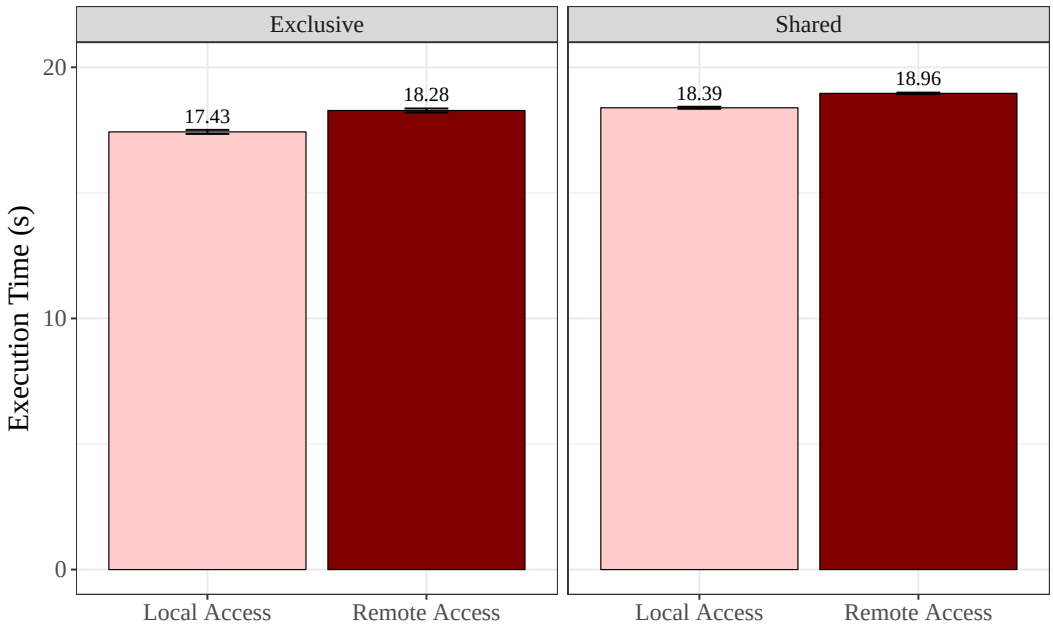


Figure 5.12: Average execution time of CUSHAW for exclusive and shared modes using local and remotes deployment localities.

- Then, a multi-GPU scenario, where all the cGPUs of a VM work together to solve a problem, is examined.
- Finally, a distributed CUDA-enabled application uses the MPI paradigm to parallelize a problem in several processes.

GPU Locality and Shareability In this experiment we leverage CUSHAW [44], a well-established leading next-generation sequencing read alignment CUDA-compatible software package. This problem allows us to check the behavior of GSaaS when there is a significant amount of data to be transferred from host to device and vice versa. The VM sends to the GPU the human reference genome (around 2.4GB) and the sequenced chromosome (in our case 240MB for the sample human chromosome Chr1) to be analyzed. When the alignment finishes, it receives the result (around 200 MB).

With this application we analyze the effect of the cGPU locality and the impact of sharing accelerators. For this purpose, 2 instances of CUSHAW are concurrently executed in a VM equipped with 4 vCPUs, 8 GB of memory and 2 cGPUs. While the *exclusive* cGPUs match 1-to-1 the GPUs, in the *shared* mode each cGPU allocates 4 GB of memory of the same physical GPU. Furthermore, we distinguish among two deployment locality options: *local* if the VM is deployed in the same host where the GPUs are installed (*compute-GPU*) or *remote* if these are in a non-GPU compute node (*compute-rGPU*).

Figure 5.12 depicts the total execution time of both CUSHAW instances running concurrently. Results of the exclusive scenario show an increment of 5% in execution time when the experiment is performed using remote access instead of local access, because the VM has to reach other compute host to transfer the data to and from the accelerator. The shared scenario presents a similar general behavior.

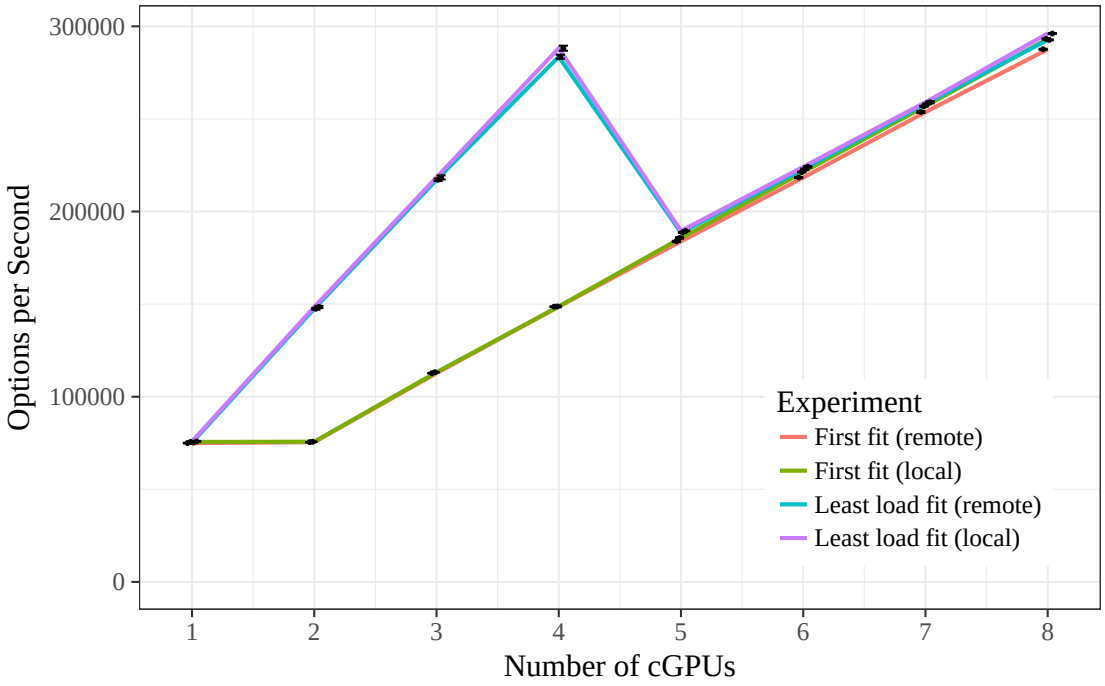


Figure 5.13: MonteCarloMultiGPU throughput using different cGPUs assignation policies.

It is worth noting that the execution time in the shared scenario does not double the time obtained in the exclusive scenario, because there are periods featuring different GPU usage. Therefore, it is possible to execute 2 CUSHAW works simultaneously during the low usage periods without experiencing a high performance penalty when considering remote locality or GPU shareability (see [59]).

Multi-GPU Computation and Scheduling Policies Multi-GPU applications leverage all the assigned cGPUs to the VM to perform their operations. A classical Multi-GPU application based in the Monte Carlo algorithm, widely used in this type of tests, is found in the NVIDIA SDK. *MonteCarloMultiGPU*[7] is a single-process application which evaluates fair call price for a given set of European options using a Monte-Carlo approach, benefiting from all CUDA-capable cGPUs assigned to the VM.

In this regard, we have deployed a VM with 30 vCPUs and 30 GB of RAM. Through GSaaS, we have progressively attached to the VM more cGPUs (with 4 GB of memory) in order to evaluate the productivity of *MonteCarloMultiGPU* when sharing GPUs. The VM can own up to 8 cGPUs (each GPU has 8 GB of memory) with this configuration.

Figure 5.13 depicts the number of options calculated per second (throughput) by *MonteCarloMultiGPU* when assigning an increasing number of cGPUs to the VM and considering different working modes (local and remote) and selection policies (first fit and least load fit).

As in the previous section, local-remote modes do not present any significant improvement. However, when comparing selection policies we appreciate significant differences in the results.

From the perspective of an end-user, the *least load fit* policy provides better results up to 4

[7] `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/samples.html#MonteCarloMultiGPU`

cGPUs. In this particular case, this policy selects 4 different GPUs to host the first 4 cGPUs, while *first fit* hosts the first 2 cGPUs in the same GPU, the third and forth cGPUs in the second GPU and so on, oversubscribing the accelerators earlier. *Least load fit* prioritizes devices with lower allocation of memory; that is why *least load fit* selects different accelerators (exclusive usage) for the cGPUs until they are finished (up to 4); then, from the 5th cGPU on, GPUs are oversubscribed.

On the other hand, the *first fit* policy depicts a more appealing approach for a cloud provider. This policy deploys cGPUs in one accelerator while it has enough memory to host them. With the requirement of 4 GB of memory per cGPUs, each physical device can host up to 2 cGPUs; hence we are using few GPUs at the expense of lower performance (fewer options calculated per second). However, depending on the pay-per-use price of the cGPUs, a user may sacrifice the performance for cost.

Distributed GPU-enabled Computation For this last experiment we have employed a distributed application with support for GPUs. The application, CUDA-MEME [45], is a highly efficient scalable motif discovery algorithm. Particularly, we have used *mCUDA-MEME*, a further extension of CUDA-MEME in terms of sensitivity and speed, which enables users to leverage a GPU per MPI process in order to accelerate motif finding. The application was configured to work on the input dataset `nrsf_2000.fasta`, from its test-cases, with a maximum size of 2,000,000 elements.

mCUDA-MEME is capable of using all the GPUs in a host provided that it is spawned one MPI process pre GPU. This feature limits the grade of parallelism to the number of installed GPUs in the target host. Apart from that, GPUs involved in the resolution of *mCUDA-MEME* are not fully computationally loaded. This experiment provides an insight into the use of GSaaS, which demonstrates its versatility when it comes to improve the performance of an application and the system utilization.

To increase the parallelism with the same hardware infrastructure, we share the GPUs in order to provide access to more cGPUs in two different ways. On the one hand, we configure a VM with 12 cGPUs (intranode distributed computation). On the other hand, we deploy up to 12 single-cGPU VMs (internode distributed computation). Both scenarios aim to increase the performance of *mCUDA-MEME* instances by spawning more MPI processes with access to a GPU. The VM for the intranode experiment consisted of 30 vCPUs and 30 GB of RAM, while the internode VMs were equipped with 2 vCPUs and 4 GB of RAM. Each cGPU was configured with 1 GB of memory, and GPGPUMS leveraged the *least load fit* policy.

Figure 5.14 shows the average time of 50 executions of *mCUDA-MEME* with an increasing number of CUDA-enabled processes running on the same VM (intranode) and on independent VMs (internode). For the sake of clarity, we have omitted the result of 1 cGPU (that is, 3,997.56 seconds) and beyond 12 processes, which did not experience further performance improvement. Increasing the number of CUDA-enabled MPI processes, by sharing the 4 underlying physical GPUs, reduces the execution time in both modes. However, offloading the processes in different VMs (internode scenario) provides better results (an improvement up to 1.5x). The intranode scenario processes run on the same compute host (where the VM is running), while the internode scenario processes run on different VMs, spread across the 3 compute hosts. Hence, compute hosts experience less load and the performance increases.

It is worth noting that the low utilization rate of the accelerators gives us an improvement opportunity. Figure 5.15 illustrates the average utilization rate of each physical GPU (obtained from the `nvidia-smi` command with a sampling rate of 1 second) during the execution of *mCUDA-MEME* for the configuration of different processes, when considering intranode (Figure 5.15a) and internode (Figure 5.15b) modes. Both charts depict a pattern where utilization rate increases a step

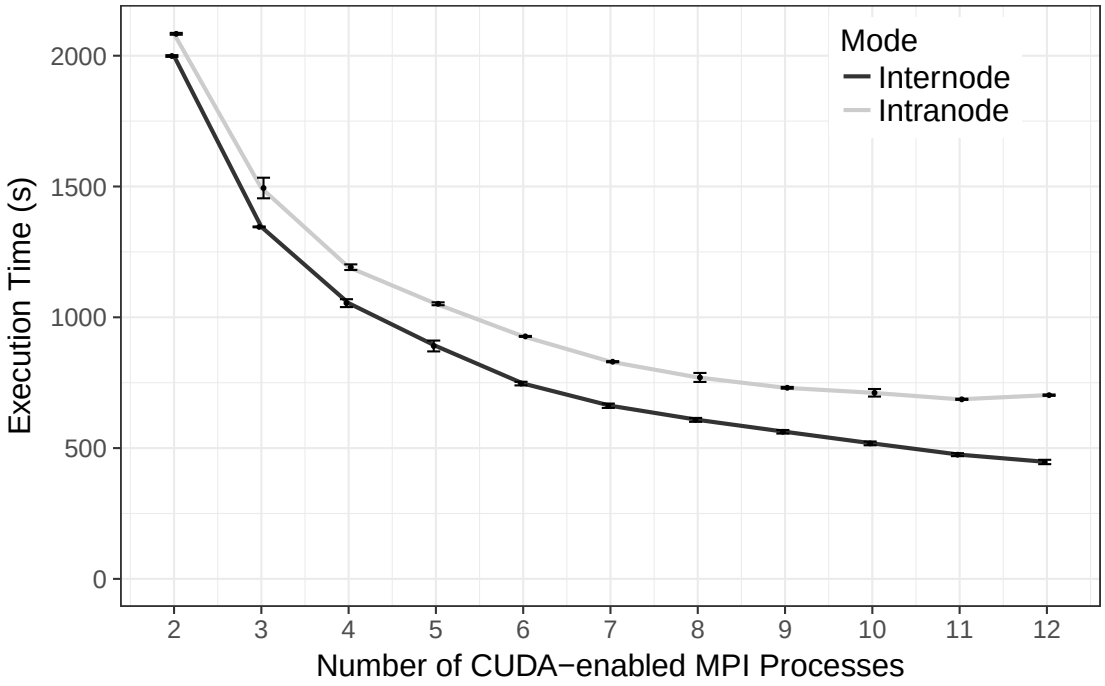


Figure 5.14: mCUDAmeme execution time with different number of CUDA-enabled MPI processes running on the same VM (intranode) and on different VMs (internode).

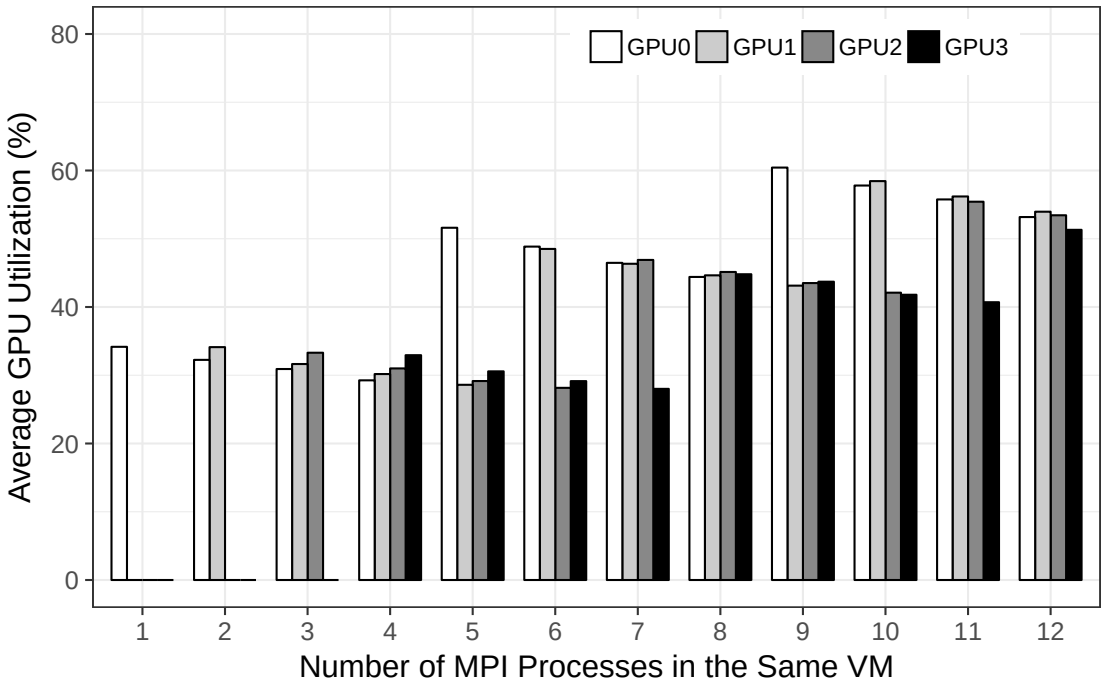
every 4 processes. Once the 4-process point is exceeded, where each process is using exclusively one of the 4 GPUs available, GPUs are oversubscribed by the subsequent processes. As it is shown in Figure 5.15b, processes running on different VMs can obtain higher GPU utilization, a 20% more than the intranode counterpart.

5.2.3 Conclusions

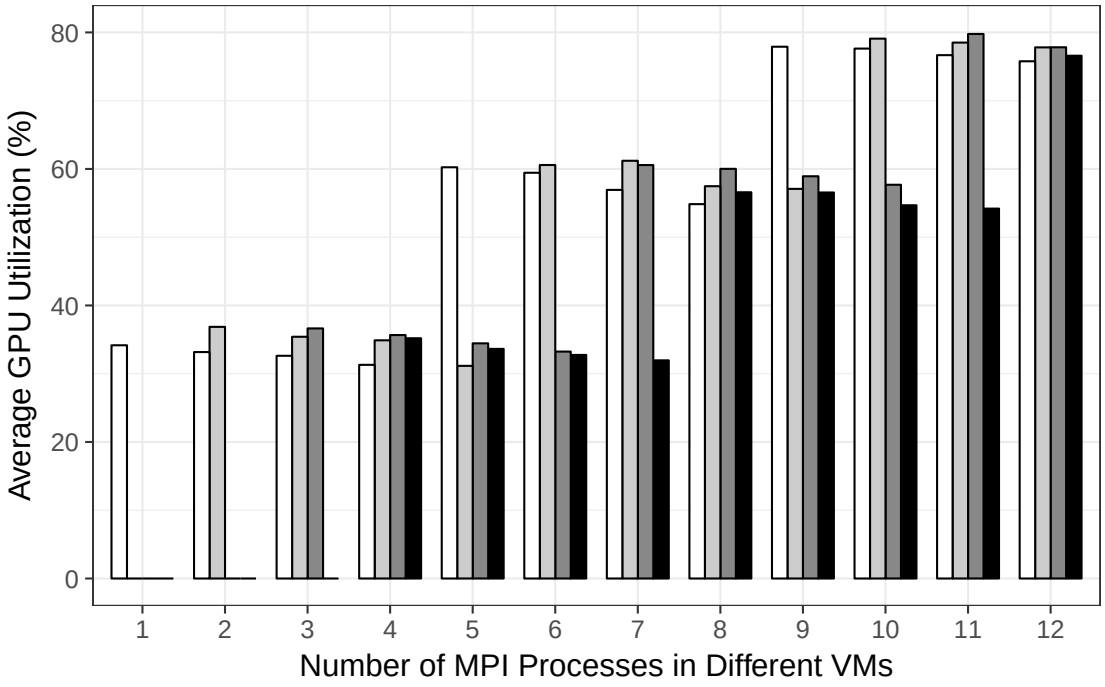
This work develops and evaluates GSaaS, a service to cloudify and schedule the access to physical GPUs from VMs, aimed to public cloud infrastructures.

The main benefits achieved by GSaaS are: adaptive scheduling of GPU resources, decoupling of the interface between the client and the rCUDA server, hiding the real location of the resources, preventing GPU unauthorized accesses, and detaching VM traffic from GPU traffic by using a dedicated network. Besides, the proposed solution automates the configuration of its distributed components.

A GSaaS prototype has been evaluated in an actual cloud deployment based on OpenStack. We have demonstrated its versatility in different scenarios where GSaaS can be leveraged to scale-up applications, facilitate the provision of accelerators or increase the utilization rate of the GPU. Deployment scalability experiments denote that our solution introduces low overhead, since the deployment time is only increased 0.2% when assigning 50 cGPUs to a VM. Performance experiments reveal the importance of VM-GPU tenant locality and GPU shareability in different scenarios. Our results show that the application performance is barely affected, and the proposed service can increase GPU utilization, showing up to a 20% improvement in a distributed scenario.



(a) Intranode Mode



(b) Internode Mode

Figure 5.15: GPU utilization rate for both distributed modes.

Part III

Dynamic Management of Resources

CHAPTER 6

DMR API: An OmpSs-like Malleability Solution

Traditionally, parallel applications running in high-performance computing (HPC) facilities keep allocated all the resources assigned at submission during the complete duration of their execution. This behavior can lead to the monopolization of large fractions of resources by only a few jobs, preventing others from being initiated. Job malleability can avoid the negative effects produced by a fixed job scheduling.

Malleable jobs are able to re-scale themselves at execution time by expansion or shrinkage. When malleable jobs are included in a workload, the result is an adaptive execution that can benefit from job reconfigurations in order to meet specific requirements given by the system administrator. Readjusting the workload to the cluster status on–the–fly requires a system-aware job scheduler, that for instance, considers information about resource utilization or the number of pending and running jobs. Taking into account the system status yields benefits not only to the HPC facility—which can experience an increase in the number of completed jobs per second (global throughput) and a better exploitation of its resources—but also to the end users, who can enjoy shorter completion (waiting plus execution) times for their jobs.

This part of the PhD dissertation is focused on MPI malleability, which can be understood as the capability of resizing jobs by reconfiguring the number of MPI ranks during the execution of a job. Specifically, in this chapter we present the dynamic management of resources application programming interface (DMR API): a malleability solution based on the offload semantics of OmpSs. For this purpose, a communication layer between Slurm and Nanos++ has been implemented. The sections of this chapter thoroughly describe the protocol of the communication layer and the design decisions for the development of the procedures that allow the malleability.

The main contributions in this chapter are:

- a malleability solution able to perform automatically the data transfers among processes and a demonstration of its benefits when malleable jobs are processed in a workload;
- an extensive evaluation of the API and its features;
- and the case study of the malleability adaptation of a bioinformatics application.

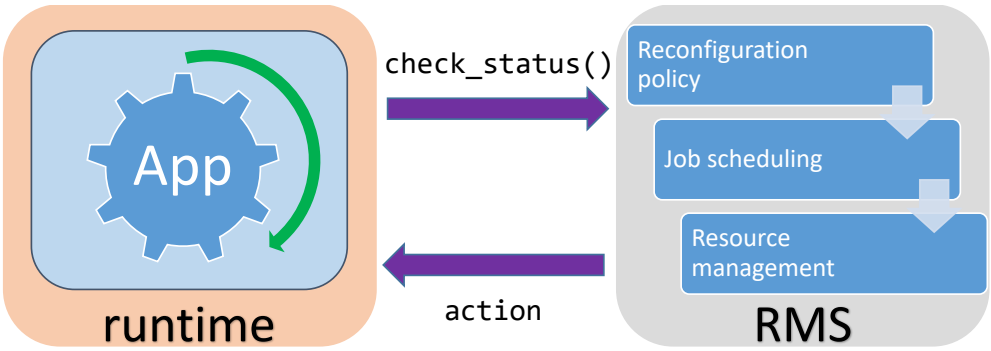


Figure 6.1: Communication protocol between the RMS and the runtime.

6.1 Methodology

Job malleability combines interactions from different entities. Specifically, the DMR API involves: i) user applications, which have to accept being resized at some points of their execution (malleable jobs); ii) a parallel distributed runtime responsible for re-scaling the jobs; iii) an RMS that makes adaptive decisions considering the cluster status and capable of reallocating job resources; and iv) a communication layer between them in order to perform the job reconfiguration process.

A malleable application contains a set of malleability points which initiate the communication between the RMS and the runtime. Commonly, these malleability points matches with the start or end of an iteration, even though, they can also indicate different computational stages. Hereafter, each iteration or stage is referred as step, and each step could define a malleability point.

The methodology of our malleability solution establishes the communication between Slurm (the RMS) and Nanos++ (the parallel distributed runtime) as it is depicted in Figure 6.1.

The RMS is aware of the resource utilization and the queue of pending jobs. When an application is in execution, it periodically contacts the RMS, through the runtime, communicating its rescaling willingness (to expand or shrink the current number of allocated nodes). The RMS inspects the global status of the system to decide whether to initiate any rescaling action, and communicates this decision to the runtime. If the framework determines that a rescale action is due, the RMS, the runtime, and the application will collaborate to continue the execution of the application scaled to a different number of MPI processes.

6.2 Slurm Reconfiguration Policy

We designed and developed a resource selection plugin responsible for reconfiguration decisions. This plugin realizes a node selection policy featuring three modes that accommodate three degrees of scheduling freedom:

Request an Action

Applications are allowed to "strongly suggest" a specific action. For instance, to expand the job, the user could set the "minimum" number of requested nodes to a value that is greater than the number of allocated nodes. However, Slurm will ultimately be responsible for granting the operation according to the overall system status.

Preferred Number of Nodes

One of the parameters that applications can convey to the RMS is their *preferred* number of nodes to execute a specific computational stage. If the desired size corresponds to the current size, the RMS will return "no action". Algorithm 2 depicts the steps taken by the Slurm policy when *preferred* is set. If there is no outstanding job in the queue, the expansion can be granted up to a specified "maximum" (lines 1-3). Otherwise, if the preference is different from the current allocation (line 4), the RMS will try to expand (lines 5-7) or shrink the job to the preferred number of nodes (lines 8-11).

Algorithm 2 Slurm reconfiguration policy when *preferred* is set

```
 1: if am I the only job in the queue? then
 2:     action ← expand.
 3:     processes ← jobMaxProcs.
 4: else
 5:     if can I expand to preferred? then
 6:         action ← expand.
 7:         processes ← max_procs_to(preferred).
 8:     else
 9:         if can I shrink to preferred? then
10:             action ← shrink.
11:             processes ← preferred.
12:         end if
13:     end if
14: end if
```

Wide Optimization

The cases not covered by the preceding methods are handled following Algorithm 3.

A job is expanded if there are sufficient available resources to fulfill the new requirement of nodes and either (1) there is no job pending for execution in the queue (lines 9-11), or (2) no pending job can be executed due to insufficient available resources (lines 6-8). By expanding the job, we can expect it to finish its execution earlier and release the associated resources.

A job is shrunk if there is any queued job that could be executed by performing this action (lines 1-3). If the job is going to be shrunk, the queued job that has triggered the shrinking event will be assigned the maximum priority in order to foster its execution (lines 4-5).

6.3 Nanos++ Runtime Extension

We implemented the necessary logic in Nanos++ to reconfigure jobs in tight cooperation with the RMS. In this section we discuss the extended API and the resizing mechanisms.

We designed the DMR API with two main functions: dmr_check_status and its asynchronous version `dmr_icheck_status`. These routines instruct the runtime (Nanos++) to communicate with the RMS (Slurm) in order to determine the resizing action to perform: "expand", "shrink", or "no action".

Figure 6.2 depicts the processes performed in each computational iteration when using the synchronous version. At the beginning of the iteration, the reconfiguration mechanism is triggered.

Algorithm 3 Slurm reconfiguration policy wide optimization

if *are there pending jobs in the queue?* **then**
  if *can other job run with my resources?* **then**
    $action \leftarrow$ `shrink`.
    $processes \leftarrow min\_procs\_run(targetJobId)$.
    $set\_max\_priority(targetJobId)$.
  else
    $action \leftarrow$ `expand`.
    $processes \leftarrow max\_procs\_to(jobMaxProcs)$.
  end if
else
  $action \leftarrow$ `expand`.
  $processes \leftarrow jobMaxProcs$.
end if

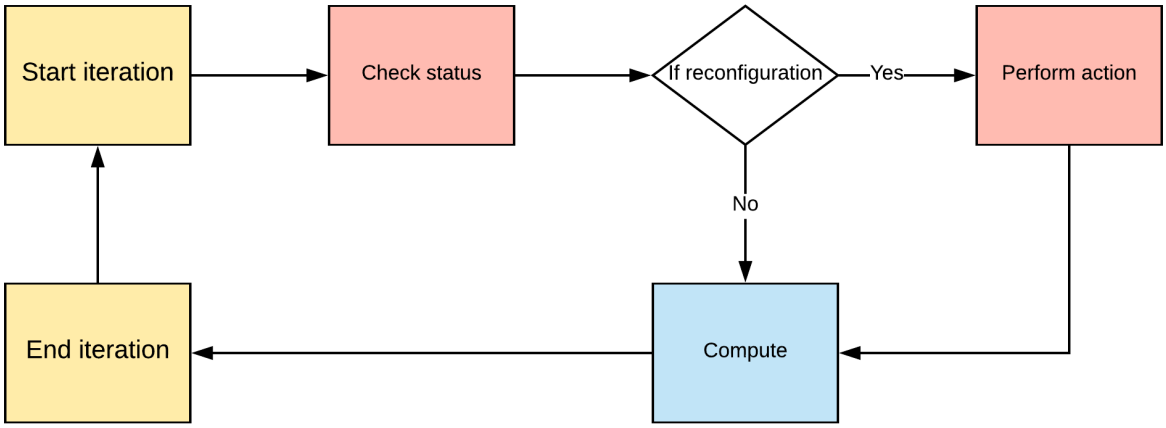


Figure 6.2: Execution flow of the synchronous reconfiguration method.

If it resolves to resize the job, the action will be performed and then the application will continue with the computation of that iteration.

The asynchronous counterpart schedules the action for the next execution step, at the same time that the current step is executed. Hence, by skipping the action scheduling stage, the communication overhead in that step is avoided. Figure 6.3 shows this behavior, checking at the begin of the iteration if a previous reconfiguration has been scheduled. If it is not the case, concurrently the runtime will check the cluster status while the application continues with its computation stage in the iteration. The decision of the reconfiguration will be applied in the next iteration.

Thus, in case an action is to be performed, these functions spawn the new set of processes and return an opaque handler. This API is exposed by the runtime and it is intended to be used by applications.

These functions present the following arguments:

- IN:
  - *min*: Lower limit of a reconfiguration.
  - *max*: Upper limit of a reconfiguration.
  - *factor*: Geometric distribution factor.

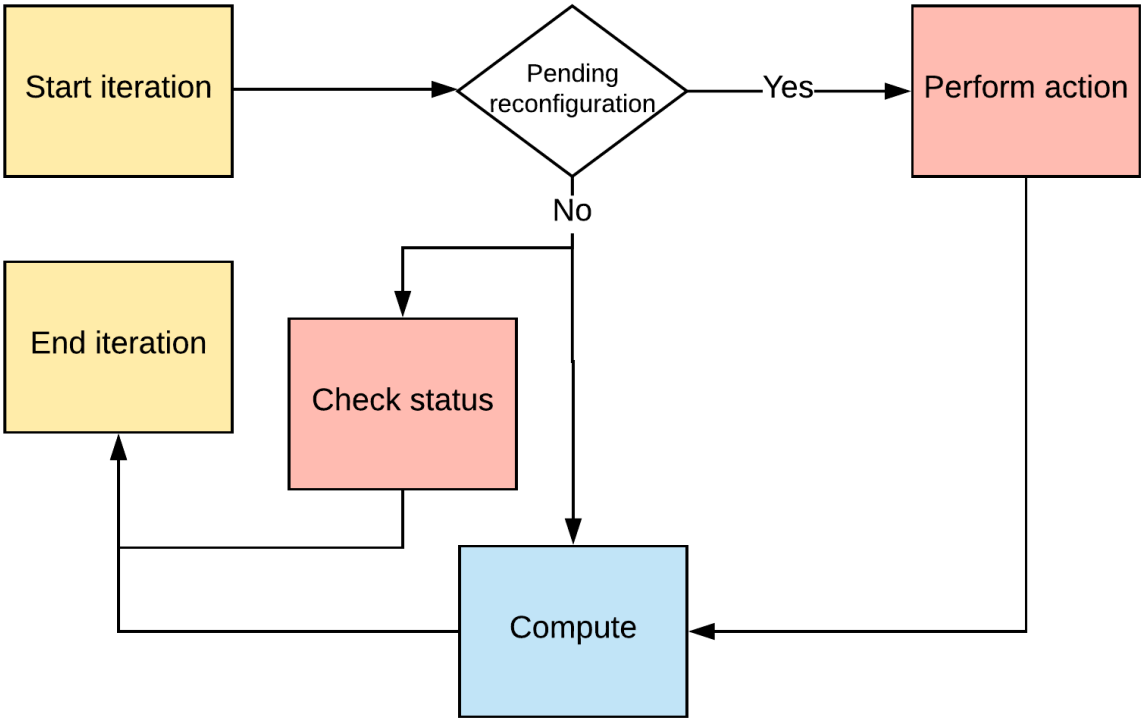


Figure 6.3: Execution flow of the asynchronous reconfiguration method.

  - *pref*: Preferred number of processes for running the application.

- OUT:
  - *nProcs*: Number of processes after the reconfiguration.
  - *handler*: MPI communicator with the new processes.

- RETURN:
  - *action*: Identifier of the scheduled action (0: None, 1: Expand and 2: Shrink).

An additional mechanism implemented to reach a fair balance between performance and throughput is the "checking inhibitor". This introduces a timeout during which the calls to the DMR API are ignored. This knob is mainly intended to be leveraged in iterative applications with short iteration intervals. The inhibition period can be tuned by means of an environment variable (`NANOX_SCHED_PERIOD`).

The runtime reconfigures a job leveraging the Slurm API resizing mechanisms described in Section 2.1.2. For instance:

Expand A new *resizer* job (RJ) is first submitted requesting the difference between the current and total amount of desired nodes. This enables the original nodes to be reused. There is a dependency relation between the RJ and the original job (OJ). In order to follow better the RMS decisions, RJ is set to the maximum priority, facilitating its execution.

The runtime waits until RJ changes from "pending" to "running" status. If the waiting time reaches a threshold, RJ is canceled and the action is aborted. This situation may occur if the RMS

assigns the available resources to a different job during the scheduling action. This is more likely to occur in the asynchronous mode because an action then can experience some delay during which the status of the queue may change. Once OJ is reallocated, the updated list of nodes is gathered and used in a call to `MPI_Comm_spawn` in order to create a new set of processes.

Shrink The shrinking mechanism is slightly more complex than its expansion counterpart because Slurm will have to kill all processes executing in the released nodes. To prevent premature process termination, we need a synchronized workflow to guide the job shrinking. Hence, the RMS sets a *management node* in charge of receiving an acknowledgment from all other processes. These acknowledgements (ACKs) will signal that they finished their offloading tasks and the node is ready to be released.

After a scheduling is complete, the DMR API call returns the expand-shrink action to be performed and the resulting number of nodes. The application is responsible for stating the appropriate data dependencies and triggering the tasks offloading to the new set of processes, as we will see in the next section.

6.4 Programming Model

In this section we review the programming model offered by the DMR API to address job malleability coordinated by the RMS. The programmability of our solution benefits from relying on the OmpSs offload semantics versus directly using MPI.

To showcase the benefits of the OmpSs offload semantics, we make use again of the specific case of migration studied in Section 3.2.3. This analysis allows us to focus on the fundamental differences among programming models.

Listing 6.1 presents how malleability is adopted with the DMR API. There, we can see that the *main* function is unaltered; it only initializes the data and invokes the *compute* function (lines 1-6). Once the iterations start (line 8), a call to the API is done (line 9). This call returns the reconfiguration action scheduled by the RMS, in this case Slurm, and the MPI communicator where the new processes are spawned. The reconfiguration is conducted by the #pragma in line 11, with which the user defines the data dependencies and the communication pattern among processes in different communicators. With this directive, we are explicitly indicating that *data* has an input dependency for the new processes in the *handle* MPI communicator. Since this is a migration, the communication is *rank-to-rank*. Furthermore, in order to resume the processes execution at this point of the code instead of starting from scratch when they are spawned, after the #pragma directive the user indicates the resuming function for the execution (line 12).

6.4.1 A Practical Example

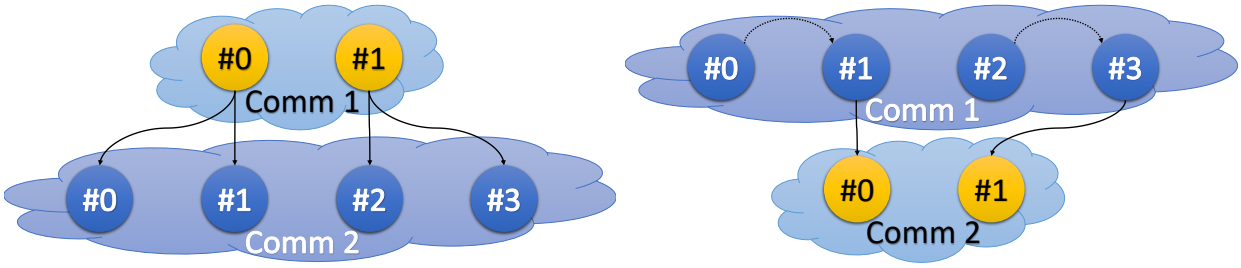
The excerpt in Listing 6.2 is derived from that on Listing 6.1 and it is aimed to discuss the malleability procedure. For the sake of clarity, here we only describe the procedure of geometric redistributions, as it is illustrated in Figure 6.4.

The function `compute` presents a malleability point in line 3, where the reconfiguration process is triggered. Depending on the action, the user has to perform a different set of operations:

- If *none* is returned (line 4), the computation will continue as originally expected.
- When expanding (line 7), each original process has to convey the data to other `factor` (line 8) processes in the new communicator. For this reason, the processes calculate the data partition

```
1  void main(int argc, char **argv) {
2    MPI_Init(&argc, &argv);
3    step = 0;
4    /* Initialization */
5    compute(data, dataSize, step);
6  }
7  void compute(double *data, int dataSize, int step) {
8    for (t = step; t < TIMESTEPS; t++) {
9      action =  dmr_check_status(&newNProcs, &handle);
10     if (action == MIGRATION) {
11       #pragma omp task in(data) onto(handle, myRank)
12       compute(data, dataSize, t);
13     } else
14       /* Computation */
15   }
16 }
```

Listing 6.1: Pseudo-code of job reconfiguration using the DMR API.



(a) Expand. (b) Shrink.

Figure 6.4: Data transfers among processes in the communicators when expanding and shrinking.

(lines 11-12) that has to be sent to each new process (line 10), as detailed in example of Figure 6.4a. The data transfers are performed by the runtime according to the information included in the task offloading directive (line 13)

- The shrinking (line 16) involves preliminary explicit data movement. The processes in the original communicator are grouped into "senders" and "receivers". This initial data movement is illustrated in the example in Figure 6.4b. There, data is gathered in "receivers" (lines 17-28) in order to posteriorly send it to the new communicator processes (line 31).

6.5 Experimental Evaluation

This section is divided in three parts. The first introduces the experimental environment, not only the HPC facility, but also the workloads and the applications converted into malleable. The second evaluates and discusses the utility of all the implemented features of DMR API through workloads of synthetic jobs. Finally, the last part describes and analyzes a realistic use case for our framework, demonstrating the benefits of deploying adaptive workloads in a production cluster.

```
void compute(double *data, int dataSize, int step) {
  for (t = step; t < TIMESTEPS; t++) {
    action = dmr_check_status(&newNProcs, &handle);
    if (action == NONE)
      /* Computation */
    else {
      if (action == EXPAND) {
        factor = newNProcs / nProcs;
        for (i = 0; i < factor; i++) {
          dst = myRank * factor + i;
          iniPart = (dataSize / factor) * i;
          finPart = (dataSize / factor) * (i + 1);
          #pragma omp task in(data[iniPart:finPart]) onto(handle,dst)
          compute(data + iniPart, dataSize / factor, t);
        }  // End for
      } else if (action == SHRINK) {
        factor = nProcs / newNProcs;
        sender = (myRank % factor) < (factor - 1);
        if (sender) {
          dst = factor * (myRank / factor + 1) - 1;
          MPI_Isend(data, dataSize, dst, myComm);
        } else {  // Receiver
          for (i = 1; i <= factor; i++) {
            src = myRank - factor + i;
            MPI_Irecv(&allData, dataSize, src, myComm);
          }  // End for
        }  // End if (sender)
        MPI_Waitall();
        if (!sender) {
          dst = myRank / factor;
          #pragma omp task in(allData) onto(handle,dst)
          compute(allData, allDataSize, t);
        }  // End if (!sender)
      }  // End if (action == ...)
    }  // End if (action)
  }  // End for
}  // End compute()
```

Listing 6.2: Full example of a malleable application using the DMR API.

6.5.1 Experimental Setup

HPC Infrastructure

Our evaluation was performed on the Marenostrum III Supercomputer at *Barcelona Supercomputing Center*. Each compute node in this facility is equipped with two 8-core Intel Xeon E5-2670 processors running at 2.6 GHz with 128 GB of RAM. The nodes are connected via an InfiniBand Mellanox FDR10 network.

For the software stack we used MPICH 3.2, OmpSs 15.06, and Slurm 15.08. Slurm was configured with the following plug-ins:

- Job scheduling: `sched/backfill` with a 10-second interval time among scheduling attempts. Since the backfilling policy allows overtaking, queued jobs are checked every 10 seconds.
- Job priority: `priority/multifactor` without wall time duration of jobs. Jobs are not submitted with they expected completion time, so the policy ignore the wall time when calculating priorities.
- Resource selection: `select/linear`. This policy understands the "node" as the minimum unit for assigning resources.

Malleable Applications

For our experimentation we used one synthetic and three real applications. Down below, their adaptation to malleability is described.

Flexible Sleep flexible sleep (FS) is an iterative synthetic application that performs a *sleep* in each step (iteration). Synthetic applications like this one or others that model real programs have been used in evaluations of several malleability solutions (i.e.: [61, 41]).

The time of the step depends on the number of processes deployed in that iteration, so if the job is resized, the sleep time is modified assuming perfect linear scalability. Apart from the *sleep* that simulates the computation time, the application also manages an array of doubles, distributed among the ranks. This array is presented to OmpSs as a dependency in order to perform the appropriate data redistribution among ranks in case of a job reconfiguration.

Conjugate Gradient The CG method is an iterative algorithm for the numerical solution of sparse systems of linear equations that produces a solution after a finite number of iterations. For our experimentation the method will perform a specific number of iterations in order to have control of the execution time.

The data of the application propagated in each iteration step of CG is constrained to a matrix flat-stored and four vectors. This version is implemented using OpenMP+MPI, and each MPI process works on a block of rows of the matrix and the corresponding elements from the vectors. The local matrix-vector products are parallelized with OpenMP.

We have applied our OmpSs-based extensions in order to ease the creation of new processes while maintaining the data dependencies after the resizing procedure. The five data structures in CG conform the data dependencies among iterations in the OmpSs programming model, and they are redistributed when a rescaling is necessary.

During a resize, the data in the matrix and vectors must be redistributed according to the new number of MPI processes.

Jacobi The Jacobi method is an iterative and embarrassingly-parallel algorithm for the solution of a system of linear equations.

Our OpenMP+MPI version of this solver is based on the implementation presented in [36]. The program layout is similar to the CG implementation. In this application, we also have a flat matrix, but only two vectors. These three structures conform the data-dependencies for OmpSs and they are all distributed among the processes.

N-body The N-body problem[1] simulates the individual motions of a group of objects interacting with each other by means of a given force.

We have used an OmpSs+MPI version of this simulator where each process stores a subset of particles, while the intranode parallelism is exploited by OmpSs.

The amount of work of N-body per iteration is considerably larger than that present in the remaining two full applications described in this section. Apart from computing the position and forces of its own particles, each process exchanges its local subset of particles with the other processes. At the end of the iteration, all the processes have worked with the whole set of particles.

The data-dependency in this particular case is dictated by an array of particles with information about position, velocity, mass and weight. This array is split or merged when an scale-up or down is respectively scheduled.

Workload Configuration

The workloads were generated using the statistical model proposed by Feitelson [14], which characterizes rigid jobs based on observations from logs of actual cluster workloads. These include the distribution of job sizes in terms of number of processors, the correlation of runtime with parallelism, and the number of repeated runs. Among others, we found several customizable parameters to be especially relevant:

- `Jobs`: Number of jobs to be launched.
- `Arrival`: Average job inter-arrival time to the queue using a Poisson distribution with a factor of 10.

Moreover, for the workloads composed of synthetic jobs, we also leveraged the following model parameters:

- `Job size`: Number of nodes determined by a complex discrete distribution.
- `Runtime`: Fixed following a hyper-exponential distribution based on the `job size`.

For every job executed in these experiments, the resizing factor was set to 2.

6.5.2 Preliminary Study

Here we present an in-depth analysis of the framework's features using synthetic malleable jobs. In order to test thoroughly the features of our solution, we performed 4 different experiments comprising: synchronous and asynchronous scheduling, heterogeneous workload, and micro-steps. Finally, we evaluate the performance of the DMR API in terms of scheduling and reconfiguring time.

[1] `https://en.wikipedia.org/wiki/N-body_problem`

Table 6.1: Configuration parameters for the applications

Application	Iterations	Number of Processes			Scheduling period
		Minimum	Maximum	Preferred	
FS	25	1	20	-	-
CG	10000	2	32	8	15 seconds
Jacobi	10000	2	32	8	15 seconds
N-body	25	1	16	1	-

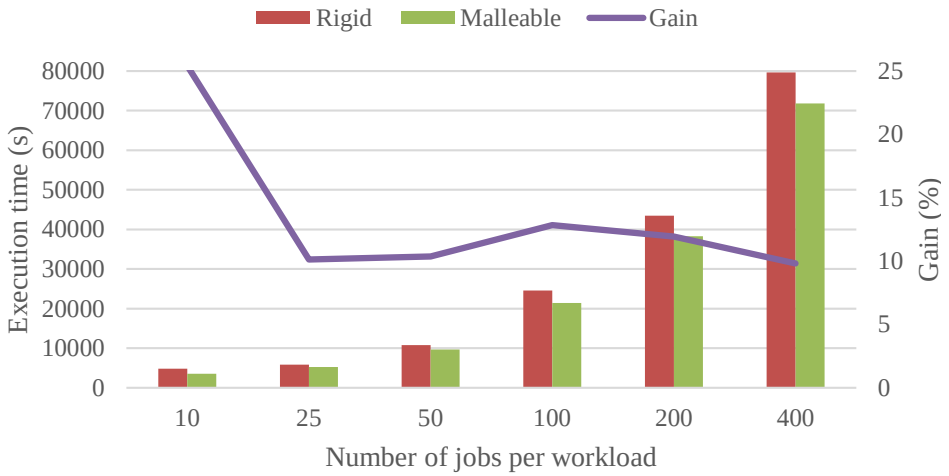


Figure 6.5: Comparison of different workload sizes composed of rigid and malleable jobs with synchronous scheduling.

For this study we only used the FS application; we generated several workloads of different size (number of `jobs`), assigning up to 20 nodes to each job (the number of available nodes in this experiment) with the parameter `job size`; the maximum `runtime` was set to 60 seconds for each step; and the average `arrival` time was 10 seconds. Table 6.1 details the rest of the FS parameters for the reconfigurations. Besides, the table also contains the information of the remaining applications used in the next section. The array was determined to have 1 GB of data transferred in each iteration. Furthermore, by not providing a preferred reconfiguration value (see Section 6.3) the RMS has absolutely freedom to reallocate resources.

6.5.2.1 Synchronous Reconfiguration Scheduling

For the first test, we launched workloads of different sizes in terms of number of jobs. Figure 6.5 depicts the execution time for this experiment. Each workload features both a rigid as well as a malleable version (see Section 3.2). The line "Gain" in that chart indicates the reduction of the execution time (in %) attained by the malleable workload with respect to the rigid workload.

Except for the 10-job workload, we can appreciate a gain in the interval 10-15% for the execution time, although the benefit decreases as the workload grows. Nonetheless, this occurs because we are evaluating a finite workload. Under these conditions, the scheduler is able to backfill jobs and fulfill resources, but malleability cannot bring a higher resource utilization. In a more realistic scenario, involving a much larger workload, the throughput would always be higher for the malleable

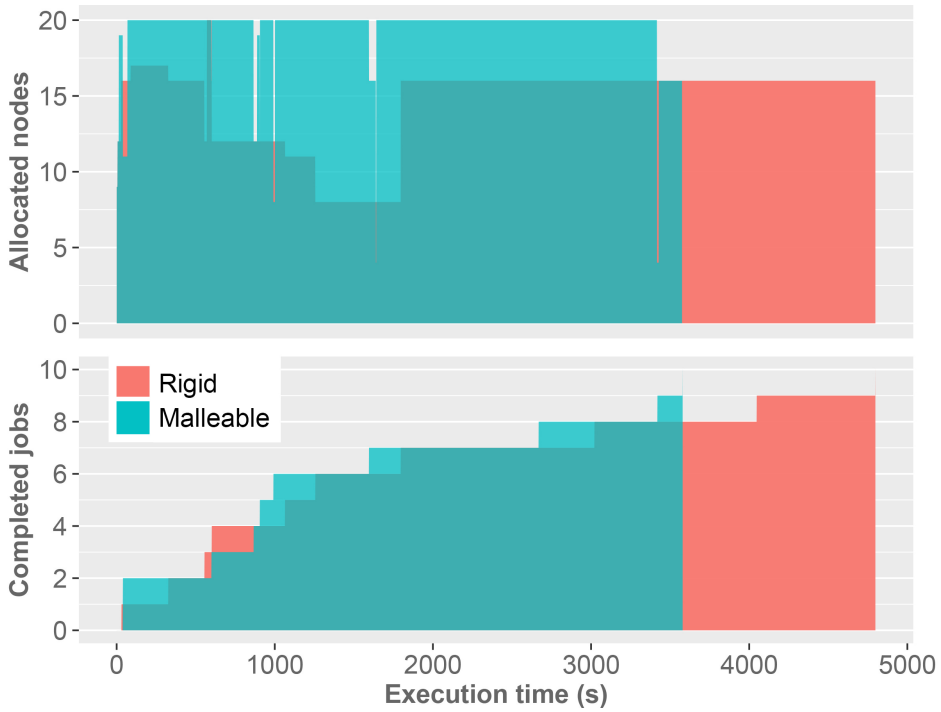


Figure 6.6: Evolution in time for the 10-job workload with synchronous scheduling.

workloads. For instance, bottom plots in figures 6.6 and 6.7 show that the productivity attained by the malleable workload is always higher than that offered by the rigid workload.

Figure 6.6 reports an almost-full allocation of resources during the malleable execution, exposing that the remarkable gain is due to the increment in resource usage.

In contrast, Figure 6.7 depicts the behavior of a 25-job workload, for which we observe a lower gain. The vertical black line points the instant of the workload execution when only 2 jobs remain in the queue, concretely, the penultimate job (PJ)that allocates 16 nodes and the last job (LJ) using 4 nodes. When PJ finishes (in the next step of the completed jobs chart), 16 nodes are released, but until the next check, LJ cannot be expanded, that is why the resource allocation drops to 4. At that point, the scheduler decides to expand the job to its maximum, in this experiment 16 nodes. At the end of the timeline, no more jobs can use the spare resources. This is the same situation that appears in the rigid workload. The consequence is that there is no further improvement, as the malleable policy had already obtained the gain from the first reallocation of resources.

6.5.2.2 Asynchronous Reconfiguration Scheduling

In this test we evaluated the asynchronous scheduling version. Again, we compare a fix workload and its malleable counterpart, but now the decision is made asynchronously. Here, we remind that the asynchronous scheduling takes a decision in a specific step but the action takes place in the next step. In the meantime, the status of the system may change. Therefore, the conditions found when the action is applied in the next step might not be the same that were present when the RMS decided the future action. In this situation, when there still are resources to perform an action, enforcing an outdated action may result in an inefficient use of resources. If there are not enough resource to perform the action, the reconfiguration timeout will be activated, but having wasted all that time.

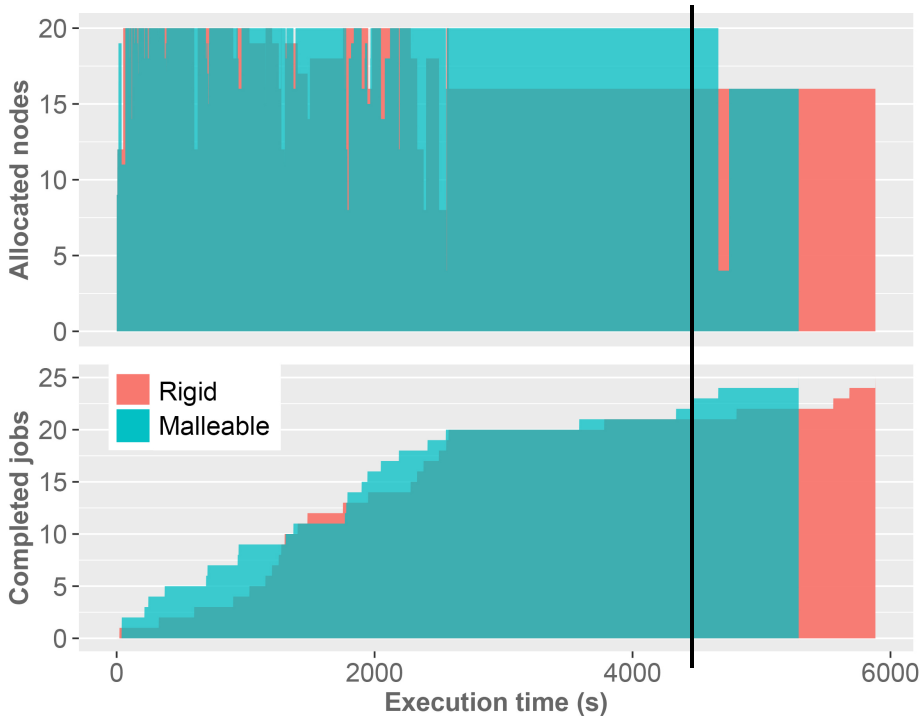


Figure 6.7: Evolution in time for the 25-job workload with synchronous scheduling.

Let us analyze the effect of adopting outdated decisions for the asynchronous scheduling in the 10-job workload. In this example the malleable workload performs worse than the rigid execution. We can explain the cause analyzing Figure 6.8. If we focus on the resource allocation evolution (top of Figure 6.8), two relevant gaps can be identified between seconds 2000-3000 and 3000-4000. At the beginning of the first gap in Figure 6.8, there are 3 jobs in execution that occupy a total of 19 nodes: J1 with 16, J2 with 2, and J3 with 1. At that instant, the RMS has decided to expand J3 to 2 nodes in the next step. When J1 finishes, the RMS schedules the expansion to 16 nodes of J2 (around second 2500). J3 needs more time to complete its iteration and performs its pending action of expanding to 2, having a total of 18 allocated nodes in the cluster. If J3 had checked the resources at that moment, it would have been expanded to 4 nodes, but the asynchronous scheduling was negotiated earlier, when the conditions were different. In addition to realizing the expansion, the scheduler decides that J3 will expand to 4 nodes in the next step.

The beginning of the second gap indicates the completion of J2. A few seconds later J3 expands to 4 nodes (instead of doing it to 16 if the scheduling had been synchronous).

Despite the lack of good results for small workloads, the larger workload completion times reveal a higher gain. In that type of situation the malleability overcomes the initial problem described above, since there are more jobs that trigger a reconfiguration and leverage the resources.

As we did in the synchronous benchmark (Figure 6.5), if we dismiss the small executions (10-to-50-job workloads), we can observe around a 6%-gain, with the improvement decreasing as we add more jobs to the workload.

In Table 6.2 we compare both modes, synchronous and asynchronous in more detail, analyzing their performance at cluster level and at job level. The most remarkable aspect here is that the synchronous scheduling occupies almost all the resources during the complete executions (the low standard deviation reveals that the mean value is barely unchanged for all the sizes). Moreover, the

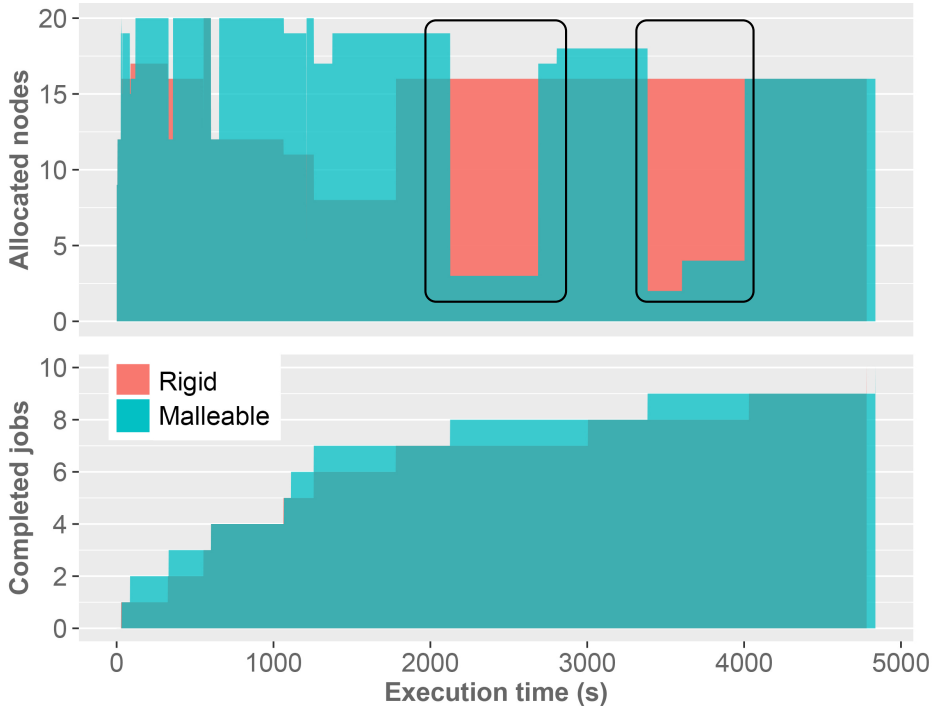


Figure 6.8: Asynchronous scheduling of the 10-job workload.

Figure 6.9: Comparison of different workload sizes composed of rigid and malleable jobs with asynchronous scheduling.

Table 6.2: Cluster and job measures of the 400-job workloads with synchronous scheduling.

Cluster Measures		Rigid	Synchronous	Asynchronous
Resources utilization	Avg. (%)	83.607	93.909	86.687
	Std. (%)	5.353	1.012	8.735
Per Job Measures		**Rigid**	**Synchronous**	**Asynchronous**
Waiting time gain	Avg. (%)	-	27.980	30.575
	Std. (%)	-	12.124	17.282
Execution time gain	Avg. (%)	-	-58.482	-97.294
	Std. (%)	-	26.731	34.378
Completion time gain	Avg. (%)	-	12.786	7.799
	Std. (%)	-	4.083	5.548

asynchronous mode still presents a higher utilization rate than the configuration without malleable jobs. However, the high standard deviation means that the utilization is not as regular as in the synchronous case. In fact, this result hides a low average utilization for small workloads (as we already reported in this subsection, the small workloads performed worst) compared with a high average for large workloads, similar to the synchronous scenario.

The last three rows offer information about timing measures: the wait-time of a job before entering execution, the execution time of the job, and the difference of time from the job submission to its finalization (completion). Malleability ("synchronous" and "asynchronous" columns) provides an important reduction of the wait-time in both modes for all the sizes. This is because the resource manager can shrink a job in execution in favor of a queued one.

With respect to the execution time, we experience a high degradation in the performance of each individual job. For the synchronous scheduling, the negative gain of around a 58% is closely related to the fact that the application scales linearly. Thus, halving the resources produces a proportional reduction of the performance. In the asynchronous scenario, the degradation is even worse due to the drawbacks previously discussed in this section. The high standard deviation means that not all the jobs perform so bad; in fact, the jobs in the small workloads are the most affected instances (as showed in Figure 6.9).

Finally, the global job time (completion time) places malleability in a good position, especially the synchronous scheduling that completes, on average, the jobs for a 12% earlier than the normal scenario.

This test reveals that so far, there is no need of using an asynchronous scheduling. Hence, the rest of the experiments will exclusively use the synchronous mode.

6.5.2.3 Heterogeneous Workloads

In this benchmark we mixed malleable and rigid jobs in the same workload in order to study their interaction. The workloads were composed of 100 jobs and the percentage of malleable jobs determined the probability of a job being malleable. We raised the ratio of malleable jobs between 0% and 100% in steps of 25%. Figure 6.10 depicts the execution times for these configurations. In general, we can appreciate that the execution time decreases as the ratio of malleable jobs grows. The results shown in this chart reveal a 10%-gain with only a 50%-rate of malleable jobs, and up to a 12%-improvement when all of them are malleable.

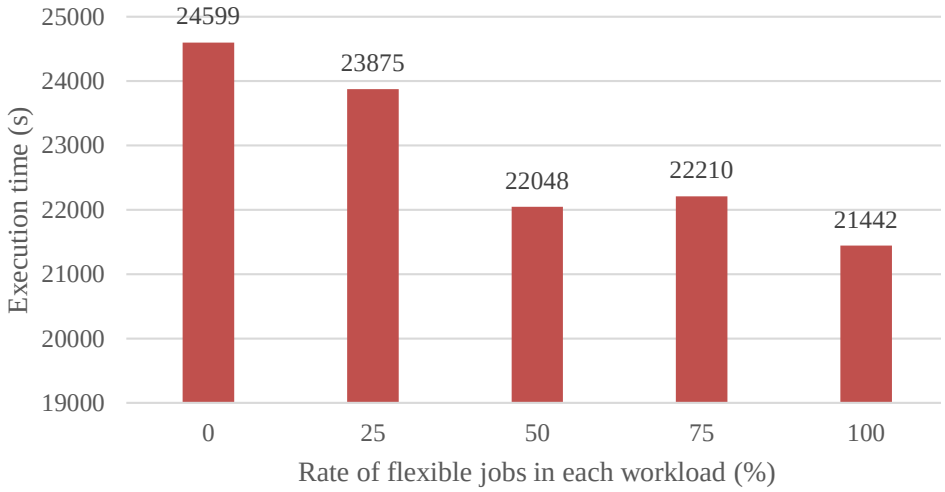


Figure 6.10: Execution times of 100-job workloads with different rates of malleable (showing the top of the chart; Y axis is not starting in 0) with synchronous scheduling.

6.5.2.4 Checking Period Inhibitor in Micro-step Applications

For this test we reduced the time step in the model to 2 seconds, corresponding to the minimum value for the experimented *period inhibitor* in order to investigate the importance of the overhead incurred by the scheduling process. Again, we generated workloads with a different number of jobs and executed them as both rigid and malleable workloads. For the malleable workloads, we enabled the *checking inhibitor* (see Section 6.3) to prevent that each iteration triggers a check. Figure 6.11 depicts the variation of malleability for a rigid execution. The group at the top ("malleable"), represents an execution without the *checking inhibitor* mechanism. The rest of the groups in the chart show the execution time when configuring the inhibitor period to: 2, 5, 10 and 20 seconds (from top to bottom), and its gain respect the rigid workload (percentage in the bar labels). Positive gain percentages reveal that by enabling periods of *checking inhibition* burst of communications between the runtime and the RMS can be avoided, reducing the overhead. In this particular example, setting a period of 5 seconds among actions scheduling not only offers better results than the rigid workload, but also outperforms a simple malleable workload.

6.5.2.5 Reconfiguration Scheduling Performance Evaluation

To conclude this section, we present a thorough analysis of the overhead of using our framework to enable malleability. For this purpose, we used the FS application configured to perform 2 steps and to transfer 1 GB of data during the reconfiguration. The idea is that each job executes an iteration, then it contacts with RMS and resumes the execution in the second step with the new configuration of processes.

Figure 6.12 shows the average time of 10 executions for each reconfiguration. On the left (a) we can see the times taken by the RMS to determine an action (scheduling time). From top to bottom, the first half of the chart depicts the expansions, while the second half shows the shrinks. The chart reveals a slight increment in the scheduling time when more nodes are involved in the process.

Figure 6.12(b) shows the time needed to perform the transfers between old and new processes. We can appreciate two interesting behaviors:

- The more processes involved in the reconfiguration, the shorter resize time. That is why

Figure 6.11: Execution time for the different inhibition periods (bars) and the gain respect the rigid workload (percentage on the right side of the bars) with synchronous scheduling.

the chunks of data are smaller and the time needed to transfer them concurrently is lower (compare the time between 1 to 2 and 64 to 32 processes).

- Shrinks involve much more synchronization among processes and the greater the difference in the number of processes is, the more time is needed to synchronize all of them.

We have also studied the overhead in a real workload execution. Table 6.3 reports the statistics collected for a 400-job workload. The table is divided in three parts and shows the amount of actions performed and their execution time during the workload execution in both synchronous and asynchronous scheduling.

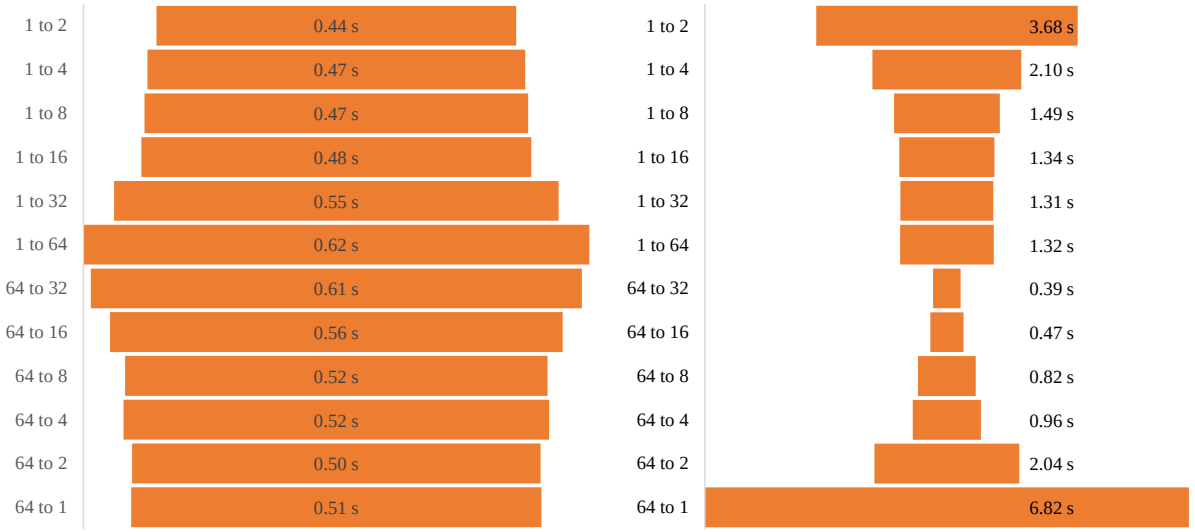
When no action is performed, the time to decide is virtually null (average time and standard deviation of "none action"). The time increases when the RMS performs an action because of the scheduling itself and the operations of reconfiguration performed by the runtime.

The first two rows of the second and the third part, provide information about the number of reconfigurations that are scheduled per workload and per job. We can see that the synchronous version schedules fewer reconfigurations and not all these jobs are expected to be resized. Moreover, since we are processing workloads with many queued jobs, running jobs are likely to be shrunk in favor of the pending.

The table also demonstrates the negative effect of a timeout during an expansion. The timeout mechanism is enabled when an action cannot be performed (probably because of a lack of resources) and this effect is shown in the asynchronous scheduling column with the "maximum", "average" and "standard deviation" values. In addition to the maximum time taken by the runtime to assert the expanding operation, these timeouts reveal a non-negligible dispersion in the duration values for the "expand" action. In fact, having such a high standard deviation turns the average time little representative.

6.5.3 Performance Analysis

We have conducted this evaluation generating workloads of different sizes, composed of jobs which instance 3 real applications (CG, Jacobi and N-body). These applications have been config-



(a) Scheduling time.

(b) Resize time.

Figure 6.12: Time needed to reconfigure from/to processes (y axis in both charts).

Table 6.3: Analysis of the actions taken by the DMR API in a 400-job workload.

		Synchronous	Asynchronous
No Action	Minimum Time (s)	0.0010	0.0003
	Maximum Time (s)	0.2078	0.1140
	Average Time (s)	0.0094	0.0137
	Standard Deviation (s)	0.0102	0.0112
Action Expand	Quantity	50	107
	Actions/Job	0.125	0.267
	Minimum Time (s)	0.367	0.366
	Maximum Time (s)	0.530	40.418
	Average Time (s)	0.423	8.820
	Standard Deviation (s)	0.146	12.688
Action Shrink	Quantity	194	303
	Actions/Job	0.485	0.757
	Minimum Time (s)	0.233	0.334
	Maximum Time (s)	0.541	0.555
	Average Time (s)	0.425	0.422
	Standard Deviation (s)	0.498	0.049

Table 6.4: Applications Configuration

Application	Input Data
CG	A square matrix and 4 arrays of 16,384 elements each one.
Jacobi	A square matrix and 2 arrays of 16,384 elements each one.
N-body	A simulation space of 3,276,800 particles.

ured as detailed in Table 6.4.

In order to configure the malleability parameters of the applications (lower limit, preferred configuration and upper limit), we performed a strong scalability test. For this purpose, all the applications were executed with one process and incrementally we doubled the number of processes in each step.

With the completion time of these executions we obtain a relative index of completion time decrease, referred as *gain difference* and calculated in Equation (6.1).

$$s_{current} = \frac{t_{previous} - t_{current}}{t_{min\_procs}} \times 100 \tag{6.1}$$

In the equation, $t_{current}$ is the completion time using the current number of processes, $t_{previous}$ is the completion time of the previous number of processes configuration, and t_{min_procs} is the completion time of the minimum number of processes configuration. For example, in order to calculate the gain difference for CG executed in 8 processes ($s(8)$), we subtract the completion time of the previous configuration ($t(4)$) minus its own time ($t(8)$). This is divided by the reference completion time of the minimum processes configuration ($t(1)$). The result is finally multiplied by 100.

Figure 6.13 depicts the gain difference for each configuration of the four applications. In order to determine the limits of malleability, we considered a 10% threshold (thick horizontal line in the chart). The *lower limit* is defined by the first configuration to exceed this threshold; the *preferred* value is given by the last configuration before dropping below 10%; and the *upper limit* is the configuration with the highest performance. Although our cluster was composed of 64 compute nodes, we restricted the jobs to request a maximum of 32 nodes, assuming that a job should not monopolize more than a half of the cluster. Values below zero have been omitted since they represent an increase in the application execution time, automatically dismissed.

These tests identify two parallel behaviors:

- High scalability: CG and Jacobi. In this case both applications have a similar behavior, with the highest speed-up attained for 32 processes. However, from 8 processes on, the difference gain among tests drops below 10%, so we consider 8 processes as a "sweet configuration spot" for these two applications.

- Constant performance: N-body. In contrast, this application reaches its maximum performance for 16 processes. However, in this case, the gain does not exceed 10% with respect to the sequential run, so a single process is considered as the "sweet spot".

From the perspective of computational cost, CG and Jacobi comprise "short" iterations that complete in less than 2 seconds, while N-body executes costly iterations, in the scale of minutes. For this reason, for CG and Jacobi we enabled the *scheduling period inhibitor* featured by the runtime, in order to reduce the amount of communications with the RMS.

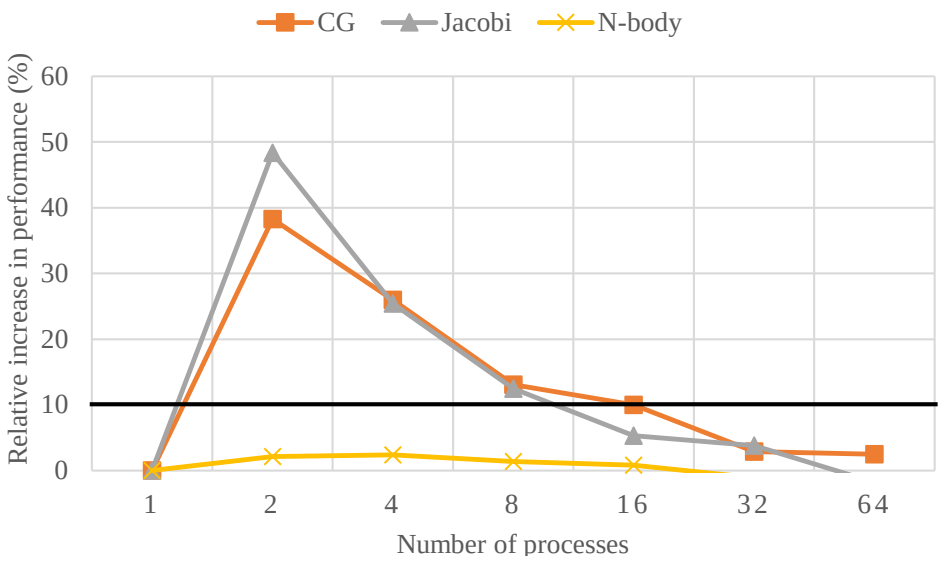


Figure 6.13: Gain difference of each application in Marenostrum III with synchronous scheduling. The thick horizontal line determines the limits of malleability with a threshold of 10%.

Table 6.1 displays the configuration employed in the experimentation. The job submission of each application is launched with its "maximum" value, reflecting the user-preferred scenario of a fast execution. Each workload is composed of a set of randomly-sorted jobs (with a fixed seed) which instantiate one of the three real applications (33% of jobs of each application class). Furthermore, the inter-arrival time among submissions is generated using the statistical model proposed by Feitelson [15], which characterizes rigid jobs based on observations from logs of actual cluster workloads.

6.5.4 Experimental Results

Figure 6.14 depicts the execution time of each workload size comparing both configuration options: rigid and malleable. The labels at the end of the "malleable" bars report the gain compared with the rigid version. Table 6.5 details the measures extracted from the executions. In the first column, we compare the average resource utilization for rigid and malleable workloads. This rate corresponds to the average time when a node has been allocated by a job compared to the workload completion time. These results indicate that the malleable workloads reduce the allocation of nodes around 30%, offering more possibilities for queued jobs and for a reduction of energy consumption.

The second column of Table 6.5 shows the average waiting time of the jobs for each workload. These times are illustrated in Figure 6.15, together with the gain rate for malleable workloads. The reduction around 60% makes the job waiting time a crucial measure to keep in mind from the perspective of throughput. In fact, this time is responsible for the reduction in the workload execution time.

The last two columns of Table 6.5 present two more aggregated measures of all the jobs in the workload: The first one is the average execution time; the second is this execution time plus the waiting time of the job, referred as completion time. The experiments show that jobs in the malleable workload are affected by the scale-down of their number of processes. However, this is compensated by the waiting time which benefits the completion time.

In order to understand the events during a workload execution, we have chosen the smallest workload to generate detailed charts and offer an in-depth analysis.

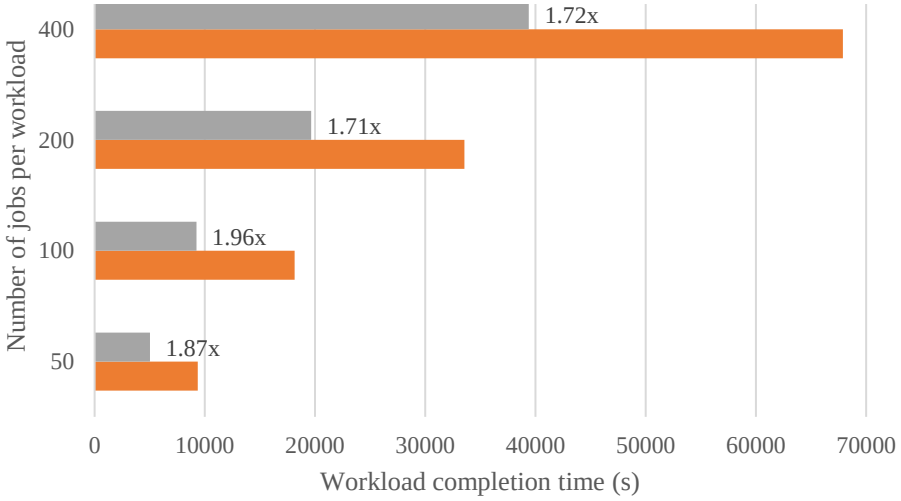


Figure 6.14: Workload execution times (bars) and gain of malleable workloads (bar labels) with synchronous scheduling.

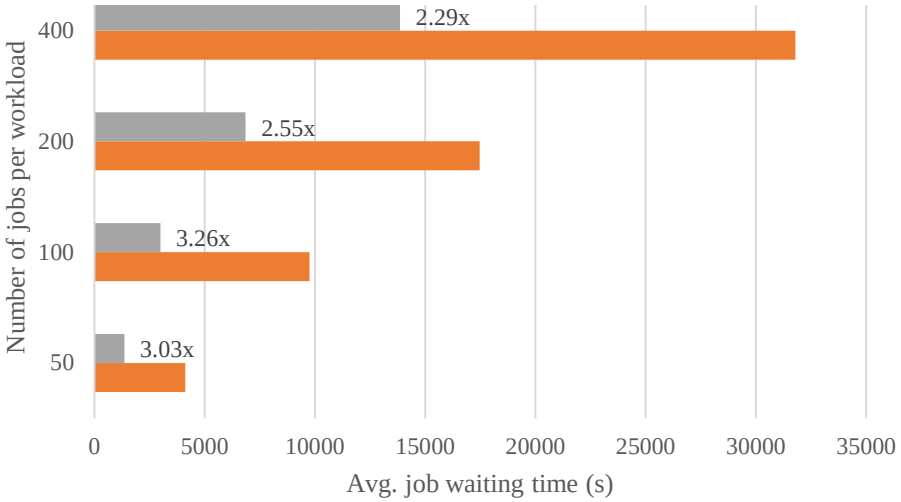


Figure 6.15: Average waiting time for all the jobs of each workload (bars) and the gain of malleable workloads (bar labels) with synchronous scheduling.

Table 6.5: Summary of the averaged measures from all the workloads with synchronous scheduling.

#Jobs	Version	Resource utilization rate (%)	Job waiting time (s)	Job execution time (s)	Job completion time (s)
50-job	Rigid	98.71	4115.02	620.26	4735.28
	Malleable	68.67	1359.92	900.30	2260.22
100-job	Rigid	97.39	9750.34	586.64	10336.98
	Malleable	71.91	2990.60	858.16	3848.76
200-job	Rigid	98.38	17466.20	520.58	17986.78
	Malleable	73.54	6856.80	825.88	7676.67
400-job	Rigid	98.38	31788.39	532.14	32320.53
	Malleable	73.54	13861.03	843.19	14704.22

The top and the bottom plots in Figure 6.16 represent the evolution in time of the allocated resources and the number of completed jobs. It also shows the number of running jobs for rigid and malleable workloads (blue and red lines respectively). The figures demonstrate that the malleable workload utilizes fewer resources; furthermore, there are more jobs running concurrently (top chart). For both configurations, jobs are launched with the "sweet spot" number of processes; the rigid jobs obviously do not vary the amount of assigned resources, while in the malleable configuration, they are scaled-down as soon as possible. This explains the reduction on the utilization of resources. For instance, in the second half of the malleable shape in Figure 6.16 (marked area), we find a repetitive pattern in which there are 5 jobs in execution which allocate 40 nodes. The next eligible job pending in the queue needs 32 nodes to start. Therefore, unless one of the running jobs finishes, the pending job will not start and the allocation rate will not be higher. When a job eventually finishes and releases 8 nodes, the scheduler initiates the job requesting 32 nodes. Now, the allocated nodes are 64 (the green peaks in the chart); however, since the job prefers 8 processes, it will be scaled-down.

At the beginning of the trace in the bottom of Figure 6.16, the throughput of the rigid workload is higher, but this occurs because the first jobs are completed earlier (they have been launched with the best-performance number of processes). Meanwhile, in the malleable workload, many jobs are initiated (blue line) and, as soon as they start to finish, the throughput experiences a boost.

Figure 6.17 depicts the execution and waiting time of each job grouped by application. The execution time (top row of charts) increases in the malleable workload for all the cases. As we explained before, we are shrinking jobs as soon as they are initiated to their preferred value. This implies a decrease in performance because of a reduction of resources. However, there is a job that leverages the benefits of an expansion. The last Jacobi job experiences a drop in its execution time hence it has been expanded thanks to completed jobs have released their resources.

The row of charts at the bottom compares the waiting time of all the jobs for their rigid and malleable versions. At the beginning there is no remarkable difference in the waiting time of both versions: however, the arrival of new jobs does not stop and resources remain allocated for the running jobs in the rigid workload. The RMS cannot provide the means for draining faster the queue. For this reason, queued jobs in the rigid workload, experience a dramatic delay in their initiation.

That difference in the initiation is crucial for the completion time of the job, as it is shown

Figure 6.16: Evolution in time for the 50-job workload. Blue and red lines represent the running jobs for rigid and malleable policies with synchronous scheduling.

in Figure 6.18. This figure represents the difference in execution, waiting and completion time for each job grouped by application. Again, the execution time remains below zero, what means that the difference is negative and the malleable workload performs slower. Nevertheless, this small drawback is highly compensated by the waiting time. As can be seen, completion time difference shows a heavy dependency on the waiting time, making it the main responsible for reducing the individual completion time, and in turn, the high throughput obtained in the experiments.

6.6 Case Study of a Scientific Application: HPG-aligner

After having proved the benefits of job reconfiguration in workloads, we are in the need of fostering the adoption of malleability solutions in scientific production applications. As exposed in Section 3.2.2, several efforts have been made in order to turn into malleable large applications such as LAMMPS[2] or LeanMD[3].. The applications introduced in that section follow an iterative pattern where all the processes execute the same operations over different data, which is conveyed among processes. In that set of applications, we also find one with a *master-worker* scheme, where independent *workers* load from disk the data to process. All in all, those applications are perfectly suitable for malleability because they present a structure with clearly identifiable malleability points. Unfortunately, not all scientific applications follow the guidelines of a process-level malleable job.

In this section we study the particular case of HPG (High-performance Genomics) Aligner, a distributed-memory non-iterative genomic sequencer featuring an irregular communication among processes. Through HPG-aligner, we exemplify the methodology to convert a non-malleable-oriented

[2] `http://lammps.sandia.gov`
[3] `http://charm.cs.illinois.edu/research/leanmd`

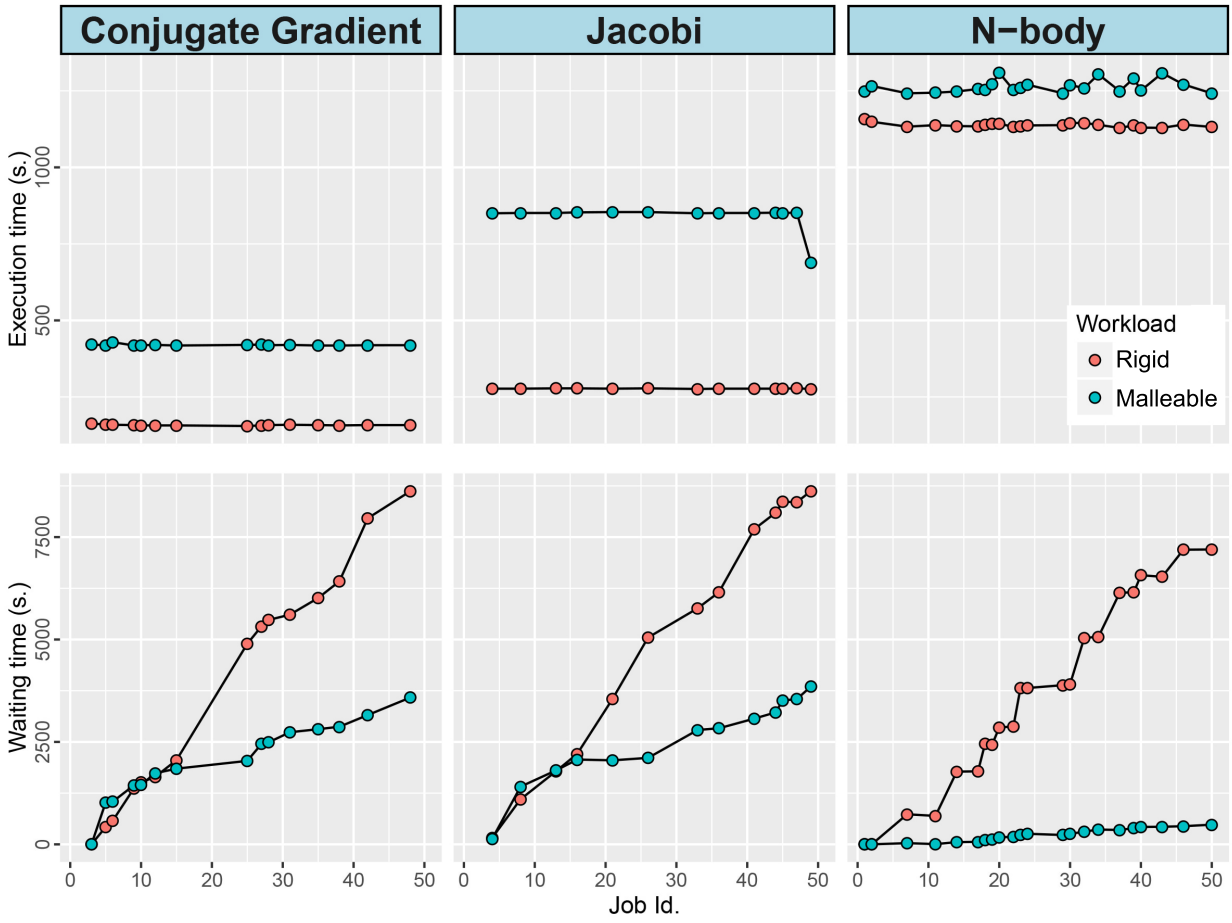


Figure 6.17: Execution (top) and waiting (bottom) times of each job grouped by application (columns) with synchronous scheduling.

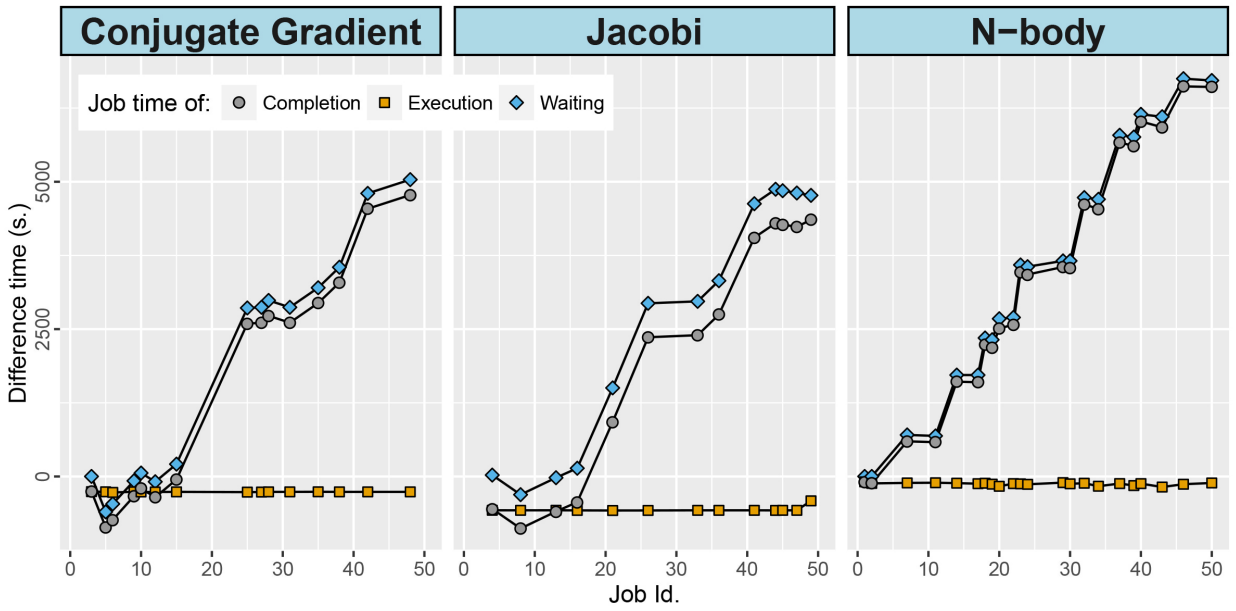


Figure 6.18: Time difference between the rigid and malleable version of each job: completion, execution and waiting time with synchronous scheduling.

application and, with an extensive experimental analysis, we prove its actual feasibility in production clusters.

6.6.1 Overview of HPG-aligner

HPG-aligner is a bioinformatics application for fast and accurate mapping of RNA sequences on a cluster of computers [50]. It employs several MPI processes as *workers*, and an additional MPI process as a *writer* (see Figure 6.19). Each *worker* operates over a particular part of the input file, which contains short RNA fragments (*reads*) produced by a Next Generation Sequencing (NGS) sequencer. At the beginning of the execution, each *worker* calculates the indexes of its part of the input file. Then, each *worker* performs its computation over its self assigned reads, and sends the alignments it obtains to the *writer*, whereas the *writer* stores the reported alignments to disk.

6.6.2 HPG-aligner Malleable Version

In this section we describe how the DMR API has been employed to convert HPG-aligner into a malleable application.

6.6.2.1 Adapting the HPG-aligner Workflow

The original implementation of HPG-aligner performs a static distribution of data among all the processes of the whole dataset (i.e., all the RNA *reads* obtained from an NGS sequencer). This strategy is not desirable for a malleable application, since it would complicate both the redistribution of the yet-to-be-done work among the new processes, and the determination of the synchronization point that all the processes should reach to evaluate whether a reconfiguration should be done. Thus, we have redesigned the HPG-aligner workflow in order to split the input workload into a user-defined number of chunks, each one will be dynamically assigned to a *worker* process.

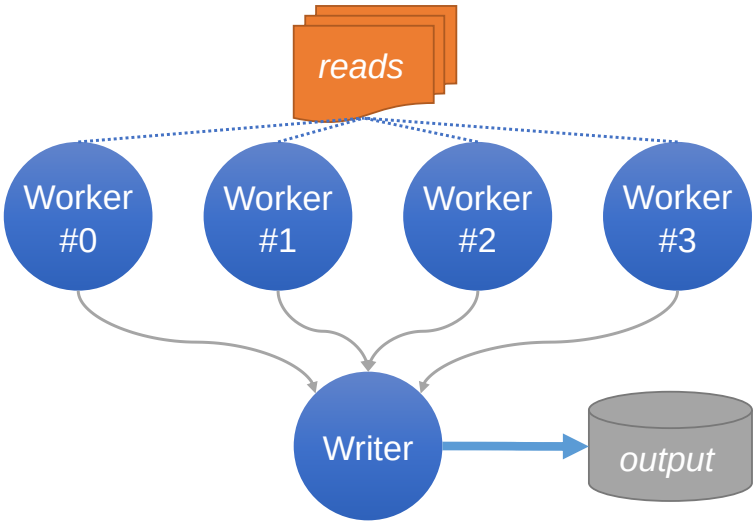


Figure 6.19: HPG-aligner original version workflow.

To allow HPG-aligner to dynamically distribute the dataset, a new process is created, a *manager*, which is in charge of the distribution of the input dataset chunks among the *worker* processes. In this new workflow, see Figure 6.20, when a *worker* has finished its previously assigned work, it asks the *manager* for more work. Then, the *manager* computes the index of the next dataset chunk to be processed, and assigns it to that *worker*.

Moreover, the HPG-aligner workflow has also been modified in order to allow the processes to periodically reach a malleability point, i.e., a synchronization point where the RMS can be asked if a reconfiguration should take place. This can be achieved by instructing all the HPG-aligner processes to reach the malleability point after a given number, n, of chunks have been processed. Figure 6.20 illustrates this "iterative" schema: all the HPG-aligner processes cooperate to compute n chunks of reads, and then, all of them reach the malleability point, where a reconfiguration could be triggered. These two stages, processing and reaching the malleability point, are repeated until all the chunks have been processed.

To implement this "iterative" schema and synchronize all the processes at a malleability point, the following strategy and communication schema have been employed. After dispatching n chunks, the *manager* stops distributing work. Instead, whenever a worker asks for more work, the *manager* signals him to proceed to the malleability point. The *worker*, in its side, after receiving this signal, propagates it to the *writer*. This communication schema, see Figure 6.21, ensures that all processes will eventually reach the malleability point after n chunks are processed.

The algorithms used to implement the described strategy and communication schema are depicted next using an MPI pseudo-code. In this pseudo-code, the first argument of the MPI send calls is the data to be transferred, and the second, its destination. Likewise, the first argument of the MPI receive calls is a container for the data to be received, and the second argument, the senders from whom that data is received.

Algorithm 4 shows the pseudo-code corresponding to a *manager* iteration. While there are chunks to be processed (line 1), the *manager* waits for a work petition from any *worker* (line 2), and in response to each of these, it provides the index of the next chunk to be processed (line 3). The malleability point is reached (line 7) when all the chunks of the current iteration have been assigned and a special signal (-1) is sent to all the *workers* (lines 4 to 6).

Algorithm 5 describes the pseudo-code of a *worker* iteration. First, the *worker* informs the

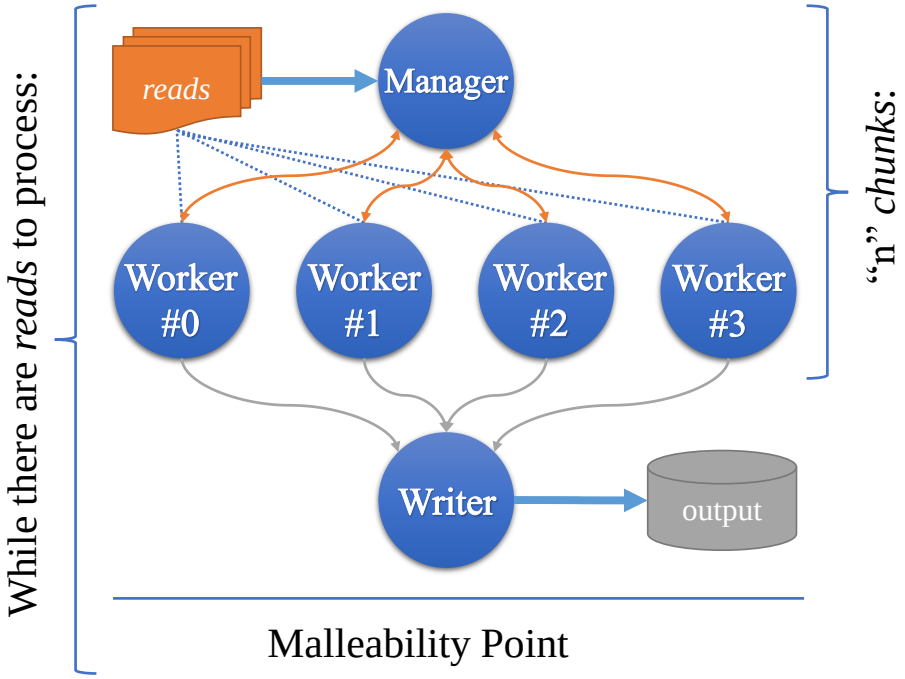


Figure 6.20: HPG-aligner malleable version workflow.

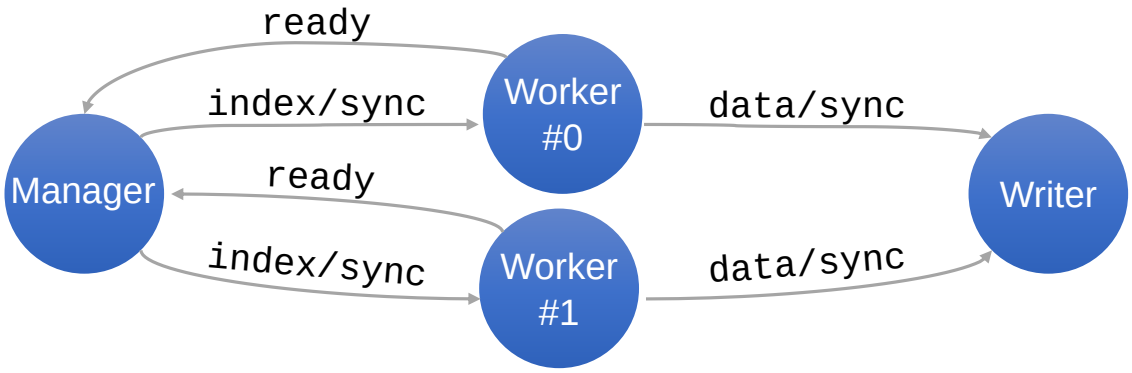


Figure 6.21: Communication schema of a DMR API reconfiguration.

Algorithm 4 Pseudo-code of a *manager* iteration

```
1: while chunks do
2:     MPI_Recv(&worker_id, MPI_ANY_SOURCE)
3:     MPI_Send(index, worker_id)
4: end while
5: for each worker do
6:     MPI_Irecv(&worker_id, worker)
7:     MPI_Isend(-1, worker)
8: end for
9: /* Malleability point */
```

manager that it is ready to process more work (line 1 and 6). Then, it waits until the *manager* sends back the index of the next chunk of reads assigned to it (line 2 and 7). If index -1 is received (line 3), it will propagate this signal to the *writer* (line 9), and proceed to the malleability point (line 10). Otherwise, it will process the assigned chunk and send its results to the *writer*.

Algorithm 5 Pseudo-code of a *worker* iteration

```
1: MPI_Send(worker, manager)
2: MPI_Recv(&index, manager)
3: while index ≠ −1 do
4:     compute(index, &data)
5:     MPI_Send(data, writer)
6:     MPI_Send(worker, manager)
7:     MPI_Recv(&index, manager)
8: end while
9: MPI_Send(-1, writer)
10: /* Malleability point */
```

Finally, Algorithm 6 describes the pseudo-code of the *writer* iteration. The *writer* listens for data from any worker (line 2). If an actual result is received (line 3), it will write it to disk (line 4). Otherwise, if the received data contains -1 (line 5), a counter is increased (line 6), and when the value of this counter reaches the number of *workers* (line 8), it will proceed to the malleability point (line 9).

Algorithm 6 Pseudo-code of the *writer* iteration

```
1: do
2:     MPI_Recv(&data, MPI_ANY_SOURCE)
3:     if data ≠ −1 then
4:         write_to_disk(data)
5:     else
6:         cnt+ = 1
7:     end if
8: while cnt ≠ n_workers
9: /* Malleability point */
```

6.6.2.2 HPG-aligner Data Redistribution Patterns

On the previous section a malleable workflow has been proposed for HPG-aligner so that it could periodically reach a malleability point where asking the RMS whether it should be reconfigured or not before continuing processing the input data. In this section, we will discuss how the data of the HPG-aligner processes could be redistributed in event of a reconfiguration.

As for the data involved in a redistribution, on HPG-aligner we can differentiate two types. On the one hand, all the HPG-aligner processes share some identical information, hereafter referred as *common data*. On the other hand, HPG-aligner *workers*, since they map each read to the reference genome, populate two data structures, either from scratch or from already-initialized values. These data structures, hereafter referred as *worker data*, are used by the HPG-aligner *workers* to improve their performance and their mapping quality. It should be noted that, although the workers do not require their particular worker data to be initially populated, HPG-aligner uses the aggregated data

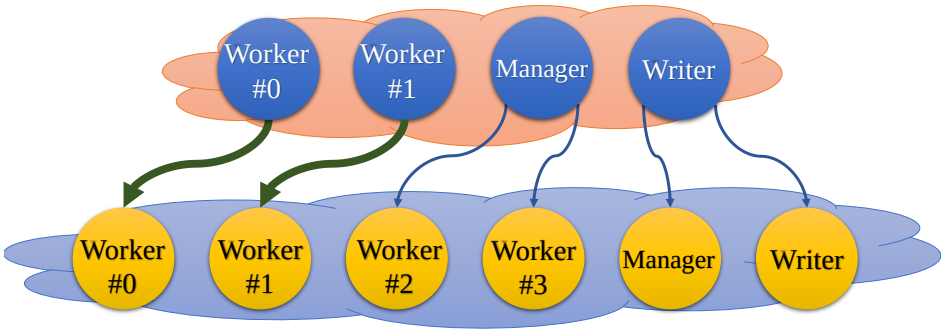


Figure 6.22: Data redistribution scheme of an expansion from 4 to 6 processes.

from all the workers in a later stage in order to be able to correctly align those reads that could not be previously aligned.

If a reconfiguration is performed, the *common data* and the *worker data* (referred as *cData* and *wData*, respectively) should be distributed from the current processes to the new processes. We have defined two different patterns depending on the reconfiguration type. If the reconfiguration is an expansion, i.e., the application will be executed on more processes after the reconfiguration, the following next strategy is performed:

- Each current w_i *worker* sends both its common and worker data to the new w'_i *worker*.
- The current *manager* sends its common data to half of the remaining new processes (which will become new *workers*).
- The current *writer* sends its common data to the other remaining new processes (two of which will become the new *manager* and the new *writer*).

Figure 6.22 illustrates this redistribution pattern when expanding from 4 to 6 processes. The initial *workers* send their common and worker data to their peer MPI ranks in the new communicator (wide arrows in the figure). At the same time, the initial *manager* and *writer* send their common data to the remaining newly spawned processes (narrow arrows in the figure).

Notice that each initial worker transfers its data to the homonym worker in the new communicator. As stated before, the data generated by each worker must be preserved, since it is required in later computational stages.

Otherwise, when the action is a shrinking, the application will be executed on less processes after the reconfiguration. In this case, the already populated worker data of those workers that will be removed should be preserved (since this information is paramount on a later stage of HPG-aligner). In order to ensure that the metadata will not be lost, we leverage an efficient parallel merge [52] implemented in HPG-aligner, which uses a minimum spanning tree pattern to merge all the workers data into the first worker (see Figure 6.23). Therefore, over the different options that could be employed to preserve and distribute this information, we have chosen to first call HPG-aligner parallel merge, and then follow the next strategy:

- The first worker, w_0, will send its common and worker data to the new first worker, w'_0 (the aggregated worker data will be preserved on this worker).
- The next workers will send their common data to the remaining new processes (the two last of them becoming the new *manager* and the new *writer*).

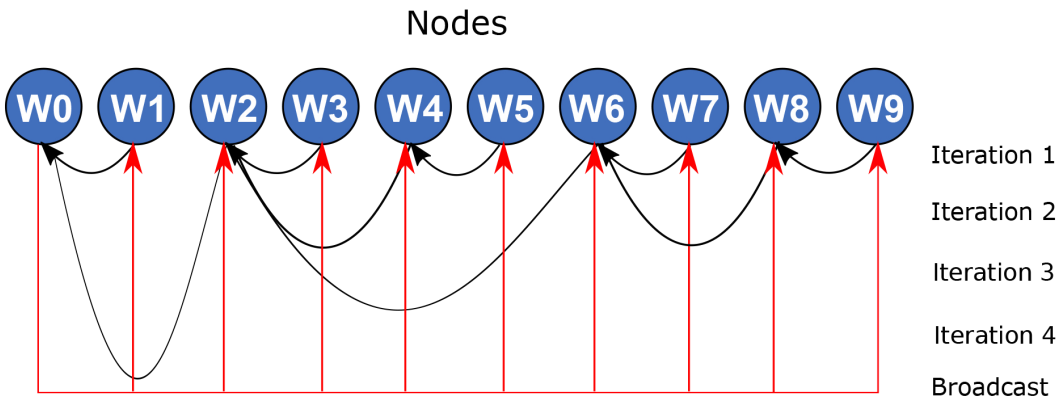


Figure 6.23: Distributed merge of the meta-data structures.

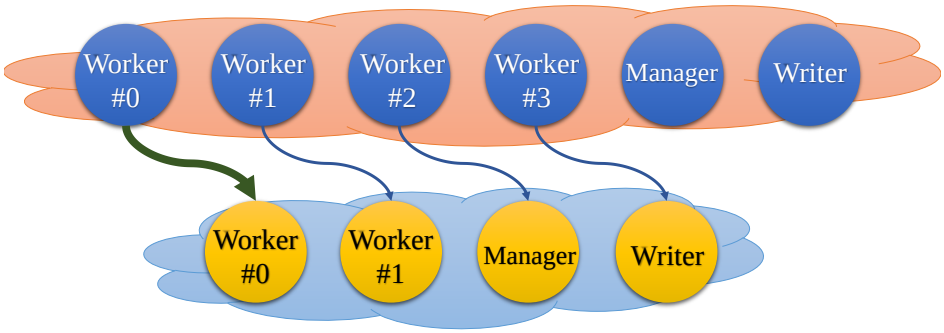


Figure 6.24: Data redistribution scheme of a shrink from 6 to 4 processes.

Figure 6.24 illustrates the redistribution pattern when shrinking from 6 to 4 processes. The first worker sends its common and worker data to the MPI rank 0 in the new communicator (wide arrow in the figure), and the remainder processes send their common data to their peer MPI ranks (narrow arrows in the figure).

6.6.2.3 HPG-aligner Malleable Version Outline Using the DMR API

Once the modified HPG-aligner workflow has been presented and the data redistribution patterns have been defined, we can outline the HPG-aligner malleable main loop, which proceeds as follows (see Listing 6.3). When a process finish its part on the processing of the first n chunks (line 5), it will reach the malleability point and call the `dmr_check_status()` function. If no resize action is planned, the current processes will proceed with the next n chunks. Else, if an expanding or shrinking action is scheduled, the corresponding data redistribution pattern, described on Section 6.6.2.2, is applied. Since the actual data redistribution is performed by the DMR API runtime, all that is required is to indicate in the source code which data should be distributed and to whom, using a DMR API OmpSs-like `#pragma` directive. After the data redistribution, the new processes will proceed with the next n chunks.

Notice that the `dmr_check_status()` function allows to define some malleability conditions that the RMS will take into account. These are: i) the minimum number of processes (`MIN`), ii) the maximum number of processes (`MAX`), and iii) the preferred number of processes (`PREF`) that should be assigned to this application. If a resize is due, the function will return: the action, the new number of processes (in `handlerNProcs`) and the new MPI communicator (in `handler`).

```
void hpga_malleable(void *cData, void *wData, int chunkIndex) {
  for (ci = chunkIndex; ci < TOTALCHUNKS; ci += N) {

    /* Computation */
    do_my_part();

    /* Malleability point */
    action = dmr_check_status(MIN, MAX, PREF, &handlerNProcs, &
   handler);
    if (action == EXPAND) {
        if (am_i_a_worker()) { // Workers processes
          #pragma omp task in(cData) in(wData) onto(handler, myRank)
          hpga_malleable(cData, wData, ci);
        } else { // Manager or Writer process
            for (dst = firstDst(); dst < getDsts(); dst++) {
              #pragma omp task in(cData) onto(handler, dst)
              hpga_malleable(cData, NULL, ci);
            }
        }
    } else if (action == SHRINK) {
      merge_in_rank0(wData);
      if (myRank == 0) {
        #pragma omp task in(cData) in(wData) onto(handler, myRank)
        hpga_malleable(cData, wData, ci);
      } else if (myRank < handlerNProcs) {
          #pragma omp task in(cData) onto(handler, myRank)
          hpga_malleable(cData, NULL, ci);
      }
    }

  }
}
```

Listing 6.3: HPG-aligner malleable version outline using the DMR API.

As for the `#pragma omp task` directives, as previously stated, these are used to indicate which data should be redistributed and to whom. For example, the directive:

```
#pragma omp task in(cData) in(wData) onto(handler, myRank)
```

indicates that the `cData` (common data) and `wData` (worker data) structures should be redistributed to the new `myRank` MPI process in the `handler` communicator.

6.6.3 Experimental Results

In this section we describe the setup for the experiments, the results obtained with a production-size dataset, and, the results achieved when a larger dataset is employed.

6.6.3.1 Experimental Setup

The experiments were performed using the *Marenostrum IV* cluster at the Barcelona Supercomputing Center (BSC). Each cluster node integrates 2 Intel Xeon Platinum 8160 (24 cores running at 2.10 GHz each) for a total of 48 cores with 96 GiB of RAM. The nodes are interconnected through a 100 Gb/s Intel Omni-Path network. One out of the 50 nodes of a standard queue in *Marenostrum IV* was used to run the Slurm manager daemon while the remaining 49 nodes were used to run the jobs. The following software versions were used: MPICH 3.2, OmpSs 15.06, and Slurm 15.08.

Workloads Setup We have generated fixed and malleable workloads of 100, 250, 500, 1,000, and 2,000 jobs, where all the jobs in a fixed workload are not malleable; all the jobs in a malleable workload are malleable.

The workloads were generated using the same methodology used in 6.5.1. Concretely, for our experiments, we have customized the following parameters of that model: i) the number of jobs to be launched, and ii) the jobs inter-arrival time which was modeled using a Poisson distribution with factor 10, in order to prevent receiving bursts of jobs while preserving a realistic job arrival pattern.

Jobs Setup Each job in a workload will execute an instance of HPG-aligner with a simulated dataset of RNA reads and the GRCh37.p73 human genome[4] as the reference genome.

As for the malleable jobs, the number of chunks have been set to be 4 times the maximum number of *workers*. Also, the number of chunks to be processed on each iteration has been selected to be variable and equal to the number of *workers* available at that moment.

6.6.3.2 Validation of the Proposed HPG-aligner Malleable Version

To compare the original and the malleable versions of HPG-aligner outputs, we have executed both versions 10 times and collected their execution times and results.

The input to both versions consisted of a dataset with 20 millions of 100 nucleotides RNA reads (generated with BEERS [18]), and the reference genome GRCh37.p73 human genome.

On the malleable version, we configured the number of chunks to be equal to the number of initial *workers* in order to mimic a static dataset distribution and to prevent any reconfigurations.

The execution times were virtually the same for both versions, with close-to-negligible time differences, of 0.28%, which were due to the malleability code overhead.

As for the outputs obtained by each version, it should be noted that HPG-aligner output depend on the order in which the reads are processed [49]. Therefore, it is not possible to obtain the same results on different parallel executions. However, we can consider that the obtained results are

[4] `http://www.ensembl.org/index.html`

comparable if they present a similar accuracy. As the accuracy deviation of the results obtained by the malleable version with respect to those in the original version was a negligible ±0.2%, we can conclude that both outputs are correct.

Therefore, the malleable version of HPG-aligner proposed in this chapter is functionally equivalent to the HPG-aligner in [49]. Furthermore, when no reconfiguration is performed, the execution time of the malleable version is almost the same as the original version.

6.6.3.3 Experiments with a Production-size Dataset

The input production-size dataset used in these experiments consists of 40 millions of 100 nucleotides RNA reads, sizing 8 GiB in total. It was generated using BEERS [18].

Since the malleable version of HPG-aligner has to inform the RMS of its minimum, maximum, and preferred number of jobs, we have first experimentally determined these for the already described experimental setup. For this purpose, we have launched the HPG-aligner malleable version with different number of processes and obtained their execution time, which is shown on Table 6.6.

While the lower and upper bounds are the minimum and the maximum number of processes allowed for this job, respectively, the preferred configuration is a number of processes with a fair trade-off between performance and the amount of resources assigned to that job. Table 6.6 presents the execution time (second column) for each process configuration (first column) of the malleable version of HPG-aligner. The table shows that with 12 processes the application reaches its highest performance: hence this will be used as the upper limit.

The lower limit and the preferred value were obtained, again, from the Equation 6.1. The results are collected in the third column of Table 6.6. With this metric, we obtained the relative increase in the gain performance of each configuration comparing the target configuration performance (t_i) with its previous (t_{i-1}) configuration. The reference point for calculating the gain was the baseline configuration performance (t_0). The 3-process configuration was considered the baseline, because we need, at least, a *worker*, a *manager*, and a *writer* process.

On the one hand, the lower limit is determined by the first configuration whose gain difference does not reach a threshold of 75%. In this case, the 6-process configuration cannot reach that threshold, and the lower limit is defined in 3 processes. On the other hand, in order to determine a "sweet spot" of execution, where we can reach a fair balance between resources and performance, we have set a threshold that yields a 25% reduction in time compared with the previous configuration of processes. Again, the last column of Table 6.6 shows that difference, and points to 6 processes as the preferred value, since the use of 12 processes does not meet the specified requirements. This application is I/O-bound and we cannot expect higher speedups when increasing further the number of processes.

In these experiments, we always submit the jobs requesting between 3 and 12 nodes. In our case, the possible values will be 3, 6, and 12; the resource manager decides the number of nodes to be assigned to the job before being initiated.

The malleable version of HPG-aligner can be initiated with fewer processes than its rigid counterpart, so the average execution time can be negatively affected. This can be seen on Figure 6.25, where the slow-down in the average execution time of the malleable jobs reaches 60%. However, when a job is submitted to a cluster, not only its execution time is taken into account, but the time elapsed from the submission to the execution (that include the waiting time) has to be also considered. Jobs are expected to be initiated earlier on a malleable workload because the running jobs can release part of their resources in favor of the pending jobs. As expected, Figure 6.26 shows that the average waiting time in a malleable workload can be up to 77% lower than that in the fixed counterpart.

Table 6.6: Execution time and gain difference of HPG-aligner malleable version when executed with an input dataset of 40 millions of 100 nucleotides reads for different numbers of processes.

#	Execution time (s)	Gain slope
3	238	—
6	68	71.43%
12	40	11.76%
24	44	-1.68%
48	60	-6.72%

Figure 6.25: Average job execution time on different fixed and malleable workloads with an increasing number of jobs.

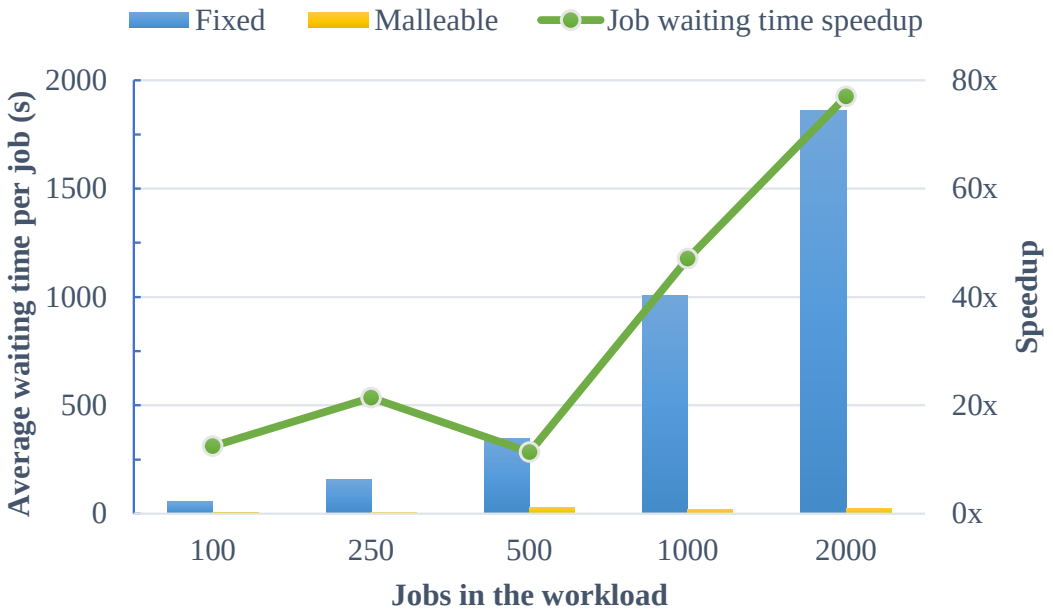


Figure 6.26: Average job waiting time on fixed and malleable workloads with an increasing number of jobs.

All in all, the balance is positive for malleable workloads, since the average job completion time, i.e., the average job waiting and execution time, may be up to 20 times faster than that for its equivalent fixed workload, as can be seen in Figure 6.27.

Figure 6.28 shows the completion time (the elapsed time between the first job was submitted and the last job is completed) of the different fixed and malleable workloads. As it can be seen, the completion time of the malleable workloads is up to 16% faster than their fixed counterparts. This result is specially significant from a cluster administrator/manager point of view, since it shows how the throughput of the system is increased when the malleable workloads are used.

Figure 6.29 details the evolution along the time of: i) the allocated nodes (top chart), ii) the concurrent jobs being executed (red and blue lines on top chart), and iii) the completed jobs (bottom chart) on the 1,000 jobs of fixed and malleable workloads.

The flat shape in the top chart of Figure 6.29 depicts how the nodes have been allocated during the execution of the fixed workload. It reveals that almost all the resources are allocated all the time. On the other hand, the malleable workload finishes earlier, mainly because more resources are generally available and that provides higher flexibility on the scheduling of new jobs.

The red and blue lines on the top chart of Figure 6.29 represent the concurrent running jobs on the fixed and malleable workloads, respectively. It can be seen, as was expected, that for the current experimental setup, the fixed workload is almost always running 4 jobs (with 12 nodes) at the same time. On the other hand, the malleable workload presents a greater variability, where the 8-job (with 3 nodes) configuration is the most used.

Finally, the bottom chart of Figure 6.29 represents the time evolution of the completed jobs on the fixed and malleable workloads. The evolution of both is overlapped until second 1,000, where more jobs per second begin to be completed on the malleable version. This behavior is consistent with the observed differences on the average job completion time of both workload types with different number of jobs (see Figure 6.27): the differences were minimum for the 100 job workloads and became greater as the number of jobs in the workloads was increased. Furthermore, due to its

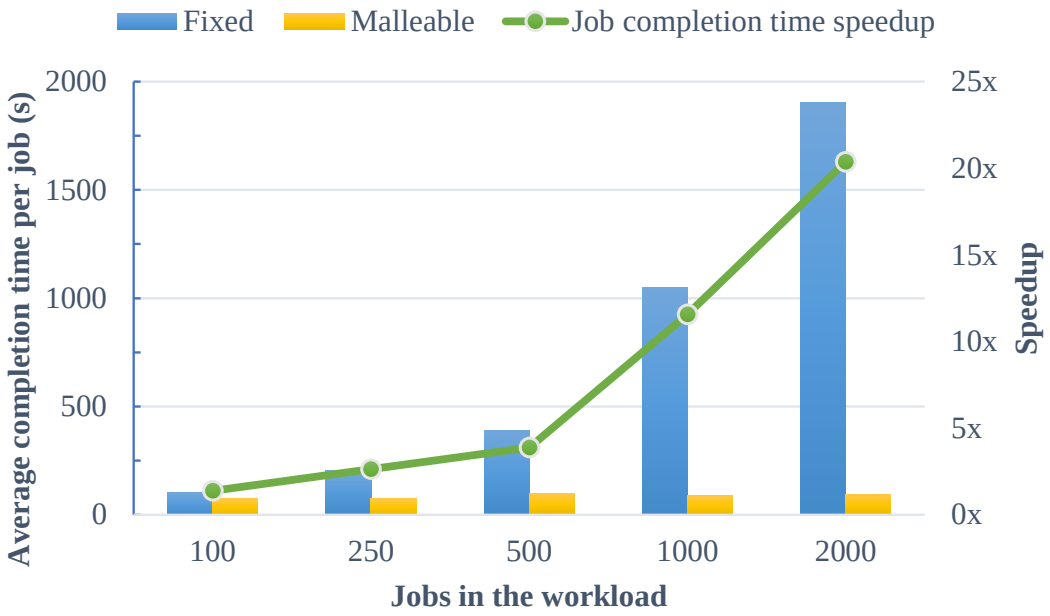


Figure 6.27: Average job completion time (waiting and execution time) on different fixed and malleable workloads with an increasing number of jobs.

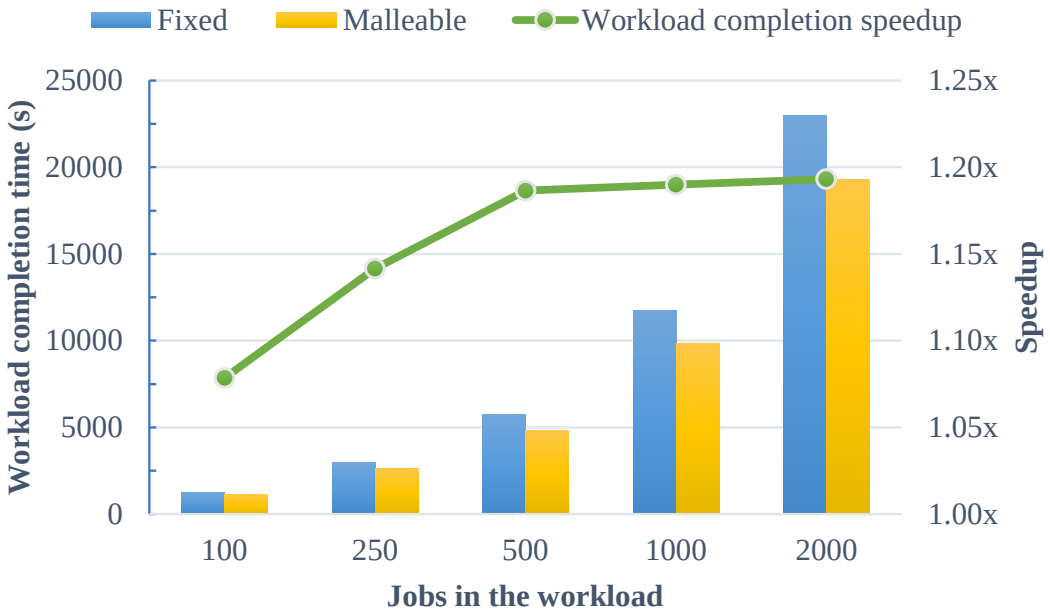


Figure 6.28: Completion time of different fixed and malleable workloads with an increasing number of jobs.

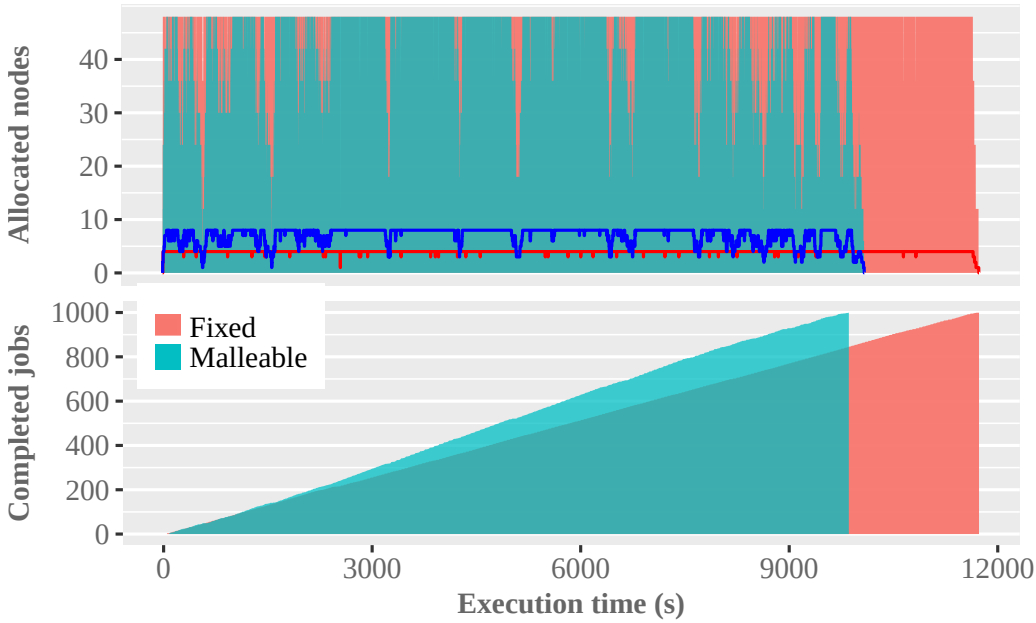


Figure 6.29: Time evolution of the allocated nodes (top chart), the concurrent jobs being executed (blue and red lines on top chart), and the completed jobs (bottom chart) on the 1000 jobs fixed and malleable workloads.

higher throughput, the malleable workload ends before the fixed counterpart.

The results presented so far can be summarized in a remarkable improvement of the individual job completion time when a malleable workload is processed. Figure 6.30 illustrates this fact, since it shows that all the malleable jobs are completed in almost a constant time, while the completion time of the fixed jobs becomes more variable: the later they are queued, the worse completion time they have.

6.6.3.4 Experiments with a Larger Dataset

The input dataset used in these experiments is larger than those currently used in production. We have considered this larger size as a possible future production value. It consists of 80 millions of 400-nucleotide RNA reads, 61 GiB in total[5]. This dataset was also generated using BEERS [18].

Again, since the malleable version of HPG-aligner has to inform the RMS of its minimum, maximum, and preferred number of jobs, we have first experimentally determined these for this particular experimental setup. In order to do this, we have launched the HPG-aligner malleable version with different number of processes and obtained their execution times, which are shown on Table 6.7, alongside with their gain slope. Using the gain slope metric, the minimum, preferred, and maximum number of processes are determined as explained on the previous section:

- minimum: 6
- preferred: 6
- maximum: 24

[5]the production-size dataset described in the previous section was 8 GiB

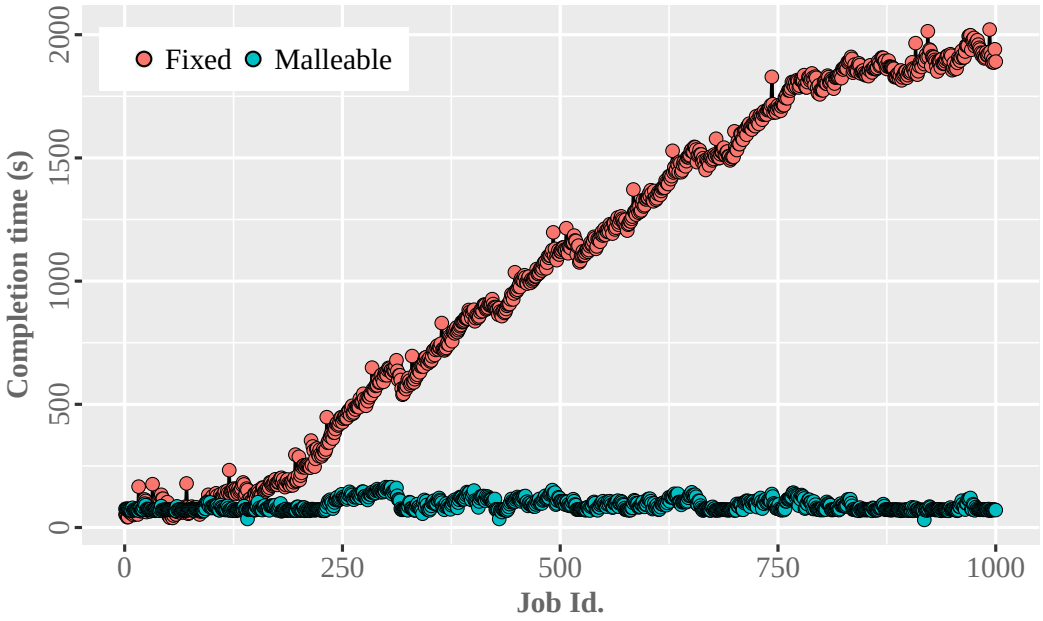


Figure 6.30: Completion time of each job on the fixed and malleable workloads. The greater the job id, the later the job was queued.

Table 6.7: Execution time and gain difference of the HPG-aligner malleable version when executed with an input dataset of 80 millions of 400-nucleotide reads for different number of processes.

#	Execution Time (s)	Gain Slope
3	1,382	—
6	345	75.04%
12	155	13.75%
24	152	0.22%
48	171	-1.37%

Figure 6.31: Average job completion time (waiting and execution time) on different fixed and malleable workloads with an increasing number of jobs.

The experiments reveal that the malleable workloads outperform their fixed counterparts. Figure 6.31 shows the average job completion time as the sum of the average waiting and execution times on different fixed and malleable workloads. The waiting time (blue part of the bars) is the predominant addend in all the cases, while the execution time (orange part of the bars) is far smaller in the malleable cases and almost imperceptible in the fixed workloads. It can also be seen in this figure that the average job completion time for the malleable workloads is around half of their fixed counterparts. In the interest of clarity, we have only shown the workloads of up to 500 jobs, which let us appreciate the difference among both averaged times.

The speedups of the job completion time, the job execution time, and the job waiting time on malleable workloads with an increasing number of jobs over their fixed counterparts are shown on Figure 6.32. The speedups of the job execution time are under 1, which indicates that jobs in the malleable workloads are running slower than fixed jobs. As already discussed, this is due to the malleable jobs being usually shrunk in order to accommodate for more running jobs. Nevertheless, the job waiting time speedup widely compensates the slower execution time, leading to a job completion time speedup of around 2 in all the workloads.

6.7 Conclusions

We have implemented a resource-aware malleability solution that improves the system behavior by targeting the global throughput of a high-performance facility. For this purpose, we have based on first-class tools the design of this new approach that introduce, a dynamic reconfiguration mechanism for malleable jobs, composed of two modules: the runtime and the resource manager.

As we prove in this chapter, our approach can significantly enhance resource utilization while, at the same time, reducing the wait-time for enqueued jobs and decrease the workload completion time. Although this is achieved at the expense of a certain increase in the job execution time, we have reported that, depending on the scalability of the application, this drawback can be negligible

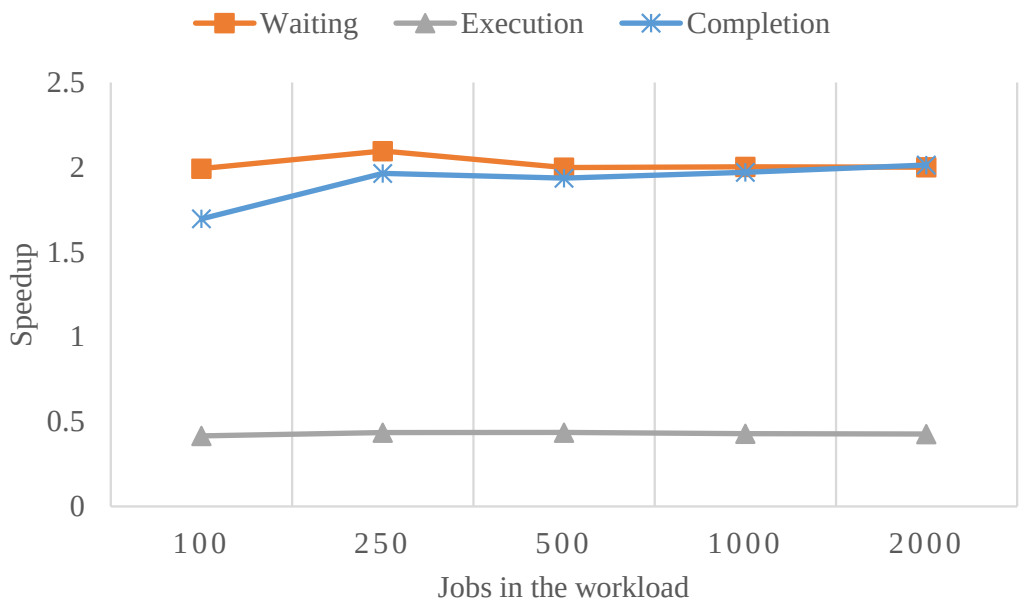


Figure 6.32: Speedups of the job completion time, the job execution time, and the job waiting time on malleable workloads with an increasing number of jobs over their fixed counterparts.

and utterly beneficial for the individual job completion time.

Furthermore, building upon the usability study of malleability tools, with the DMR API we expose other example of how to implement malleability in the sample code listed in Listing 3.1. Our implementation, based on an OmpSs-like syntax (see Listing 6.1), presents a more usable solution. The resultant code is less interfering than other solutions if we use as a baseline a non-malleable version of the program. Besides, the DMR API semantics allow the user to write easier code for implementing malleability.

Adapting a regular application to accommodate malleability can be a hard task, since usually applications are developed without ever considering the possibility of being requested to reconfigure their processes. Nevertheless, in this work we have shown that DMR API can be used to ease this task, and we expect to have started to pave the road towards dynamic job reconfiguration and the standardization of adaptive workloads through the inclusion of malleable jobs on them.

In particular, we have presented a malleable version of HPG-aligner using the DMR API. This is the first case, to our knowledge, where a non-iterative producer-consumer application with irregular communication patterns of complex data structures has been turned into malleable. This work, together with the different malleable applications presented in Section 6.5, prove that the DMR API can handle a wide variety of applications, including those featuring an irregular design.

As for the experimental results, we have shown that by adding malleability to HPG-aligner, the throughput of malleable workloads can be doubled. At the same time, since the waiting time is greatly reduced, the jobs in a malleable workload are completed faster.

CHAPTER 7

DMRlib: An MPI-like Malleability Solution

Apart from the Charm++ based reconfiguration solutions introduced in Section 3.2, the DMR API presents one of the most user-friendly interfaces for malleability. With these new paradigms, users are expected to make less effort to implement malleability in their applications; however, users have to learn the specifics of each programming model and their particular syntax.

Although the DMR API provides a highly usable interface, irregular applications (e.g., applications implementing a consumer–producer scheme) require a special effort to implement reconfiguration capabilities, because not every process features the same data structures. Another disadvantage that we find is in object-oriented applications (such as those leveraging C++ classes), which need a code refactoring regarding how the data is passed to the functions as parameters instead of as class attributes.

DMRlib is created with the intention of providing the user with a familiar-syntax-based interface to implement malleability. Compared with the reviewed approaches, the solution presented in this Chapter is basically composed of a trigger mechanism for reconfiguration that hides most malleability internals, providing users with the freedom to perform data redistributions using standard MPI routines.

7.1 The Dynamic Management of Resources Library

The dynamic management of resources library has been designed to ease malleability adoption by application developers. Built on top of the DMR API, DMRlib hides all the interactions among the application, the runtime, and the RMS. For this purpose, the library is in charge of the whole job reconfiguration procedure, honoring the malleability parameters provided by the user.

DMRlib enables the communication with Nanos++ (the OmpSs runtime) and the RMS. The RMS is an extension of Slurm with support for malleability, previously used by the DMR API.

7.1.1 Main Procedure

DMRlib's main procedure is a macro that triggers and handles the whole reconfiguration process. The macro, defined in Listing 7.1, expects five arguments corresponding to five function names:

Listing 7.1: Reconfiguration macro definition in DMRlib.

```
1 #define DMR_RECONFIG(compute, send_expand, recv_expand, send_shrink,
    recv_shrink)
```

- `compute`: The function that will be executed when the reconfiguration procedure ends and the child processes resume the execution of the application. Typically, in an iterative application, this function is the same function that invokes the reconfiguration.

- `send_expand`: Function executed by parent processes in *Comm 1* when performing an expansion. This function implements the algorithm for sending data from parent processes to child processes.

- `recv_expand`: Function executed by child processes in *Comm 2* when performing an expansion. This function implements the algorithm for receiving data in child processes from parent processes.

- `send_shrink`: Similar to the `send_expand` but for a shrink.

- `recv_shrink`: Similar to the `recv_expand` but for a shrink.

The macro proceeds as Algorithm 7 shows. First of all, in order to define the operations of the processes, the current stage of the reconfiguration is checked (line 1). For this purpose, the library tries to retrieve the parent communicator. If there is a parent communicator (line 2), it will be used to handle the data redistribution. Line 3 determines, by comparing the number of processes in both communicators, if the resize action is an expansion or a shrink. Thus, if the current global communicator (`MPI_COMM_WORLD`) contains a larger number of processes than the parent communicator, the library invokes the user function for receiving data in an expansion (line 4); otherwise, the receive data function for shrinks will be called (line 6).

In case `MPI_Comm_get_parent()` returns "null", meaning that the current communicator does not have a parent and hence no reconfiguration is occurring, the application communicates its readiness of being resized to the runtime (line 9). At this point, *action* may obtain three values: "expand", "shrink", or "none". In the latter case, the program execution continues normally (line 27); otherwise, the macro performs the job rescaling arrangements. For the sake of clarity, in our examples we always assume that the resizes are *multiple of* or *divisible by* the number of processes in the invoking communicator.

In line 11, the scalability factor is calculated to determine the number of links to be established by each initial process. With this *factor*, each process assesses its peer processes in the new communicator (line 13) and establishes the communication to identify the function where the execution will be resumed after the reconfiguration (lines 14 and 15). In line 16, the data is sent utilizing the function provided by the user `send_expand()`. Finally, the new processes are disconnected from *Comm 1*, being allowed to continue the execution of the application in a new computational step, whereas the initial processes terminate their execution (line 17).

In case of a shrink (line 19), the procedure is similar to that in an expansion. DMRlib calculates the communication factor (line 20), establishes the communication among the processes, and sets the resuming point for the execution (line 23). Finally, in this case the data is sent using the function `send_shrink()` and processes are detached from the initial communicator in line 25.

7.1.2 Parameterization

DMRlib provides routines to fully customize the job malleability. For instance, with the function `DMR_Set_parameters()`, the user is able to define the boundaries of malleability in terms of number of processes. The arguments required by this function are:

- `min`: the minimum number of processes to run the job—the lower limit for malleability.
- `max`: the maximum number of processes to run the job—the upper limit for malleability.
- `pref`: the preferred number of processes to run the job. This is a special value that the reconfiguration policy in Slurm considers to schedule an action.

Although the DMR API has proved to pose very low scheduling times, in those applications performing short-step executions—that is, computational steps which only need a few milliseconds to finish—requesting reconfigurations in every step may generate a non-negligible overhead, since the time taken by the computational step is in the same scale that the time needed for the reconfiguration scheduling. For this reason, DMRlib implements two mechanisms for controlling scheduling reconfigurations:

- `DMR_Set_sched_period`: this function expects a number of seconds. During this period all the reconfiguration scheduling requests will be ignored.
- `DMR_Set_sched_iteration`: this function expects a number of computational steps. For all the steps inside the given iterations, the scheduling requests will be ignored.

7.1.3 Usage

In order to turn an application into malleable, a user has to call into its code the macro `DMR_RECONFIG`, as illustrated in Listing 7.2. This excerpt of code shows the function (line 1) containing the main loop and hence where the malleability will occur. This function features three parameters: the data structure pointer, its size, and the current iteration (if it is the first call, *step* will be usually set to zero). The function starts by configuring the malleability limits in line 2. These limits may be modified any time to meet the requirements of different computational stages in the application.

At the beginning of each iteration of the main loop (line 3), the `DMR_RECONFIG` macro checks if a reconfiguration is being performed or if the resource manager can improve the system status by resizing the job leveraging the DMRlib (line 4). The macro has to be used indicating the appropriate functions for the reconfiguration: the first function is the invoking function itself, while the remaining four arguments are user-defined redistribution function calls. Notice that "receiving" functions in the macro use the memory address instead of the pointer (or the value itself for the *data_ size* variable). The reason is that those functions are called just after the new processes have been spawned, and before receiving the data these have to allocate new memory. The rest of the function will remain unaltered (line 5).

Listing 7.3 shows an example of the redistribution functions used in the previous macro for the case of an expansion action (`send_expand()` and `recv_expand()`). Here we describe a case similar to that shown in Figure 6.4a, that is, data transfers for an expansion to a multiple of the initial number of processes.

The first function (line 1) will be executed by the processes in the initial communicator, which are in charge of sending the data. Dividing the number of spawned processes by the number of current processes returns the scalability factor of this expansion (line 2). With this value, we

```
1 void compute(double *data, int data_size, int step) {
2     DMR_Set_parameters(min, max, pref);
3     for (int i = step; i < TOTAL_STEPS; i++) {
4         DMR_RECONFIG(compute(data, data_size, i), send_expand(data,
      data_size), recv_expand(&data, &data_size), send_shrink(data,
      data_size), recv_shrink(&data, &data_size));
5         /* Computation */
6     }
7 }
```

Listing 7.2: Enabling malleability using DMRlib in a user code.

calculate the size of the data chunks (line 3). Furthermore, *factor* is used to determine the number of sends each original process performs (line 4) and the rank in the new communicator which is the destination of each chunk of data (line 5). In line 6, we use the standard `MPI_Send` function, defining the buffer pointer to the appropriate data chunk (argument 1), its size (argument 2), its destination rank (argument 4), and the new communicator (argument 6). DMRlib provides the variable `DMR_INTERCOMM` to represent the inter-communicator between the original and spawned processes.

The second function in the listing (line 10) is called by the processes in the spawned communicator, which are the responsible for receiving the data and continuing the execution. Again, the scalability factor is calculated using the same operation as in the former case (line 11); however, since this is performed by the processes of the new communicator, the variable names are swapped. With this value we obtain the source rank of the data in the initial communicator (line 12). Chunk size is calculated (line 13) and used to allocate memory for the data structure (line 14). In this process, the variables *data_ size* and *data* are overwritten; the data array (*data*) is a null pointer in the memory of the new processes and hence we have to allocate the necessary memory. Eventually, the `MPI_Recv` operation is performed receiving as arguments the memory pointer to store the data (argument 1), the number of received elements (argument 2), the source rank in the initial communicator (argument 4), and the initial communicator itself (argument 6).

7.1.4 Predefined Redistribution Patterns

In an effort to ease the coding of the data redistribution process, the DMRlib design provides predefined redistribution functions with the most common communication patterns. The current version of the library provides the following patterns:

- **Default Redistribution**: This pattern corresponds to a classic uniform distribution to a number of processes multiple of or divisible by the original (i.e. Figure 6.4a).
- **Block Cyclic Redistribution**: It implements the data transfers necessary when data is block-cyclically distributed over the processors (`https://computing.llnl.gov/tutorials/parallel_comp/#distributions`).

Table 7.1 shows the headers of all the redistribution functions currently available in DMRlib. All of them require the same two initial arguments: the pointer to the data (when receiving, a pointer address) and the MPI data type. For the default redistribution these also need the number of elements of the data array, while for the block cyclic, the functions require the number of blocks and their size.

```
void send_expand(double *data, int data_size) {
    factor = dmr_intercomm_nprocs / comm_world_nprocs;
    new_data_size = data_size / factor;
    for (i = 0; i < factor; i++) {
        dst_rank = my_rank * factor + i;
        MPI_Send(data + new_data_size * i, new_data_size, MPI_DOUBLE,
    dst_rank, tag, DMR_INTERCOMM);
    }
}

void recv_expand(double **data, int *data_size) {
    factor = comm_world_nprocs / dmr_intercomm_nprocs;
    src_rank = my_rank / factor;
    *data_size = (*data_size) / factor;
    *data = malloc((*data_size) * sizeof (double));
    MPI_Recv(*data, *data_size, MPI_DOUBLE, src_rank, tag,
   DMR_INTERCOMM, MPI_STATUS_IGNORE);
}
```

Listing 7.3: User data redistribution functions for an expansion.

Table 7.1: Predefined Redistribution Headers in DMRlib

Default Redistribution
void DMR_Send_expand_default(void *data, MPI_Datatype type, int size);
void DMR_Recv_expand_default(void **data, MPI_Datatype type, int *size);
void DMR_Send_shrink_default(void *data, MPI_Datatype type, int size);
void DMR_Recv_shrink_default(void **data, MPI_Datatype type, int *size);
Block Cyclic Redistribution
void DMR_Send_expand_blockcyclic(void *data, MPI_Datatype type, int n, int size);
void DMR_Recv_expand_blockcyclic(void **data, MPI_Datatype type, int *n, int *size);
void DMR_Send_shrink_blockcyclic(void *data, MPI_Datatype type, int n, int size);
void DMR_Recv_shrink_blockcyclic(void **data, MPI_Datatype type, int *n, int *size);

```
void compute(double *data, int data_size, int step) {
    DMR_Set_parameters(min, max, pref);
    for (int i = step; i < TOTAL_STEPS; i++) {
        DMR_RECONFIG( compute(data, data_size, i),
                      DMR_Send_expand(data, data_size),
                      DMR_Recv_expand(&data, &data_size),
                      DMR_Send_shrink(data, data_size),
                      DMR_Recv_shrink(&data, &data_size));
        /* Computation */
    }
}
```

Listing 7.4: Enabling malleability using the default redistribution pattern of DMRlib.

Table 7.2: Malleability Solutions Usability Comparison

	# Lines	Data transfer	MPI library	Standard MPI	Paradigm
Bare MPI	26	Manual	Any	Yes	MPI
PCM API	27	Manual	MPICH	No	MPI
AMPI	13	Auto*	-	Yes	Charm++
Flex-MPI	21	Manual	MPICH	No	MPI
Elastic MPI	26*	Manual	MPICH	No	MPI
DMR API	17	Auto	Any	Yes	OmpSs
DMRlib	13	Auto*	Any	Yes	MPI

In the example in Listing 7.2, we could implement the same functionality using the *default redistribution* pattern as presented in Listing 7.4, hence avoiding the user implementation of data redistribution presented in Listing 7.3.

7.2 Usability Evaluation

This section evaluates the usability, from the software developer point of view, of the malleability solutions studied in Section 3.2.3 together with the DMR API, introduced in Chapter 6, and DMRlib, presented in this chapter.

Table 7.2 compares the most relevant usability features. The number of lines (second column) associated to each framework corresponds, respectively, to Listings 3.1, 3.2, 3.3, 3.4, 3.5, 6.1 and 7.2 (for the latter we have added the lines of the *main* function). Notice that Elastic MPI number of lines lacks the data redistribution lines of code.

The reduction of the number of lines of code is closely related to the type of data transfers (third column). The solutions with *automatic* data transfers (AMPI, DMR API, DMRlib) are able to drastically reduce the coding effort since the runtime is responsible for managing the data. However, we catalogue AMPI and DMRlib as *automatic* provided given that we use "isomalloc" in AMPI (see Section 3.2.3.2) and the *predefined redistribution patterns* in DMRlib (see Section 7.1.4).

An additional important issue is the solution dependency on the underlying MPI library. The fourth column of Table 7.2 shows which frameworks are bound to a specific MPI implementation.

This is relevant because it makes them susceptible to changes. While AMPI[1] is an implementation of MPI on top of Charm++'s runtime, the solutions that admit *any* MPI library can be expected to be more reliable, portable and long-lasting. The fifth column indicates which frameworks have also been adapted to the MPI-2 standard[2] in order to provide malleability.

In terms of usability, we consider beneficial a lower number of lines, an automatic data transfer, support for any MPI implementation, and the fact that the framework does not modify the MPI standard. Although AMPI and the DMR API meet all those requirements, we also consider crucial to provide a solution that follows the MPI programming paradigm instead of Charm++ or OmpSs. The majority of solutions adhere to the MPI programming paradigm (fourth column) since it is widely accepted in HPC. Hence, we assume that the MPI paradigm is preferable in terms of usability. For this reason, we consider DMRlib a highly appealing solution for malleability.

7.3 Experimental Evaluation

In this section we present four malleable applications implemented with DMRlib and how they have been used in a production cluster to evaluate the benefits of adaptive workloads. In this case the evaluation was performed using 129 nodes from the Marenostrum IV supercomputer at Barcelona Supercomputing Center (description of the infrastructure detailed in Section 6.6.3.1). One out of the 129 nodes hosted the Slurm management daemon, while the remaining were used as compute nodes.

At this point, it is worth noting that we cannot contrast performance results with other already existent malleability solutions because of, to the best of our knowledge, they are not publicly available or present such different working modes that make impossible a fair comparison.

7.3.1 Malleable Applications

We have leveraged DMRlib to turn into malleable the implementations of the CG method[3], the Jacobi method[4], the N-body simulation[5], and the bioinformatics tool HPG-aligner[6]. Table 7.3 shows the problem size and configuration of each application.

For CG and Jacobi, since both handle double-precision elements, the coding process for malleability is quite similar, although these do not feature the same number of data structures. While Jacobi deals with a flat-stored square matrix plus 2 arrays, CG handles 2 more arrays. For CG, the `DMR_RECONFIG` call may be invoked like this:

```
DMR_RECONFIG( CG(m, a1, a2, a3, a4, size, step),
              send_expand(m, a1, a2, a3, a4, size),
              recv_expand(&m, &a1, &a2, &a3, &a4, &size),
              send_shrink(m, a1, a2, a3, a4, size),
              recv_shrink(&m, &a1, &a2, &a3, &a4, &size));
```

For instance, the sending function for an expansion may include the following calls to the already implemented redistribution functions:

[1] `http://charm.cs.uiuc.edu/research/ampi/`
[2] `http://www.mpi-forum.org/docs/mpi-2.2/index.htm`
[3] `https://en.wikipedia.org/wiki/Conjugate_gradient_method`
[4] `https://en.wikipedia.org/wiki/Jacobi_method`
[5] `https://en.wikipedia.org/wiki/N-body_simulation`
[6] `https://github.com/opencb/hpg-aligner`

Table 7.3: Applications Configuration

Application	Input Data	Iterations
CG	A square matrix and 4 arrays of 32,768 elements	10,000
Jacobi	A square matrix and 2 arrays of 16,384 elements	10,000
N-body	6,553,600 particles	50
HPG-aligner	40 millions 100-nucleotide reads	$\#workers \times 4$

```
void send_expand(double *m, double *a1, ..., int size) {
    DMR_Send_expand_default(m, MPI_DOUBLE, size * size);
    DMR_Send_expand_default(a1, MPI_DOUBLE, size);
    ...
}
```

N-body only manages an array in the redistribution; however, this array is composed of *particles*, which is a non-standard data type. For this reason, we defined an MPI struct derived datatype named `MPI_PARTICLE`, composed of 6 vectors of floats and 2 more single floats in order to leverage the predefined redistribution functions as follows:

```
DMR_RECONFIG( N-body(particles, size, step),
              DMR_Send_expand_default(particles, MPI_PARTICLE, size),
              DMR_Recv_expand_default(&particles, MPI_PARTICLE, &size),
              DMR_Send_shrink_default(particles, MPI_PARTICLE, size),
              DMR_Recv_shrink_default(&particles, MPI_PARTICLE, &size));
```

Finally, for HPG-Aligner, ad-hoc redistribution functions were developed, since it does not present a regular communication pattern. Notice that HPG-aligner presents a producer–consumer architecture, where two processes are in charge of reading/writing data while the remaining act as *workers*. For this reason, the minimum number of processes required to run this application is three (at least it needs one *worker* plus read and write processes).

With this set of applications we consider that a large variety of use cases are represented in the study. From N-body, which is believed to be highly relevant during the next decade, with applications in various domains such as scientific computing simulations or machine learning [2], to HPG-aligner, a producer–consumer application.

The malleability parameters of the applications have been calculated again using the gain difference equation introduced in 6.1. For example, in order to assess the gain difference for HPG-Aligner executed in 12 processes ($s(12)$), we calculate the completion time of the previous configuration ($t(6)$) minus its own time ($t(12)$). This is divided by the reference completion time of the minimum processes configuration, in this case $t(3)$ (notice that the reference of the rest of the applications is $t(1)$). The result is finally multiplied by 100.

Figure 7.1 depicts the gain difference for each configuration of the four applications. In order to determine the limits of malleability, we considered a 10% threshold (black line in the chart). The *lower limit* is defined by the first configuration to exceed this threshold, the *preferred* value is given by the last configuration before dropping below 10%, and the *upper limit* is the configuration with the highest performance. Although our cluster was composed of 128 compute nodes, we restricted the jobs to request a maximum of 32 nodes, assuming that a job should not monopolize more than a quarter of the cluster.

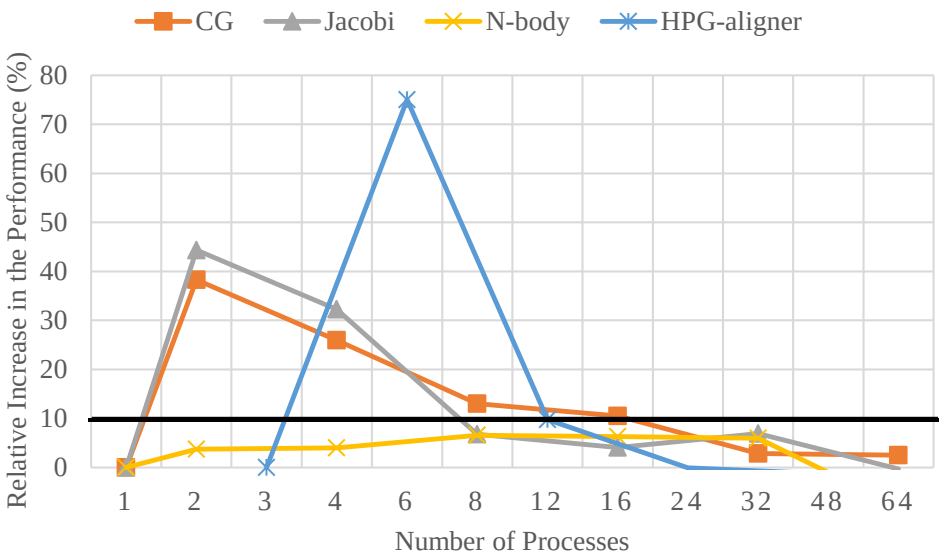


Figure 7.1: Gain difference of each application and a 10% threshold (thick line) to determine the limits of malleability.

Table 7.4: Malleability Parameters for the Applications

Application	Lower Limit	Upper Limit	Preferred	Scheduling Period
CG	2	32	16	10 seconds
Jacobi	2	32	4	10 seconds
N-body	1	32	1	-
HPG-aligner	6	12	6	-

Table 7.4 summarizes the malleability configurations. The last column includes the scheduling inhibitor periods for CG and Jacobi. In this case, a period of 10 seconds minimizes the number of reconfiguration requests reducing the overhead and without a significant impact in the scheduling. The other two applications do not need this inhibitor because their iterations are coarser grained.

7.3.2 Job Submission and Reconfiguration

Applications are submitted as jobs to the workload manager. Depending on the submission mode and whether jobs are malleable (i.e., if the system can resize automatically a job during its execution), we classify them in four categories, as shown in Table 7.5. For example, if a non-malleable job is submitted rigid, the job will be referred as "fixed". On the other hand, if a malleable job is submitted moldable, we consider the job as "flexible".

In this work we rely on Slurm to schedule jobs and manage resources. Apart from the rigid job

Table 7.5: Job Classification Depending How it Can be Resized

Job Malleable?	**Rigid submission**	**Moldable submission**
No	Fixed	Pure Moldable
Yes	Pure Malleable	Flexible

Table 7.6: Job Submission in Slurm using `sbatch`

Application	Rigid Submission	Moldable Submission
CG	-N32 ./cg	-N2-32 ./cg
Jacobi	-N32 ./jacobi	-N2-32 ./jacobi
N-body	-N32 ./nbody	-N1-32 ./nbody
HPG-aligner	-N12 ./hpgaligner	-N6-12 ./hpgaligner

submission, where jobs request a fixed number of resources, Slurm provides a moldable submission with which a job may request a range of resources.

We have implemented in Slurm a reconfiguration policy as described next. When a job triggers a reconfiguration, Slurm first checks if the job is running below its *preferred* configuration (this can only happen when leveraging a moldable submission). If this is the case and there are available resources, the job is expanded, never exceeding the *upper limit*.

The policy is designed to improve the throughput of the system. Therefore, if there are pending jobs in the queue, the policy checks if shrinking the evaluated job and deallocating part of its resources, other job could be initiated. When this action is scheduled, the queued job responsible for the shrinking is assigned the maximum priority to run. A job may only be shrunk—but never under *preferred*—if it is running above *preferred* and a pending job may benefit from the released resources; otherwise, if no pending job can be initiated and there are available resources, the job is expanded.

For the rigid submissions, jobs are always launched at their highest performance (*upper limit*) assuming that a user expects their job to run as fast as possible. The moldable submission defines its range between both limits: *lower* and *upper*. Table 7.6 describes how jobs are submitted in both modes using the performance analysis of Table 7.4.

We have generated workloads of jobs, where each job randomly instances one of the four applications. The workloads are composed of 100, 250, 500, 1,000 and 2,000 jobs, each size leveraging four homogeneous versions of fixed, pure moldable, pure malleable, or flexible jobs (see Table 7.5). With these workload sizes, in our study we cover from small workloads, with hardly any pending job, to large workloads where the queue of pending jobs grows significantly.

The average inter-arrival time is determined by the Feitelson model [15] with a factor of 1, what represents a highly loaded scenario where jobs are massively submitted while fitting the arrival Poisson distribution of the model.

7.3.3 Experimental Results

Figure 7.2 depicts the speedup, of different workload sizes, for the malleable workloads compared to the non-malleable ones in the metrics: average job waiting, execution and completion times. Lines are grouped by submission mode: the dotted lines correspond to rigid submissions, while thicker lines represent moldable submissions.

We start analyzing the rigid case (dotted lines). Although the average job execution time increases for the malleable jobs ($speedup < 1$), the completion time benefits from the reduction in the waiting time ($speedup \simeq 3.25x$). The chart also shows a strong correlation between the completion time and the waiting time. This leads to malleable jobs finalizing over 3x faster than their non-malleable counterpart in a workload when these are submitted in rigid mode.

In the case of moldable submissions (dashed lines), the speedup behavior in the averaged times is more regular when the workload reaches a minimum size. We can see again the relevance of the

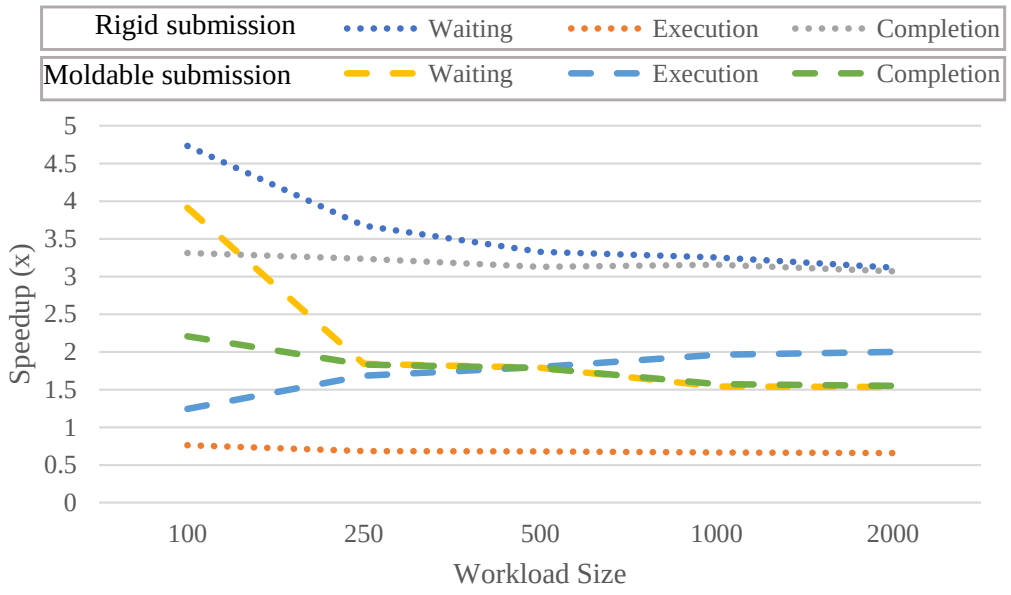


Figure 7.2: Comparison of the 4 types of workloads. The lines show the speedup of malleability for the average job waiting, execution, and completion time, grouped by submission mode.

waiting time for the job completion time, since the lines that represent their speedup are almost overlapped. When the workload size increases, the queued jobs reach a saturation degree where the waiting time cannot be improved further and the speedup remains constant around 1.5x. Apart from the waiting time, flexible jobs (malleable submitted moldable) show a higher average completion time speedup. In a workload of pure moldable jobs (non-malleable submitted moldable), the job execution time increases because jobs are likely to be initiated with few resources (it is easy to find a slot of 2 nodes rather than one of 32) and they have to finish their execution with their initial allocation.

As an example, we emphasize the 1,000-job workload experiment with moldable submission. Figure 7.3 illustrates the described behavior through the representation of the workload evolution over time. At the top chart, the shapes represent the resource allocation in each second, showing how flexible jobs are able to reallocate their resources to benefit from virtually all the nodes during the whole execution. On the other hand, pure moldable jobs keep their initial allocation and that is why the resource allocation decreases near second 10,000. The lines at the top chart depict the evolution of the number of running jobs over the time. The pure moldable workload (red line) not only displays a regular evolution, but also a higher average number of running jobs, since we have more small jobs running for long time. However, the flexible workload (blue line) redistributes the resources prioritizing running jobs by assigning them fewer nodes than their *preferred* values. The reconfiguration policy in Slurm always tries to compensate scenarios with a reduced load, such as the beginning of the workload, with an early initialization of execution of new jobs.

The bottom chart in Figure 7.3 represents the number of completed jobs in each second of the execution. The green shape unveils a sharper increase in the throughput in terms of completed jobs per unit of time. Accordingly to the speedup chart (Figure 7.2), flexible jobs are finishing an average of 1.5x faster, what, in this case, produces a 1.5x speedup of the global throughput when using malleability. In the case of the 100-job workload, the completion time speedup is 2.2x and the workload is processed 3x faster than the purely moldable counterpart.

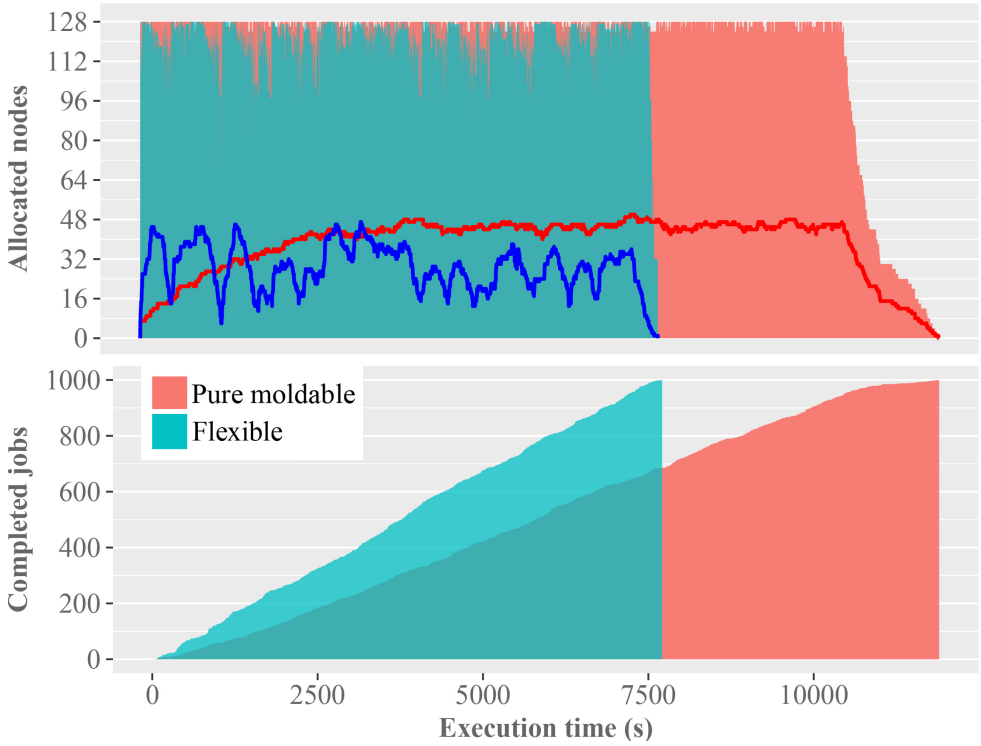


Figure 7.3: Comparison of the evolution in time of a 1,000-job for the pure moldable and flexible workloads. The top chart represents the allocated resources (shapes) and the number of running jobs (lines). At the bottom, the shapes show the number of completed jobs in each second of the execution.

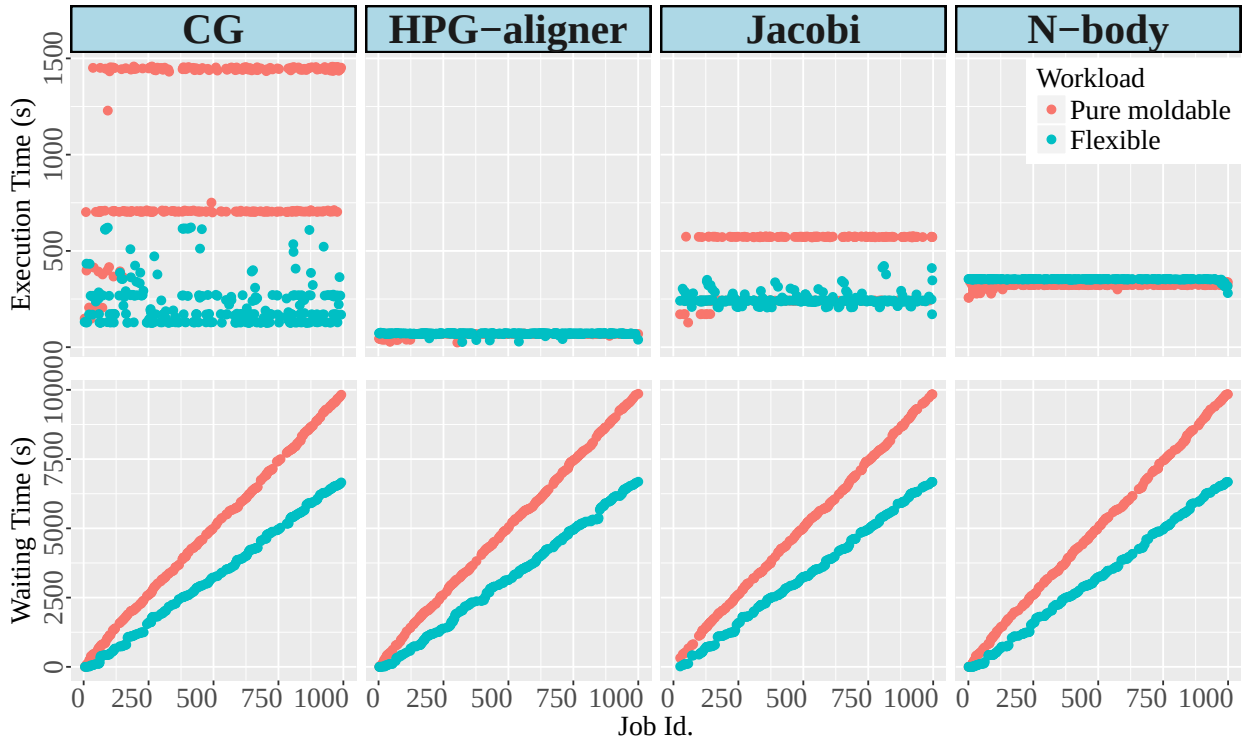


Figure 7.4: Execution and waiting times per job in the 1,000-job workload with moldable submission, grouped per application.

Continuing with the same example, Figure 7.4 depicts the waiting and the execution time of each job comparing the pure moldable and the flexible versions. At the top chart, the applications with a poor scalability (HPG-aligner and N-body) show virtually the same execution time in both versions. However, applications like CG or Jacobi show an interesting variability in the execution time. In the flexible case there is a visible trend to reduce the execution time by expanding the jobs. The pure moldable jobs cannot be resized during their execution and, as we have noted before, these are very likely to be launched in a reduced set of resources, stretching their execution. The bottom chart shows a regular behavior in the waiting time of all the applications, reaching a difference of more than 3,000 seconds for the last queued jobs.

Figure 7.5 gathers waiting, execution, and completion times in the same chart and groups per application the time difference for each job in of both versions. This chart reveals the strong correlation of the waiting time with the job completion time.

Figure 7.6a compares the workload completion time when using malleability in both submission modes: rigid and moldable. The vertical bars represent the total completion time for each configuration. Bars are grouped per workload size. The two first bars of each group correspond to the results when using the rigid submission, while the other two bars refer to the moldable submission. The first bars of these subgroups (the first and third) represent the non-malleable workloads, while the second bars (second and fourth) show the time for the malleable workloads. We have analyzed in depth the rigid submission (first and second bar of each group) and how positively malleability improves upon it with speedups of around 3x (see blue line). However, this chart also reveals the performance benefits of the moldable submission of non-malleable jobs (third bars). We can see that using the moldable submission, we can obtain a similar completion time to that obtained by a

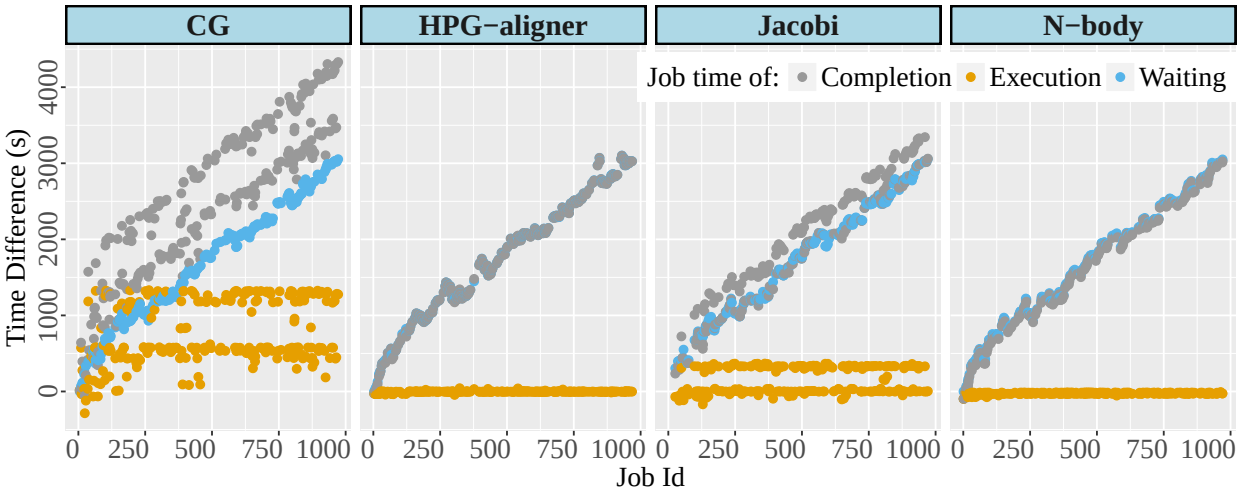


Figure 7.5: Time difference between the 1,000-job pure moldable and the flexible workloads, grouped per application

malleable workload with the traditional rigid submission.

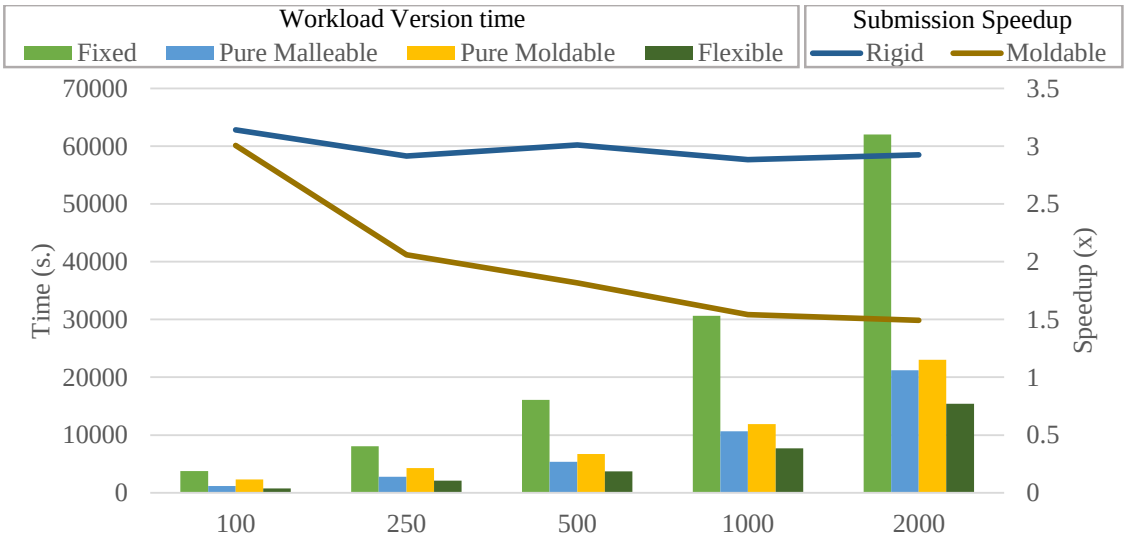
For this reason, *a priori* the moldable submission may be presented as an easy–to–adopt high-throughput solution. However, Figure 7.6b shows one of the most relevant drawbacks for its utilization in production environments: the increasing job execution time. While the individual average job execution time for rigid submission (first and second bars of each workload size) and flexible jobs (fourth bars) remain unaltered, jobs in the pure moldable workloads (third bars) experience an upward trend in this time. We have identified this behavior as the main obstacle for the adoption of the moldable submission in production environments. Although moldable submission is likely to improve the global job throughput without posing an additional effort to application developers (they are not expected to modify their code), these experience an undetermined increase in the execution time that depends on the queue load. The fourth bar of each group in the figure corresponds to the average job execution time that remains constant and yields speedups of up to 2x in the average job execution time.

Another drawback of leveraging moldable submissions without malleability is revealed in Figure 7.7, where we can see that the resource allocation rate drops for small workload sizes (third bar of each group). In this case, the pure moldable workload is under-utilizing resources when the number of jobs in the queue is moderated.

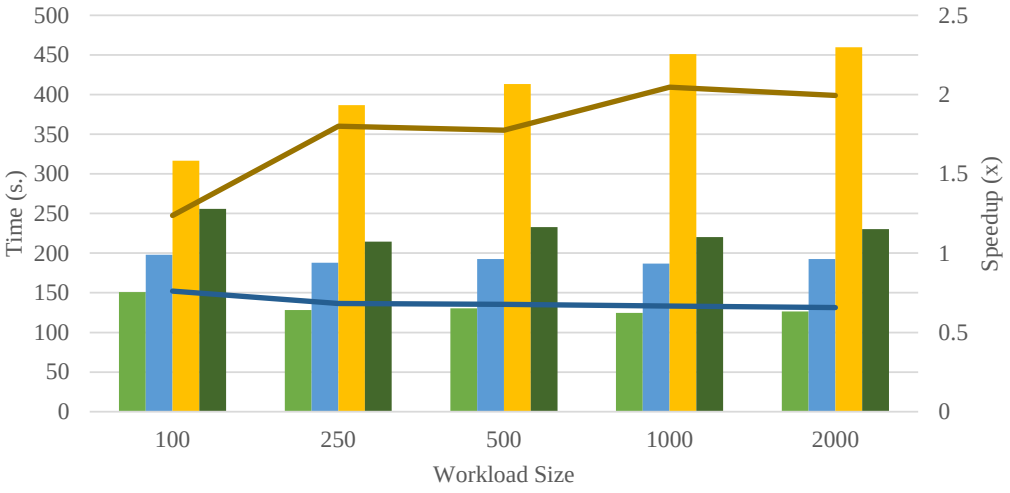
Identified and solved these issues, we can consider the moldable submission of malleable applications (referred as flexible jobs) as a remarkable technique for high-throughput computing (HTC), while DMRlib provides an unprecedented easy–to–use approach.

7.3.4 Energy Consumption

The reduction in time and the different usage of the resources derived from the use of DMRlib pose a strong impact on energy consumption. Figure 7.8 depicts the energy consumption of each configuration grouped by workload size (we have skipped the 2,000-job workload because it does not provide additional information and reduces the clarity of the chart). In this figure, the top bar of each size is the reference energy consumption representing the KW/h of executing a non-malleable workload with rigid submission of jobs. The remaining bars in the group, apart from the energy consumption, additionally display a label with the relative consumption with respect to the



(a) Workload completion time.



(b) Average job execution time.

Figure 7.6: Workload type comparison and speedup of submission modes.

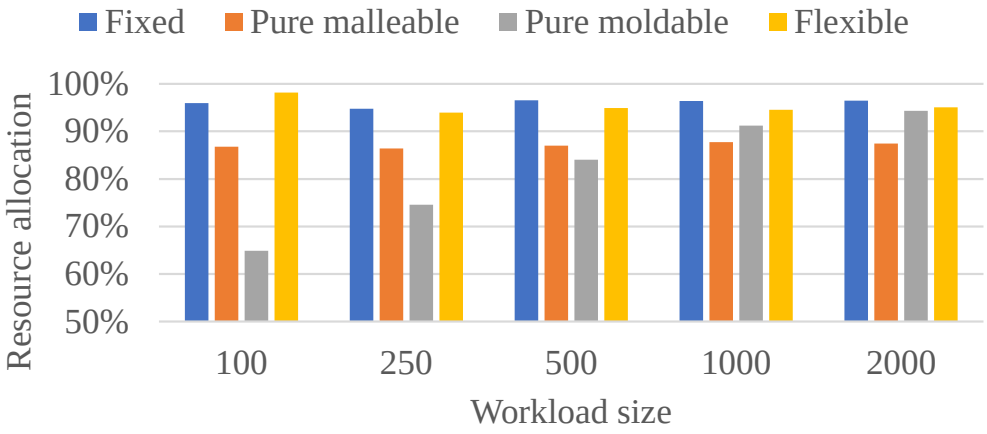


Figure 7.7: Workload type resource allocation comparison.

reference value.

From these results we observe that DMRlib, with its minimalist interface for malleability, can reduce the energy consumption up to around 70% (second bar of each group). Furthermore, just changing the submission mode and using the moldable method, the energy consumption rounds 40–50% (third bar) of its original value. Last, the greatest reduction is given by combining malleable jobs with moldable submissions, when workloads can be processed with a reduction in the consumption of almost 80%.

The energy was estimated using the consumption information provided by the technical support of Marenostrum IV: idle nodes consume 100 Wh, while loaded nodes consume 340 Wh. Energy is then simply calculated by taking into account the time each node is idle/loaded.

7.3.5 Impact of Malleability on the System

In this section we study heterogeneous workloads where not all their jobs can be resized. For this purpose, we have designed two types of experiments using the previous 1,000-job workload. On the one hand, we set rates of flexible jobs, specifically 25%, 50% and 75%, which determine the proportion of flexible jobs in the workload. On the other hand, we generated workloads where only one application is malleable, in other words, workloads where only one type of job may be resized while the others remain fixed. All the workloads had two versions using both the rigid and the moldable submission.

Table 7.7 contains the resource allocation rate and the percentage of workload completion time, with respect to the fully fixed workload. The column colored in gray ("All") represents the reference values for the fully fixed workload and the fully flexible workload.

The most interesting cells have been highlighted in order to determine which kind of application has a greater impact in the results.

For the rigid submissions (third and forth rows), the heterogeneous workloads cannot beat the reference fixed workload resource allocation rate (96.37%). Nevertheless, workloads where CG or HPG-aligner is malleable show a quite similar allocation rate.

Regarding the completion time, we have coloured in green the 75%-flexible and the N-body-only workloads that run in about half of the reference time (53.77% and 56.39%, respectively). While the workloads with a 25-50-75% malleable jobs define a progression where the execution time is inversely proportional to the rate of malleable jobs, when only N-body is malleable the completion

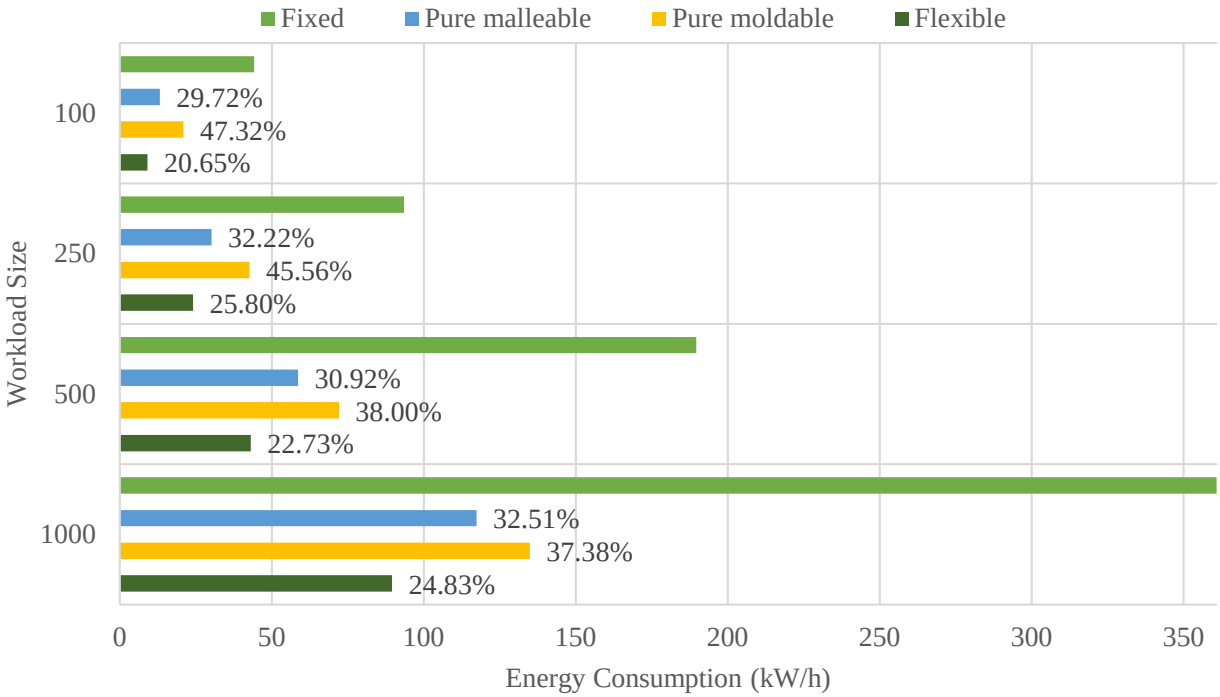


Figure 7.8: Energy needed to complete a workload compared to the fixed mode.

Table 7.7: Resource Allocation Rate and Completion Time of a 1,000-job Workload When Not All the Jobs are Flexible

		Flexible Jobs								
Submission	Percentage	None	25%	50%	75%	All	CG Only	Jacobi Only	N-body Only	HPG-aligner Only
Rigid	Res. Alloc.	96.37%	87.43%	87.07%	88.50%	87.29%	94.75%	88.34%	84.36%	93.06%
	Needed Time	100.00%	92.52%	73.49%	53.77%	36.12%	89.00%	105.67%	56.39%	101.77%
Moldable	Res. Alloc.	91.23%	89.92%	88.97%	86.92%	94.57%	95.51%	92.81%	90.62%	91.80%
	Needed Time	38.82%	37.29%	33.55%	30.09%	25.15%	25.51%	43.44%	33.88%	38.40%

time is almost reduced to half (still far from the target reference (36.12%)).

On the moldable submission case (fifth and sixth rows), resource allocation rates are pretty similar, since moldable jobs can leverage resources when they are launched. However, the workload comprising malleable CG jobs only achieves almost the same completion time (25.51%) as that attained by the flexible (malleable jobs submitted moldable) workload (25.15%).

From these results we conclude that there is no correlation between resource allocation and execution time. In addition, resources are not wasted because the percentages fluctuate inside the reference rates. We have also discovered in this study that rigid submissions profit more from poorly-scalable applications like our N-body. For the moldable submissions, the resource manager can leverage better applications highly scalable like our CG. This behavior is well exploited by our malleability solution.

7.4 Conclusions

In this chapter we have analyzed the state of the art of malleability concluding that its lack of popularity its due to its difficulty to be adopted by user codes. For this reason, we have presented a minimal MPI-based solution, the DMRlib, which intends to turn into malleable any code by setting reconfiguration points.

Using this library, we have developed four malleable applications presenting different scalability patterns. With these applications we have generated fixed, pure moldable, pure malleable and flexible workloads, in order to analyze the gains of malleability in terms of throughput, resource allocation and energy consumption. Furthermore, we have unearthed and improved the already existent *moldable submission* of jobs in order to study the benefits that, combined with malleability, it can bring to the workload execution.

Our studies have proven that it is not necessary to convert all the system applications into malleable; just adapting the right type of applications, the throughput can be dramatically boosted and the energy consumption reduced more than 75%.

Algorithm 7 Reconfiguration Procedure

```
1:  parentComm ← MPI_Comm_get_parent()
2:  if parentComm ≠ null then
3:      if commWorldSize > parentCommSize then
4:          recv_expand()
5:      else
6:          recv_shrink()
7:      end if
8:      MPI_Comm_disconnect(parentComm)
9:  else
10:     action ← DMR_Reconfiguration(&newComm)
11:     if action = expand then
12:         factor ← newComm_size/comm_size
13:         for i ← 1, factor do
14:             dstRank ← myRank × factor + i
15:             #pragma omp task
                    onto(newComm, dstRank)
16:             compute()
17:         end for
18:         send_expand()
19:         DMR_Detach()
20:     else
21:         if action = shrink then
22:             factor ← commSize/newCommSize
23:             dstRank ← myRank/factor
24:             #pragma omp task
                    onto(newComm, dstRank)
25:             compute()
26:             send_shrink()
27:             DMR_Detach()
28:         else
29:             pass                          ▷ No action has been scheduled.
30:         end if
31:     end if
32: end if
```

Algorithm 8 Slurm Reconfiguration Policy

```
1: if current < preferred then
2:     if avail_resources then
3:         action ← expand. return expand
4:     end if
5: else
6:     if pending_jobs then
7:         if current > preferred then
8:             if a job can be initiated then return shrink
9:                 action ← shrink.
10:            else
11:                if avail_resources then
12:                    action ← expand. return expand
13:                end if
14:            end if
15:            if avail_resources then return expand
16:                action ← expand.
17:            end if
18:        else
19:            if avail_resources then return expand
20:                action ← expand.
21:            end if
22:        end if
23:    end if
24: end if
```

Part IV

Discussion

CHAPTER 8

Conclusion

This chapter reviews the highlights of this PhD dissertation and discusses the conclusions.

8.1 Summary

This dissertation has addressed HTC from 2 different points of view: GPGPU virtualization and MPI malleability.

From the GPU perspective, we have analyzed in detail the adoption of a remote GPU virtualization technology in a workload manager. This study arises from the pre-doctoral stage where the tool was developed. Furthermore, we have exported the management of remote GPUs from cluster to cloud environments. For this reason, we have designed, deployed and analyzed a new shared service for cloud facilities (GSaaS) able to provide more flexibility to GPU-enabled infrastructures.

Regarding process malleability, after a thorough review and study of the current state-of-the-art we concluded with the design, implementation and commissioning of 2 new approaches for developing malleable applications. These present a different syntax with the aim of reaching the widest possible audience. On the one hand, following the OmpSs syntax we unveiled the DMR API, an API integrated in Nanos++ which handles MPI processes and data redistribution in a completely transparent fashion. On the other hand, we have presented DMRlib, a malleability library with an MPI friendly syntax. This approach targets developers more familiar to the MPI programming model, who are the norm in the field of HPC. Although the process management is still automatic, in this approach users are responsible for redistributing the data among processes. Nevertheless, we have included a set of tools in the library that assist users to implement those redistributions, being the most common patterns fully implemented in DMRlib.

In summary, in this dissertation we have stepped forward towards HPC productivity by providing detailed analysis and optimized utilities in order to increase the throughput of production facilities.

8.2 Conclusions

In accordance with current regulations from the Universitat Jaume I, at least both the abstract and conclusions of a PhD dissertation defended at this university have to be additionally included in Spanish.

English Version

From a high-throughput perspective, this PhD dissertation provides a deep insight into 2 different techniques for productivity in HPC facilities. Those techniques have been applied in various scenarios, demonstrating their usefulness and appropriateness for HTC.

Combining rCUDA and Slurm, we prove that we can boost the throughput of a GPU-enabled cluster by placing the accelerators in any location, depending on the system restrictions or user necessities. Specifically, we have experienced reductions of 48% in the workload completion time and of 40% in the energy consumption with respect to traditionally CUDA-based configurations. This more efficient management of the GPUs has also been translated into GPU utilization, doubling the baseline provided with native CUDA.

Similarly, we have experimented with rCUDA in cloud environments, concluding that public cloud solutions such as AWS are not ready for HPC at the moment when the experiments were performed. However, thanks to our approach, users may reduce their economic budget by customizing the number of GPUs they need for the VMs, thus skipping the restrictions of AWS when it comes to CUDA-enabled VMs. For this reason, we deployed GSaaS, a GPU-enabled private cloud infrastructure featuring a new OpenStack shared service responsible for managing and assigning cloudified GPUs to VMs. The main benefits attained by GSaaS are: dynamic scheduling of GPU resources, decoupling of the interface between the client and the rCUDA server, securing and isolating the access to the GPUs from the VMs, preventing GPU unauthorized accesses, and detaching VM traffic from GPU traffic by using a dedicated network. This new approach presents a series of new working modes for CUDA-enabled VMs that may be leveraged to increase the performance of GPU parallel applications, as well as to provide more flexibility regarding VM-GPU assignation.

Although malleability has already proven its advantages in HPC workloads, in this dissertation we have demonstrated that our solution not only can improve the cluster productivity, but also the coding productivity. For this purpose, we have designed a malleability solution that simplifies the development of malleable applications by providing 2 frameworks with syntaxes based on OmpSs or MPI, in order to reach a wider audience. Our solution has demonstrated to reduce the completion time of the jobs in a workload by reducing their waiting time in the queue. Concretely, our experiments have proven a reduction of 75% in the needed time to complete a workload when combining moldability and malleability. Dynamically reconfiguring jobs and reallocating resources, with either the DMR API or the DMRlib, have favored an increased utilization of the underlying resources, what has been translated in to an increase of the global throughput and remarkable reduction in the amount of energy needed to process the workload. Our studies have proven that it is not necessary to port all the system applications into malleable; just adapting the right type of applications, the throughput may be dramatically increased and the energy consumption reduced more than 75%.

Spanish Version

Desde la perspectiva de la computación de alta productividad (HTC), esta tesis doctoral proporciona una profunda comprensión sobre 2 técnicas para incrementar la productividad en el campo de la computación de altas prestaciones (HPC). Estas técnicas han sido aplicadas en distintos escenarios, donde se ha demostrado su utilidad y adecuación para HTC.

Combinando rCUDA y Slurm, hemos demostrado como incrementar drásticamente la productividad de un clúster cuyas GPUs se pueden ubicar en cualquier servidor, dependiendo de las restricciones del sistema o necesidades de los usuarios. Específicamente, en nuestros experimentos hemos reducido un 48% el tiempo total de procesamiento de una carga de trabajos y un 40% la energía consumida, respecto al mismo escenario utilizando CUDA nativo.

Del mismo modo, hemos utilizado rCUDA en entornos de *cloud computing*, para analizar su viabilidad. En lo referente a soluciones de nube pública, se experimentó sobre AWS, concluyendo que, en el momento de llevar a cabo las pruebas, AWS no estaba preparado para HPC. Sin embargo, gracias a nuestro enfoque, los usuarios de esta plataforma podrían reducir su gasto económico mediante la personalización del número de GPUs necesarias para sus máquinas virtuales (VMs), evitando así las restricciones que AWS presenta a la hora de contratar VM con GPUs. Por este motivo, se llevó a cabo el desarrollo de GSaaS, una infrastructura privada de computación en la nube que integra la administración y asignación de GPUs *cloudificadas* en VMs a través de un servicio de OpenStack. Las principales ventajas de este servicio son: planificación dinámica de GPUs, desacoplamiento de la interfaz entre el cliente y el servidor de rCUDA, ocultación de la ubicación real de las GPUs, prevención de accesos no autorizados y abstracción del tráfico entre la VM y la GPU a través de una red de interconexión dedicada. Con GSaaS se consiguen una serie de nuevos modos de trabajo en VMs habilitadas para la ejecución de aplicaciones CUDA, lo que puede incrementar su rendimiento y proporciona una mayor flexibilidad en la administración de GPUs en VMs.

Aunque la maleabilidad de trabajos ya haya demostrado sus ventajas en cargas de trabajos HPC, en esta disertación demostramos que nuestra solución no sólo puede mejorar la productividad del clúster, sino también la productividad del programador a la hora de escribir el código. Por este motivo, hemos diseñado una solución para la maleabilidad que facilita el desarrollo de aplicaciones maleables. Esta solución implementa 2 versiones con una sintaxis basada en OmpSs o MPI, para llegar a un mayor público y ha demostrado reducir el tiempo de procesamiento de los trabajos mediante la reducción de sus tiempos de espera en la cola. Concretamente, nuestros experimentos revelan una reducción del 75% en el tiempo necesario para completar una carga de trabajos cuando se combina moldabilidad y maleabilidad. La reconfiguración dinámica de trabajos y los cambios de reservas de recursos en tiempo de ejecución, con la DMR API o la DMRlib, plantean una mayor utilización de los recursos del clúster, lo que se traduce en un incremento de la productividad global y una notable reducción de la energía necesaria para el procesamiento de la carga de trabajos. Nuestros estudios han demostrado que no es necesario convertir todas las aplicaciones del sistema en maleables, si no que tan sólo tratando el tipo de aplicación correcto, la productividad se dispara y el consumo energético se puede reducir en más de un 75%.

CHAPTER 9

Further Work

This chapter describes the ongoing work at the moment of writing this manuscript along with several proposals for future work.

9.1 Ongoing Work

Thanks to all the work done in this thesis, which has supposed an interesting source of ideas, we have prepared projects for students and established collaborations with other researchers.

Through the *PRACE Summer of HPC 2018*[1] program, we mentor 2 students developing the following projects:

- **Dynamic management of resources simulator**[2]: In this project we propose the development of a simulator, which will simulate the execution of a workload composed of malleable and non-malleable jobs, over a parallel system. The malleable workload manager simulator, based on the principles of DMRlib, can extend the prospective of malleability from other approaches, such as resource heterogeneity, job priorities or power-awareness. With this simulator, we will set the jobs and tune the reconfiguration policies in order to determine the best configuration for meeting a given target.

- **Get more throughput, resize me! A case of study: LAMMPS malleable**[3]: This project is based on LAMMPS[4]. Since LAMMPS is a well-known HPC application, used in a wide range of scientific fields, we are interested in obtaining a reconfigurable version of it and analyzing its behavior when it is included in a workload. In this project, we aim to convert the MPI version of LAMMPS in malleable, using DMRlib.

Currently, we have 3 research collaborations:

- With the Leibniz Supercomputing Centre (LRZ), we are working on the modelling of a real workload through malleable synthetic applications that mimic the behavior of the original,

[1] `https://summerofhpc.prace-ri.eu`
[2] `https://summerofhpc.prace-ri.eu/dynamic-management-of-resources-simulator`
[3] `https://summerofhpc.prace-ri.eu/get-more-throughput-resize-me-a-case-of-study-lammps-malleable`
[4] `http://lammps.sandia.gov`

with the purpose of studying the effect of malleability in a production system. Once studied the impact, the administrators and the users could be advised for increasing the productivity.

- With the Predepartamental Unit of Medicine at UJI, we are studying how to increase their productivity and reduce their costs by performing part of their genetic alignment (using the malleable version of HPG-aligner, developed in this thesis) in our HPC facilities.

- With the Czestochowa University of Technology (PCZ), we are developing the malleable version of MPDATA [66]. MPDATA is the main module of a multiscale fluid model, which supposes an innovative solver in the field of numerical modeling of multiscale atmospheric flows. Furthermore, MPDATA is CUDA-capable and with this application we aim to experiment the effect of a malleable application in a GPGPU virtulized cluster, combining the 2 approaches for productivity studied in this dissertation.

9.2 Future Work

Apart from the ongoing work, we foresee the following future work.

Regarding GPGPU virtualization, with GSaaS we prpose to explore the usage and adaptation of the other components provided by the networking plane of the cloud infrastructure to provide access to scheduled cGPUs as future work. Furthermore, since OpenStack includes a dashboard project named "Horizon" that provides a user web interface to deploy the cloud infrastructure, we also have in mind to extend Horizon with the GSaaS functionality, letting users manage the GPUs in their VMs through the command line as well as the user web interface.

We have realized the potential of malleability, so in the future we plan to design smarter reconfiguration policies to consider wall times, priorities and power caps. With dynamic wall times a job could automatically adjust its requested time to the current process configuration. Together with a reconfiguration priority system, malleable jobs could obtain more priority to be expanded when the wall time is close to expire. We could also include information of the power necessities of the applications and the energy budget of the data center, in order to make more accurate reconfiguration decisions. These new approaches may be evaluated through the dynamic resources management simulator.

Other interesting approach to study may be to include the per-iteration information of each job in the runtime. With this information, jobs may be automatically reconfigured, since, in this case, the runtime would be the responsible for adjusting the job size depending on its performance.

Furthermore, it is well known that some applications are topology-aware. This topic was out of the scope of the thesis, but we understand that it may be considered in further studies.

In addition, we understand that the next natural step corresponds to the integration of the DMR API and the DMRlib in intranode malleability tools, such as the dynamic load balancing (DLB) [16]. DLB is a framework to improve the use of the computational resources of a computational node by solving imbalance problems. Currently, DLB can balance processes running on the same node, since DLB is based on shared memory and needs memory among all the processes sharing resources. That is why our malleability solution may be integrated and deployed together with that system. Apart from that, as it was stated in the DLB PhD thesis [16], the framework may benefit from the integration of the DMR API and the DMRlib in Slurm in several scenarios, for instance:

- A user wants to send a second application in the resources previously allocated to another application. Instead of requesting more resources the user may launch the analysis application in the same resources where the simulation was running.

- Slurm detects that a node is being underutilized and decides to reduce the number of resources assigned to the processes running on that node.

Bibliography

[1] Bilge Acun et al. "Parallel Programming with Migratable Objects: CHARM++ in Practice". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2014, pp. 647–658. ISBN: 978-1-4799-5500-8.

[2] Laleh Aghababaie Beni and Aparna Chandramowlishwaran. "PASCAL: A Parallel Algorithmic Scalable Framework for N-body Problems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10417 LNCS. Springer International Publishing, 2017, pp. 482–496. ISBN: 9783319642024.

[3] W. Michael Brown and Masako Yamada. "Implementing Molecular Dynamics on Hybrid High Performance Computers-Three-body Potentials". In: *Computer Physics Communications* 184.12 (Dec. 2013), pp. 2785–2793. ISSN: 00104655.

[5] CC Chang. "LIBSVM: a Library for Support Vector Machines". In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (2011).

[6] E. Chirivella-Perez et al. "Hybrid and Extensible Architecture for Cloud Infrastructure Deployment". In: *15th IEEE International Conference on Computer and Information Technology*. 2015, pp. 611–617.

[7] Isaías Comprés et al. "Infrastructure and API Extensions for Elastic Execution of MPI Applications". In: *Proceedings of the 23rd European MPI Users' Group Meeting on - EuroMPI 2016*. ACM Press, 2016, pp. 82–97. ISBN: 9781450342346.

[8] K. M. Diab, M. M. Rafique, and M. Hefeeda. "Dynamic Sharing of GPUs in Cloud Systems". In: *IEEE Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*. May 2013, pp. 947–954.

[9] José Duato et al. "An Efficient Implementation of {GPU} Virtualization in High Performance Clusters". In: *Euro-Par Workshops*. Vol. 6043. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 385–394.

[10] José Duato et al. "rCUDA: Reducing the Number of GPU-based Accelerators in High Performance Clusters". In: *Proceedings of the 2010 International Conference on High Performance Computing and Simulation, HPCS 2010*. IEEE, June 2010, pp. 224–231. ISBN: 9781424468287.

[11] K. El Maghraoui et al. "Dynamic Malleability in Iterative MPI Applications". In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. May 2007, pp. 591–598.

[12] K. El Maghraoui et al. "Malleable Iterative MPI Applications". In: *Concurrency and Computation: Practice and Experience* 21.3 (Mar. 2009), pp. 393–413.

[13] Kaoutar El Maghraoui, Boleslaw K Szymanski, and Carlos Varela. "An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments". In: *International Conference on Parallel Processing and Applied Mathematics*. 2006, pp. 258–27.

[14] Dror G. Feitelson. "Packing Schemes for Gang Scheduling". In: *Lecture Notes in Computer Science book series (LNCS, volume 1162)*. Springer, Berlin, Heidelberg, 1996, pp. 89–110.

[15] Dror G. Feitelson and Larry Rudolph. "Toward Convergence in Job Schedulers for Parallel Supercomputers". In: *Job Scheduling Strategies for Parallel Processing*. Vol. 1162/1996. 5. Springer Berlin Heidelberg, 1996, pp. 1–26. ISBN: 978-3-540-61864-5, 978-3-540-70710-3.

[16] Marta Garcia-Gasulla. "Dynamic Load Balancing for Hybrid Applications". PhD thesis. Universitat Politècnica de Catalunya, 2017, p. 218.

[17] Giulio Giunta et al. "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds". In: Springer, Berlin, Heidelberg, 2010, pp. 379–391.

[18] Gregory R. Grant et al. "Comparative Analysis of RNA-Seq Alignment Algorithms and the RNA-Seq Unified Mapper (RUM)". In: *Bioinformatics* 27.18 (Sept. 2011), pp. 2518–2528.

[19] Abhishek Gupta et al. "Towards Realizing the Potential of Malleable Jobs". In: *21st International Conference on High Performance Computing (HiPC)*. 2014.

[20] Vishakha Gupta et al. "GViM: GPU-accelerated virtual machines". In: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing - HPCVirt '09*. New York, New York, USA: ACM Press, 2009, pp. 17–24. ISBN: 9781605584652.

[21] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. "GPU Virtualization and Scheduling Methods". In: *ACM Computing Surveys* 50.3 (June 2017), pp. 1–37. ISSN: 03600300.

[22] Sergio Iserte. "Un Gestor de GPUs Remotas para Clusters HPC". MA thesis. Castelló de la Plana, Spain: Universitat Jaume I, 2014.

[36] Edmond Jajaga and Jolanda Kllobocishta. "MPI Parallel Implementation of Jacobi". In: *ICT Innovations*. 2012, pp. 449–458.

[37] T. J. Jun et al. "GPGPU Enabled HPC Cloud Platform Based on OpenStack". In: *The International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, 2014.

[38] L.V. Kalé and S. Krishnan. "CHARM++: a Portable Concurrent Object Oriented System Based on C++". In: *Proceedings of OOPSLA'93*. Ed. by A. Paepcke. ACM Press, Sept. 1993, pp. 91–108.

[39] Petr Klus et al. "BarraCUDA - a Fast Short Read Sequence Aligner Using Graphics Processing Units". In: *BMC Research Notes* 5.1 (Jan. 2012), p. 27. ISSN: 1756-0500.

[40] Stefan Kurtz et al. "Versatile and Open Software for Comparing Large Genomes." In: *Genome Biology* 5.2 (2004), R12. ISSN: 14656906.

[41] Pierre Lemarinier et al. "Architecting Malleable MPI Applications for Priority-driven Adaptive Scheduling". In: *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI)*. 2016, pp. 74–81. ISBN: 9781450342346.

[42] T. Y. Liang and Y. W. Chang. "GridCuda: A Grid-Enabled CUDA Programming Toolkit". In: *IEEE Workshops of International Conference on Advanced Information Networking and Applications*. Mar. 2011, pp. 141–146.

[43] Tyng-Yeu Liang and Yu-Wei Chang. "GridCuda: A Grid-Enabled CUDA Programming Toolkit". In: *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. IEEE, Mar. 2011, pp. 141–146. ISBN: 978-1-61284-829-7.

[44] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. "CUSHAW: a CUDA Compatible Short Read Aligner to Large Genomes Based on the Burrows-Wheeler Transform". In: *Bioinformatics* 28.14 (July 2012), pp. 1830–1837. ISSN: 1460-2059.

[45] Yongchao Liu et al. "CUDA-MEME: Accelerating Motif Discovery in Biological Sequences Using CUDA-enabled Graphics Processing Units". In: *Pattern Recognition Letters* 31.14 (Oct. 2010), pp. 2170–2177.

[46] Uri Lublin and Dror G. Feitelson. "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs". In: *Journal of Parallel and Distributed Computing* 63.11 (Nov. 2003), pp. 1105–1122. ISSN: 07437315.

[47] Gonzalo Martín et al. "Enhancing the Performance of Malleable MPI Applications by Using Performance-aware Dynamic Reconfiguration". In: *Parallel Computing* 46 (July 2015), pp. 60–77.

[48] Gonzalo Martín et al. "FLEX-MPI: an MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems". In: *Euro-Par Parallel Processing*. Aug. 2013, pp. 138–149. ISBN: 978-3-642-40046-9.

[49] Hector Martinez et al. "Scalable RNA Sequencing on Clusters of Multicore Processors". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015, pp. 190–195. ISBN: 978-1-4673-7952-6.

[50] Héctor Martínez et al. "A Dynamic Pipeline for RNA Sequencing on Multicore Processors". In: *Proceedings of the 20th European MPI Users' Group Meeting on - EuroMPI '13*. New York, New York, USA: ACM Press, 2013, p. 235. ISBN: 9781450319034.

[51] T. Mastelic et al. "Cloud Computing: Survey on Energy Efficiency". In: *ACM Comput. Surv.* 47.2 (2014), 33:1–33:36. ISSN: 0360-0300.

[52] I. Medina et al. "Highly Sensitive and Ultrafast Read Mapping for RNA-seq Analysis". In: *DNA Research* 23.2 (Apr. 2016), pp. 93–100. ISSN: 1340-2838.

[53] Alexander M. Merritt et al. "Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies". In: ().

[54] Adam Moody et al. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System". In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. Nov. 2010. ISBN: 978-1-4244-7557-5.

[55] NVIDIA. *CUDA C Programming Guide 7.5*. http://docs.nvidia.com/cuda/cuda-c-programming-guide. 2016.

[56] Minoru Oikawa et al. "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, Nov. 2012, pp. 1207–1214. ISBN: 978-0-7695-4956-9.

[57] Jitendra Padhye and Lawrence Dowdy. "Dynamic Versus Adaptive Processor Allocation Policies for Message Passing Parallel Computers: An Empirical Comparison". In: *Job Scheduling Strategies for Parallel Processing (IPPS)*. 1996, pp. 224–243. ISBN: 978-3-540-70710-3.

[58] Antonio J Peña. "Virtualization of Accelerators in High Performance Clusters". PhD thesis. Castellon, Spain: Universitat Jaume I, 2013.

[59] Antonio J Peña et al. "A Complete and Efficient CUDA-sharing Solution for HPC Clusters". In: *Parallel Computing* 40.10 (2014), pp. 574–588. ISSN: 0167-8191.

[60] James C. Phillips et al. "Scalable Molecular Dynamics with NAMD". In: *Journal of Computational Chemistry* 26.16 (Dec. 2005), pp. 1781–1802. ISSN: 0192-8651.

[61] Suraj Prabhakaran et al. "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2015, pp. 429–438. ISBN: 978-1-4799-8649-1.

[62] Sander Pronk et al. "GROMACS 4.5: a High-throughput and Highly Parallel Open Source Molecular Simulation Toolkit". In: *Bioinformatics* 29.7 (Apr. 2013), pp. 845–854. ISSN: 1460-2059.

[63] Zhengwei Qi et al. "VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming". In: *ACM Trans. Archit. Code Optim.* 11.2 (2014), 17:1–17:25. ISSN: 1544-3566.

[64] C. Reaño et al. "Influence of InfiniBand FDR on the performance of remote GPU virtualization". In: *IEEE International Conference on Cluster Computing*. Sept. 2013, pp. 1–8.

[65] Felipe S. Ribeiro et al. "Autonomic Malleability in Iterative MPI Applications". In: *Symposium on Computer Architecture and High Performance Computing*. 2013, pp. 192–199. ISBN: 9781479929276.

[66] Bogdan Rosa et al. "Adaptation of Multidimensional Positive Definite Advection Transport Algorithm to Modern High-Performance Computing Platforms". In: *International Journal of Modeling and Optimization* 5.3 (June 2015), pp. 171–176. ISSN: 20103697.

[67] Florentino Sainz et al. "Collective Offload for Heterogeneous Clusters". In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2015, pp. 376–385. ISBN: 978-1-4673-8488-9.

[68] Osman Sarood et al. "Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2014, pp. 807–818. ISBN: 978-1-4799-5500-8.

[69] Lin Shi et al. "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines". In: *IEEE Transactions on Computers* 61.6 (June 2012), pp. 804–816. ISSN: 0018-9340.

[73] R. Sudarsan and C.J. Ribbens. "Scheduling Resizable Parallel Applications". In: *International Symposium on Parallel & Distributed Processing*. IEEE, May 2009.

[74] Rajesh Sudarsan and Calvin J. Ribbens. "Combining Performance and Priority for Scheduling Resizable Parallel Applications". In: *Journal of Parallel and Distributed Computing* 87 (2016), pp. 55–66.

[75] Rajesh Sudarsan and Calvin J. Ribbens. "ReSHAPE: a Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment". In: *Proceedings of the International Conference on Parallel Processing*. 2007. ISBN: 076952933X.

[76] Rajesh Sudarsan, Calvin J. Ribbens, and Diana Farkas. "Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS". In: *International Conference on Computational Science (ICCS)*. 2009, pp. 175–184.

[77] Panagiotis D Vouzis and Nikolaos V Sahinidis. "GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment." In: *Bioinformatics (Oxford, England)* 27.2 (Jan. 2011), pp. 182–8. ISSN: 1367-4811.

[78] J. P. Walters et al. "GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications". In: *Cloud Computing (CLOUD), IEEE 7th International Conference on*. IEEE. 2014, pp. 636–643.

[79] Gengbin Zheng, Xiang Ni, and Laxmikant V. Kale. "A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale". In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, June 2012, pp. 1–6. ISBN: 978-1-4673-2266-9.

APPENDIX A

Publications

During this dissertation the following manuscripts were published in (or submitted to) research journals and conferences:

A.1 Journals

- Sergio Iserte, Héctor Martínez, Sergio Barrachina, Castillo Maribel, Rafael Mayo, and Antonio J. Peña. "Dynamic Reconfiguration of Non-iterative Scientific Applications: A Case Study with HPG-aligner". In: *International Journal of High Computing Performance Application (IJHPCA)* (Sept. 2018)

- Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, and Antonio J. Peña. "DMR API: Improving the Cluster Productivity by Turning Applications into Malleable". In: *Parallel Computing* (July 2018). ISSN: 0167-8191

- Sergio Iserte, Raúl Peña-Ortiz, Juan Gutiérrez-Aguado, Jose M. Claver, and Rafael Mayo. "GSaaS: A Service to Cloudify and Schedule GPUs". In: *IEEE Access* (July 2018). ISSN: 2169-3536

- Federico Silla, Sergio Iserte, Carlos Reaño, and Javier Prades. "On the benefits of the remote GPU virtualization mechanism: The rCUDA case". In: *Concurrency and Computation: Practice and Experience (CCPE)* (2017), e4072. ISSN: 1532-0626

A.2 International Conferences

- Sergio Iserte, Javier Prades, Carlos Reano, and Federico Silla. "Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm". In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, May 2016, pp. 98–101. ISBN: 978-1-5090-2453-7

- Sergio Iserte, Francisco J. Clemente-Castelló, Adrián Castelló, Rafael Mayo, and Enrique S. Quintana-Ortí. "Enabling GPU Virtualization in Cloud Environments". In: *6th International*

Conference on Cloud Computing and Services Science (CLOSER). SCITEPRESS - Science, 2016, pp. 249–256. ISBN: 978-989-758-182-3

A.3 International Workshops

- Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, and Antonio J. Peña. "Efficient Scalable Computing through Flexible Applications and Adaptive Workloads". In: *46th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, Aug. 2017, pp. 180–189. ISBN: 978-1-5386-1044-2
- Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reano. "Remote GPU Virtualization: Is It Useful?" In: *2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. IEEE, Mar. 2016, pp. 41–48. ISBN: 978-1-5090-2121-5

A.4 Doctoral Symposiums

- Sergio Iserte, Antonio J. Peña, and Rafael Mayo. "Productivity-enhancing Malleability for HPC Applications". In: *The 27th International Conference on Parallel Architectures and Compilation Techniques (PACT18), ACM Student Research Competition (SRC)*. Limasol (Cyprus), Nov. 2018. ISBN: pending
- Sergio Iserte, Antonio J. Peña, Rafael Mayo, Enrique S. Quintana-Ortí, and Vicenç Beltran. "Dynamic Management of Resource Allocation for OmpSs Jobs". In: *1st PhD Symposium on Sustainable Ultrascale Computing Systems (NESUS PhD)*.. Timisoara (Romania), 2016, pp. 55–58. ISBN: 978-84-608-6309-0

A.5 Education

- Adrián Castelló, Sergio Iserte, and José A. Belloch. "Accessible C-programming Course from Scratch Using a MOOC Platform without Limitation". In: *4th International Conference on Higher Education Advances (HEAD)*. Valencia (Spain), June 2018. ISBN: 978-84-9048-690-0

A.6 National Conferences

- Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Vicenç Beltran, and Antonio J. Peña. "El camino desde la maleabilidad MPI hasta las cargas de trabajos adapativas". In: *XXVIII Jornadas de Paralelismo (JP)*. Málaga (Spain), 2017
- Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, Javier Prades, Carlos Reaño, Federico Silla, and Jose Duato. "Comparativa de políticas de selección de GPUs remotas en clusters HPC". in: *XXVI Jornadas de Paralelismo (JP)*. Córdoba (Spain), 2015

A.7 Posters

- Sergio Iserte, Héctor Martínez, Sergio Barrachina, Castillo Maribel, Rafael Mayo, Enrique S. Quintana, and Antonio J. Peña. "MPI Malleability Integration into a Bioinformatics Tool". In: *The EuroMPI Conference*. Barcelona, (Spain), Sept. 2018

- Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, and Antonio J. Peña. "High-throughput computation through MPI Malleability". In: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggi (Italy), July 2018
- Federico Silla, Carlos Reaño, Javier Prades, and Sergio Iserte. "Benefits of remote GPU virtualization: the rCUDA perspective". In: *GPU Technology Conference (GTC)*. Silicon Valley, California (USA), 2016
- Sergio Iserte, Francisco J. Clemente-Castelló, Rafael Mayo, and Enrique S. Quintana-Ortí. "GPU Virtualization in the Cloud". In: *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggi (Italy), July 2015

APPENDIX B

DMRlib Reproducibility Artifact

DMRlib has been deployed in several HPC infrastructures, that is why since the very beginning of its deployment we tailored a reproducibility artifact in order to be able to reproduce our experiments in other facilities.

In this appendix we explain how to install, compile and configure the experimental configuration deployed in Section 7.3 of Part III, with the purpose of reproducing the results in terms of throughput.

B.1 Description

B.1.1 Check-list (Artifact Meta Information)

- **Program:** MPI, OpenMP, OmpSs, C, C++.
- **Compilation:** gcc, g++, mpicc, mpic++, mpimcc.
- **Binary:** One binary for each malleable application with the extension .INTEL64.
- **Data set:** Workload generated using the Feitelson Model.
- **Run-time environment:** Linux environment with GCC and MPI.
- **Hardware:** Any multi-node infrastructure.
- **Output:** Slurm database and log of executed jobs.
- **Experiment workflow:** Initialize Slurm in every node and submit all the jobs in the workload. Once terminated, gather and process the data generated during the workload execution.
- **Experiment customization:** Apart from the number of jobs that compose the workload, the job inter-arrival time may be incremented or decreased.
- **Publicly available?:** Yes

B.1.2 How Software Can Be Obtained

Following we include the URLs of the software repositories utilized in this work.

- Mercurium Offload (OmpSs Compiler)
  `https://siserte@bitbucket.org/ompssmalleability/mcxx-malleable.git`
- Nanos++ Offload Detach (OmpSs Runtime)
  `https://siserte@bitbucket.org/ompssmalleability/nanox-offload_detach.git`
- Slurm with Malleability Support (Resource Manager System)
  `https://siserte@bitbucket.org/ompssmalleability//slurm-malleable.git`
- DMRlib (Malleability Library)
  `https://siserte@bitbucket.org/ompssmalleability/dmrlib.git`
- Conjugate Gradient Malleable (Application)
  `https://siserte@bitbucket.org/ompssmalleability/cg-malleable.git`
- Jacobi Malleable (Application)
  `https://siserte@bitbucket.org/ompssmalleability/jacobi-malleable.git`
- N-body Malleable (Application)
  `https://siserte@bitbucket.org/ompssmalleability/nbody-malleable.git`
- HPG-Aligner Malleable (Application)
  `https://siserte@bitbucket.org/ompssmalleability/hpg-aligner-malleable.git`
- Feitelson Parallel Workload Generator
  `http://www.cs.huji.ac.il/labs/parallel/workload/m_feitelson96/m_feitelson96.c`

B.1.3 Software Dependencies

In general, we used a Linux environment with GCC 5.3.0 and MPICH 3.2.0 to compile and run the tools and applications. Mercurium needed GPerf 3.0.4 and GPerfTools 0.8; and Conjugate Gradient was compiled with Intel MKL and OpenBLAS 0.2.19.

B.1.4 Datasets

The average inter-arrival time (determined by the Feitelson model) was configured with a factor of 1. This represents a highly loaded scenario where jobs are massively submitted while fitting the arrival Poisson distribution of the model. The maximum job size was set to 32, assuming that a job should not request more than a quarter of the total amount of resources (128-node cluster).

B.2 Installation

Regarding the installation, we have to start with configuring our Slurm version providing the installation and configuration paths:

```
./autogen; ./configure --prefix=<slurm_dir> --confdir=<slurm_conf>; make;
make install
```

With the libraries of Slurm installed, we configure the OmpSs runtime:

```
./bootstrap; ./configure --without-opencl --prefix=<ompss_dir>
--with-slurm=<slurm_dir> --with-mpi=<mpi_dir>; make; make install
```

Finally, we install the OmpSs compiler, Mercurium, in order to have OmpSs fully operative:

```
./configure --enable-ompss --prefix=<ompss_dir> --with-nanox=<ompss_dir>
--with-mpi=<mpi_dir>; make; make install
```

For DMRlib, as well as, the malleable applications, we only have to define correctly the paths for OmpSs, MPI, etc. and then generate the libraries/binaries with `make`. Notice that malleability parameters have to be configured for each application before compiling.

Once the software is ready, the next step is to configure and initiate Slurm. For this purpose, the user has to define the appropriate paths and nodes in the `<slurm_conf>/slurm.conf` file. In order to enable malleability, we must choose the following plugins:

```
SchedulerType=sched/backfill
SelectType=select/linear
PriorityType=priority/multifactor
```

After that, Slurm daemons have to be normally started in each node.

B.3 Experiment Workflow

The experiment workflow follows these steps:

1. Prepare a launch script for each application and each mode (non-malleable and malleable). For example, for our applications we have written the following scripts:

   - Conjugate Gradient:

   ```
   export NX_ARGS="--enable-block --force-tie-master"
   export DMR_SCHED_PERIOD=10
   NODELIST="$(scontrol show hostname $SLURM_JOB_NODELIST |
    paste -d, -s)"
   mpiexec -n $SLURM_JOB_NUM_NODES -hosts $NODELIST <cg_dir>/
    cg.INTEL64

   ```

   - Jacobi:

   ```
   export NX_ARGS="--enable-block --force-tie-master"
   export DMR_SCHED_PERIOD=10
   NODELIST="$(scontrol show hostname $SLURM_JOB_NODELIST |
    paste -d, -s)"
   mpiexec -n $SLURM_JOB_NUM_NODES -hosts $NODELIST <
    jacobi_dir>/jacobi.INTEL64

   ```

   - N-body:

```
NODELIST="$(scontrol show hostname $SLURM_JOB_NODELIST |
 paste -d, -s)"
mpiexec -n $SLURM_JOB_NUM_NODES -hosts $NODELIST <nbody_dir
 >/n-body.INTEL64 6553600 50

```

- HPG-Aligner:

```
export NX_ARGS="--enable-block --force-tie-master"
NODELIST="$(scontrol show hostname $SLURM_JOB_NODELIST |
 paste -d, -s)"
mpiexec -n $SLURM_JOB_NUM_NODES -hosts $NODELIST <hpg_dir>/
 hpg-aligner.INTEL64 rna -i <hpg_dir>/bwt_index -f <hpg_dir
 >/reads40M100nt.fq -o testDir$SLURM_JOB_ID -s $(((
 $SLURM_JOB_NUM_NODES-2)*4))

```

2. Generate a workload assigning randomly one of the four applications to each job and the inter-arrival time defined by the model. From this workload we have to elaborate 4 versions: the fixed, pure moldable, pure malleable and flexible.

3. Execute each workload.

B.4 Evaluation and Expected Result

In order to evaluate the performance of each workload we obtain the Slurm database entries with `sacct` and the job's log data. From there, we can get the job waiting, execution and completion time, and the number of allocated nodes in each moment.

The expected result is a reduction in the workload completion time inasmuch as the workloads have more freedom to define the number of nodes of their jobs.

B.5 Experiment customization

Malleable applications may be configured with different problem sizes. Or even we can also develop new malleable applications or synthetic codes that mimic the behavior of existent scientific applications.