UNIVERSITY OF PAVIA

FACULTY OF ENGINEERING
Department of Electrical, Computer
and Biological Engineering

Ph.D. in Electronics, Computer Science
and Electrical Engineering

HPC and Cloud Computing

Candidate
Luigi SANTANGELO
XXXII Cycle

Advisor
Prof. Marco FERRETTI

Academic Year 2018/2019

Abstract

L'utilizzo di infrastrutture cloud, quale alternativa ai tradizionali sistemi on-premise, è diventato nel corso degli anni una interessante opportunità per eseguire applicazioni commerciali. Le organizzazioni aziendali infatti possono beneficiare dei numerosi vantaggi offerti dalle piattaforme cloud, e questo si traduce in una significativa riduzione dei costi e in una maggiore efficienza nella gestione dei sistemi informativi. Tuttavia, tale affermazione potrebbe non essere vera per le applicazioni scientifiche, storicamente progettate per essere eseguite su sistemi HPC ad altissime prestazioni.

Un confronto tra sistemi cloud e infrastrutture ad alte prestazioni non sempre vede le prime vincenti su queste ultime, soprattutto quando solo le prestazioni e l'aspetto economico vengono presi in considerazione. Questa affermazione è confermata dai risultati dei test sperimentali, descritti in questa tesi, ottenuti dopo aver migrato sul cloud due applicazioni scientifiche (chiamate rispettivamente Cross Motif Search e BloodFlow), basate su due differenti modelli di comunicazione. Test di prestazioni e scalabilità ottenuti eseguendo le due applicazioni su due infrastrutture simili (cloud e HPC) hanno mostrato che le due applicazioni si comportano meglio quando sono eseguite sul sistema HPC, soprattutto per BloodFlow, la cui esecuzione è fortemente condizionata dall'infrastruttura di rete, essendo basata su un modello di comunicazione molto intensivo.

Anche sotto l'aspetto economico, il cloud risulta essere non conveniente se confrontato con il costo di un sistema HPC. Questo risultato è stato ottenuto confrontando il costo per eseguire una applicazione generica per un'ora su tre differenti architetture cloud (Google, Amazon, Microsoft) con il costo necessario per eseguire la medesima applicazione per il medesimo tempo su un sistema HPC (nello specifico su Marconi). Sebbene la piattaforma cloud di Google risulta essere più economica rispetto alle altre due alternative, il costo rimane diverse volte più alto del costo di Marconi, anche se si utilizza un cluster cloud con "prerilasciabilità", ovvero un cluster di istanze la cui esecuzione può essere interrotta automaticamente dall'infrastruttura cloud, senza alcun preavviso, quando altri task richiedono l'accesso a tali risorse. In definitiva, anche sotto l'aspetto economico, il cloud computing sembra non essere conveniente per l'esecuzione di applicazioni scientifiche.

Tuttavia un confronto tra sistemi cloud e HPC che prenda in esame solo l'aspetto economico e di performance è iniquo, perché ci sono moltissimi altri fattori che rendono il cloud vincente sui sistemi HPC. Infatti, quando ad esempio viene preso in esame il tempo di consegna e la preferenza dell'utente, il panorama sulla convenienza cambia. Un equo confronto tra sistemi cloud e HPC dovrebbe quindi prendere in esame non solo le prestazioni e il costo, ma anche il tempo di attesa dei job, il numero dei job interrotti dal sistema, il tempo di setup, il tempo di disponibilità del sistema e la preferenza dell'utente. Ad esempio, i job sottomessi in un sistema HPC non vengono eseguiti immediatamente ma vengono inseriti in una coda in attesa che le risorse richieste dal job siano disponibili. Il tempo di attesa di un job nella coda varia in base a diversi fattori, quali il numero delle risorse fisiche disponibili, il numero di job già sottomessi ma in attesa di essere eseguiti, e, qualche volta, anche dal

numero di job già eseguiti dall'utente nell'ultimo periodo di tempo o da fattori simili. Il tempo in cui i risultati saranno disponibili dipende quindi non solo dal tempo impiegato dal job per completare la sua esecuzione ma anche dal tempo in cui il job è rimasto in attesa nella coda. Un utente sensibile al tempo potrebbe quindi essere disposto a pagare di più per avere risultati velocemente e quindi l'architettura da scegliere potrebbe essere differente in base alle sue preferenze. La scelta della migliore architettura quindi può essere vista come un problema decisionale basato su molteplici attributi. Un opportuno modello, basato su tali attributi, potrebbe aiutare i ricercatori a capire quale architettura può essere la migliore in base alla loro preferenza, il tempo di esecuzione del job, il costo per la sua elaborazione e il tempo di attesa nella coda. L'architettura selezionata quindi potrebbe non essere la più performante né la più economica, ma quella che massimizza la funzione di utilità che descrive il modello.

Dopo aver introdotto il cloud computing e aver esaminato costi, benefici e compromessi derivanti dall'utilizzo dei sistemi cloud, questa tesi confronta le prestazioni di due differenti applicazioni scientifiche eseguite su un sistema HPC e su un cluster di macchine virtuali, entrambi aventi una configurazione simile, nonchè i costi affrontati per eseguire le due applicazioni sulle due differenti architetture. I risultati mostrano che entrambe le applicazioni target si comportano meglio sull'architettura HPC, dove anche i costi risultano essere inferiori, e ciò si mantiene vero anche ottimizzando il software attraverso opportune attività di profilazione. Viene inoltre mostrato come, attraverso la combinazione di parametri applicativi e di rete, sia possibile costruire un modello analitico per la predizione del tempo di comunicazione, ma che comunque non tiene conto della contesa delle risorse e della congestione di rete, rendendo di fatto il tempo stimato sensibilmente inferiore rispetto al tempo speso dalle applicazioni all'interno delle funzioni MPI. Il modello analitico resta tuttavia indicativo della complessità del modello di comunicazione. Infine viene definita una funzione utilità basata sulla funzione di aggregazione geometrica pesata, che prende in considerazione la preferenza dell'utente, il tempo di esecuzione, il costo di esecuzione, il tempo di attesa stimato e il tempo di avvio di un cluster di macchine virtuali. Tale modello è stato utilizzato per valutare per quali esecuzioni delle due applicazioni in esame il cloud risulta essere più conveniente rispetto al sistema HPC. Per rispondere a tale domanda, è stato necessario ottenere una stima del tempo medio di attesa dei job sottomessi su Marconi, nonché il tempo di avvio di un cluster di macchine virtuali. I risultati dimostrano che il modello è robusto ed esiste inoltre un significativo numero di esecuzioni di entrambe le applicazioni per le quali l'infrastruttura cloud sembra essere l'ambiente migliore per l'esecuzione di applicazioni scientifiche.

Introduction

Up to a few years ago, scientific applications were designed, developed and built to be executed just on high performance infrastructures or on-premise systems, which were, and are, able to guarantee high performances reducing execution time and increasing application scalability. Since 2006, however, the traditional IT landscape has started to change. The birth of Cloud Computing brought along several opportunities for business users who started replacing their own on premise systems with the emerging cloud services. As time goes by, cloud infrastructures became soon more powerful, reliable, affordable and secure, conquering the interest of the scientific community. Despite the considerable amount of advantages which can be met relying on cloud computing, there are a lot of barriers which slow down the transition towards the emerging environment. One of the most relevant factors limiting the transition is the complexity of moving applications. Indeed, moving an application into the cloud is not effortless and might take a huge amount of time. Therefore, before starting to move an application into the cloud, it might be worth having in advance an idea about how the application will behave when run in a different infrastructure. It is then crucial, for researchers, to understand trade-offs, costs and benefits related to moving an application into the cloud. Indeed, applications which are able to run quickly on HPC systems, might perform worse when run on a different infrastructure, such as in the cloud. There are several factors limiting the performance of an application running in the cloud, such as the virtualized environment, the architecture of the physical CPU, the amount of RAM, and so on, but, as highlighted in many works, perhaps the most important factor limiting the performance of a large amount of scientific applications is the interconnection network. For communication-intensive applications, the network infrastructure may soon become a bottleneck, reducing performance and scalability.

A rich set of different interconnection technologies (such as Infiniband, Intel Omni Path or Ethernet) have been developed to reduce the overhead introduced by the network layer and increase application scalability and performance. All these technologies are currently used in HPC systems. Indeed, according to the Top500 list (Top500 Release June 2018), 49.4% of the HPC systems uses Gigabit Ethernet technology to interconnect nodes, 27.8% uses Infiniband, 7.8% uses Omni Path and the remaining 15% makes use of proprietary or custom interconnection networks. Cloud service providers try to keep in step with the HPC systems by studying and introducing new components in their interconnection model, making the cloud environment more attractive and promising even for running scientific applications, traditionally executed on HPC systems [1–6]. This is also the case for many bioinformatics applications, and bio-scientists are thinking to move their parallel code to the cloud infrastructures. As this activity is not effortless and can take a huge amount of time, before moving an application to the new infrastructure, a deep analysis of the application and the new infrastructure layer should be done, in order to get insights about how the application will behave being run in the cloud and how it might be adapted in order to reduce the impact of the communication. It might be useful to know in advance the impact of the application communication, because this might

help researchers to get hints and insights about how the application will perform on a different architecture. Parameters describing the application, the underlying network infrastructure and the application communication model can be used to build an analytical model which is able to give a perspective about how much better or worse the application might behave being run on such infrastructure. But performance is not the unique factor to keep into account during the infrastructure comparison. Indeed, even though a system performs worse than another, it might be convenient to use it according to a different perspective, such as the economical aspect or something else. So, a fair comparison between different infrastructures, such as cloud and HPC, should take into account several factors, and not just the performance one. Building a proper utility function might help researchers to understand which platform is the best for running a scientific application. The selected infrastructure might not be the most performing but that one which maximize the utility function.

This thesis focuses on the assessment of the cloud infrastructure both from a performance and an economical perspective, to understand whether or not it could be considered a feasible place for running scientific applications. We start describing a methodology for predicting the impact of the communication. The key element of our methodology is an analytical model which is based on application-related and network-related parameters. The former parameters are gathered using profiling and tracing tools such as Intel VTune, the latters are instead gathered using a tool based on the pLogP model. Profiling and tracing activities are also useful to spot and remove bottlenecks limiting the application performance. An analytical model describing the application communication model and network infrastructure can be built by combining both application-related and network-related parameters. The analytical model describes how the time spent in communication changes when the concurrent MPI process number grows up and then can be used to get hints and insights about how the application will perform being run on that infrastructure. To validate our methodology we used two different applications, named respectively Cross Motif Search and BloodFlow, based on different communication patterns. We tried to predict the application behaviour on two different architectures, a real HPC system, named Marconi, and a cloud infrastructure provided by Google. Before moving and running the applications on both architectures, the related analytical model has been built. To validate our model, the applications have been executed on both architectures and then the predicted behaviour has been compared to the observed one. Results show that applications suffer more when executed in the cloud and the lessen in performance is due to the increase in the time spent by both applications inside the MPI functions, and this decline in performance is higher when the communication among processes gets higher. Although our methodology is not able to predict the time spent by the application inside the MPI functions, there is a good correlation between the predicted and the observed behaviour. Results also show that CPU-intensive applications or based on the master/worker communication model can be successfully executed in the cloud because the overhead introduced by the virtual infrastructure can be considered negligible. For such applications, their behaviour is comparable to those observed in a real HPC system. On the other hand, communication-intensive applications might perform and scale badly because of the overhead introduced by the network infrastructure layer. Thus cloud environment is still not suitable for running applications based on more complex communication pattern, making use of a huge amount of collective operations. Although cloud infrastructure showed a lack in performance for some applications, it might be worth from the economical perspective. We compared the cost a researcher

should afford for running an application on Marconi and on three different cloud infrastructures provided respectively by Google, Amazon and Microsoft. Results show that the cloud infrastructure is not convenient at all for running scientific application from the economical point of view either. This is true if just performance and cost are taken into account, but when the turnaround time comes into account, the conclusion might be slightly different. As known, indeed, in all HPC systems, each job before being run is put in a queue until the requiring resources (such as memory and cpus) are available. The time spent waiting in the queue depends on several factors, such as the number of jobs which are submitted before and not already completed, the amount of available resources and the amount of resources required by each job. If the waiting time in the HPC system is too high, using cloud solutions might improve the turnaround time, making the cloud infrastructure more appealing for those users who need results as fast as possible. We decided to go further on our analysis in order to understand whether or not cloud infrastructure can be considered as an interesting place when the aim is the optimization of the turnaround time. Building an evaluation model might help researchers to understand which platform might be the best depending on the user's preference, the execution time, the cost for computing and the expected waiting time in the queue. To build such a model, a characterization of the workload of a real HPC system needs to be done in order to understand the job waiting time depending on the job geometry (job size, amount of memory, maximum runtime), job failure, setup time and maintenance time. Many previous works [7–14] have already characterized the workload of the HPC systems, but most of them aimed to evaluate the resource utilization and improve the scheduling algorithms to get the highest system utilization possible. Many others instead tried to predict the waiting time using machine learning techniques [15–18]. In our work, instead, we characterize the workload of Marconi in order to assess the job waiting time. Such time is then introduced in a utility function which is used to evaluate the best infrastructure (between Marconi and Google Cloud) for running both target applications (Cross Motif Search and BloodFlow).

This thesis is structured as follows: chapter 1 gives an overview about cloud computing, describing it from different perspectives and all the factors making the cloud appealing. The chapter also gives a wide list of advantages and disadvantages users can run into working with the cloud infrastructure; chapter 2 describes the HPC system and the virtual cluster built on the cloud infrastructure and used in all the experiments described later, and assess the network performance of both architectures using the pLogP model, which is deeply described in Appendix B; chapter 3 describes Cross Motif Search and BloodFlow, the two main applications used for our tests that have been moved to the cloud infrastructure; chapter 4 gives the results we got tracing and profiling the applications using several tools. The chapter also highlights the bottlenecks limiting the application performance; chapter 5 describes all the activities we did in order to remove bottlenecks and improve the application performance before moving them to the cloud; chapter 6 summarizes performance and scalability of both applications running on the native HPC infrastructure; after describing application-related parameters and network-related parameters, chapter 7 shows how to use those parameters to build the analytical model which can be used to get insight about how the applications might perform being run on a target infrastructure; chapter 8 shows the results we got after moving and running the applications in the cloud and compares the predicted application behaviour with the real one; as described in chapter 9, although applications based on a simple communication pattern might perform well even in the cloud, HPC system proves to be the winner, not just from the performance perspective but also even from the

economical perspective. The chapter indeed compares the cost for running the same application on three different cloud infrastructures and on an HPC system; the chapter then highlights the importance of using other factors for evaluating the goodness of the cloud infrastructure, such as the turnaround time and the user expectancy. The chapter also provides a utility function based on the weighted geometric aggregation function where the attributes taken into account by the formula are the user preference, the execution time, the cost for computing, the expected waiting time in the queue for HPC system and the virtual instance startup time for the cloud; chapter 10 gives a wide characterization of the workload submitted on Marconi in nine months. Although, for our purpose, just the waiting time characterization is needed, the chapter describes the submitted jobs from different perspectives and also makes a clusterization of the jobs using the k-means method; chapter 11 studies the virtual instance startup time, which is the time spent by a virtual instance, running on the cloud infrastructure, to get ready to execute jobs; chapter 12 builds the evaluation function using the waiting time and the virtual instance startup time, and then applies the function to evaluate all runs of both applications. In chapter 13 we draw the conclusions of our research.

Contents

List of Figures

List of Tables

Chapter 1

Cloud Computing

1.1 The birth of the Cloud

The term *Cloud Computing* was first used on 2006, August 24th, when Amazon, one of the most famous online trading company, announced, with a post on his website [19], a new service called Elastic Compute Cloud that was able to provide resizeable computing capacity in the cloud. The birth of this service opened a new paradigm in computer science, allowing users to obtain and configure computation and storing capacity in an automatic way, without any (or at least with a small) human interaction, and paying just for what was really used by consumers. Since then, customers all over the world started to exploit the Amazon's Infrastructure for running their own applications such as simulation and web hosting. At same time, many other companies started offering cloud services and the request of cloud computing resources grew more and more. Although the term was coined in 2006, it was officially defined only five year later.

1.2 What is cloud computing?

To describe Cloud Computing, we will use the official definition provided by the National Institute of Standards and Technology (NIST), the U.S. government entity that formally describes standards. According to its definition, released in September 2011, [20], Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

The NIST definition lists five essential characteristics of cloud computing: on-demand self-service, broad network access, resource pooling, rapid elasticity or expansion and measured service. It also lists three service models (software, platform and infrastructure) and four deployment models (private, community, public and hybrid) that together categorize ways to deliver cloud services. To better understand Cloud Computing, all characteristics, service and deployment models will be described. A good description about cloud computing, issues and challenges can be found in [21].

1.3 Essential characteristics of cloud computing

According to the National Institute of Standards and Technology definition, Cloud Computing is characterized by five important factors:

1. **On-demand Self Service**: a consumer, having a specific need, can get configurable resources, such as CPU time, network, storage, software, in an automatic way, without any human interaction with the provider of these resources;

2. **Broad Network Access**: all resources are delivered over the network and consumers can get access using the preferred platform, such as computers, laptops, PDAs, mobile phones;

3. **Resource Pooling**: all resources are managed in a shared pool and are delivered to the consumers depending on their needs. The physical resources become invisible but at the same time they appear to be infinite from the consumer perspective. Users do not have either control or knowledge about where their data are stored or computed. The physical resources can be shared among different consumers;

4. **Rapid Elasticity**: consumers can scale-up and down their cloud infrastructure whenever they want. This approach reduces the upfront costs. Indeed, consumers can start with a small resource configuration and scale up according to the business growth, increasing hardware resources only when there is an increase in their needs;

5. **Measured Service**: Cloud computing is based on the pay-per-use model. Consumers pay just for what they really use (for example processors per hours or storage per days). Providers make use of an appropriate mechanism to measure resource usage for each consumer.

1.4 The service models of Cloud Computing

In its original definition, from the service model perspective, Cloud Computing was classified in just three classes, but many others have been added lately although the official definition has not been modified yet. According to the original definition, the three service models are:

1. **Software as a Service (SaaS)**: consumers can just use applications that are deployed on a hosting environment. The applications can be accessed through the network using different clients, such as browsers, PDAs, and so on. Consumers do not have any control on the infrastructure layer, such as network, server, operating system, storage. Sometimes, just small application configuration settings can be modified. Examples of Software as a Services include Google Gmail and Google Docs;

2. **Platform as a Service (PaaS)**: consumers can use a cloud platform where applications can be developed and deployed. The platform supports the entire life-cycle and provides programming languages, libraries, services and tools which can be used by developers. No control on the underlying infrastructure (network, server, operating system, storage) is allowed, just a few application-hosting configuration can be set up but the consumers keep the complete control over the deployed applications. An example of Platform as a Service is Google AppEngine;

3. **Infrastructure as a Service (IaaS)**: consumers can directly use the cloud infrastructure such as processing, storage and network. They can also create and

destroy all instances they need for deploying and running their own applications. Managing the underlying cloud infrastructure is usually not allowed, but customers can have control over the operating system and the storage, with a limited control over the networking component. Examples of Infrastructure as a Service are Amazon EC2 and Google Compute Engine.

As Internet has become more reliable and faster, cloud providers started to offer new technology as cloud services. All these services are identified as Xaas (Anything as a service), and can be listed as follows even though the list may not be exhaustive because other new models might be defined in the future:

4. **Data Storage as a Service (DaaS)**: consumers store their own data on the cloud. Typical storage interfaces are used, such as File Systems. Examples of Data Storage as a Service include Google Drive;

5. **Blockchain as a Service (BaaS)**: this model represents those platforms that include tools for creating blockchain applications;

6. **Business Process as a Service (BPaaS)**: typical business processes, such as Human Resource functions, Procurement functions and Advertising functions, are automated and provided by cloud infrastructure. A company can focus on its own business objectives instead of managing technology and services that are behind the business service. This can help consumers to save their time focusing on their business instead of wasting time in other activities which are outside the company business vision;

7. **Database as a Service (DBaaS)**: consumers can store and retrieve their data to and from a database, without setting up physical hardware or software. All administrative tasks are managed by the service provider. Usually consumers do not know where their data are stored or computed. Examples of Database as Service are Google BigTable, Apache HBase and Amazon S3;

8. **Function as a Service (FaaS)**: this is an emerging new model which is changing the way users interact with cloud services. In this model, consumers can use a platform where application functionalities can be developed and run on, without building the infrastructure needed to develop and launch an application. Consumers then pay depending on the number of times the functions have been invoked or the time spent to compute the functions. This kind of model is usually described as Serverless Computing. Examples of Function as a Service are OpenLambda or IronFunctions.

1.5 The deployment model of Cloud Computing

From the deployment model perspective, Cloud Computing can be classified in 4 different types:

1. **Public Cloud**: in this model, cloud services are provided by private companies, called service providers, which allow consumers to use the physical infrastructure using one or more service models. Popular public service providers are Amazon, Google and Microsoft;

2. **Private Cloud**: sometimes an organization might decide to build its own cloud infrastructure. Usually this happens for several reasons:

(a) To optimize the utilization of existing in house resources;

(b) To reduce the data transfer among local IT infrastructure and public cloud;

(c) To keep its data safe, making the data not accessible by anyone.

In this model, the organization which owns the cloud infrastructure is responsible for the management, and the services are usually not available to other companies;

3. **Community Cloud**: sometimes, some organizations having their own physical infrastructure decide to join together, building a cloud environment on all physical infrastructures. All the different organizations joined together maintain and share the same cloud infrastructure which is usually not accessible by not members;

4. **Hybrid Cloud**: this model is a combination of two or more deployment models (private, public and community). Integration among different clouds is done by using connectors in order to optimize data exchange across different platforms.

1.6 Virtualization techniques

In the previous paragraph, many interesting cloud features have been described. Briefly, we said that consumers can get all the resources they need, through the network and in an automatic way, without any human interaction and without taking care about the amount of the required resources. Furthermore, consumers can scale up and down the resource configuration in real time, paying according to what is really used. Behind all these features there is just one word: virtualization. To achieve the shrinking and growing of resources, indeed, physical infrastructure has to be virtualized. Said in few words, virtualization is a way to hide the physical characteristics of computing resources from the way of which other systems, applications or end users interact with those resources.

There are several ways to virtualize the IT infrastructure, depending on the level where the virtualization is applied:

- **Full Virtualization**: Figure 1.1 shows the architecture of full virtualization. The horizontal blue boxes at the bottom represent the physical system and the operating system (host OS) running above it. The vertical orange boxes, instead, represent all the traditional applications running on the host. One of these applications (on the right side), is the hypervisor, which is a component used to hide the underlying physical infrastructure. The hypervisor hides the host operating system and creates a virtual hardware layer where a new operating system, named Guest OS and well-known as Virtual Machine or Virtual Instance, can be executed on. The guest operating system is unaware that it is working on a virtual hardware layer. Indeed, as traditional operating systems, also the Guest OS exploits resources, such as CPU, memory, network interface, persistent disk, but instead of being real, such resources are virtual and are mapped into the physical resources. All interactions among Guest OS and the hardware are intercepted by the Hypervisor and translated to the physical hardware. In this approach, the Guest Operating System is completely isolated and cannot interact in any way with the physical hardware. The Guest OS issues commands to what it thinks is actually hardware. All virtual devices are

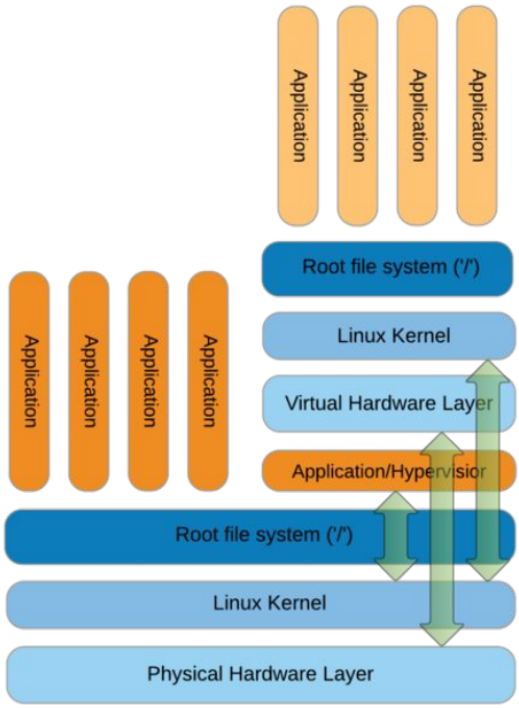

FIGURE 1.1: Full virtualization architecture

implemented via software, and this is the main reason why this virtualization technique is not able to achieve good performance;

- **Paravirtualization**: In order to improve performance of Full Virtualization, in the paravirtualization approach (Fig. 1.2) some more expensive functions issued by the Guest Operating System can be directly executed on the physical hardware instead of being executed on the virtual hardware. The Host OS has direct access to the hardware while the Guest OS has limited access to the hardware. In the paravirtualization, the guest operating system needs to be aware that it is working on a virtualized environment. It can use traditional calls to interact with the hypervisor and privileged hypercalls to interact with the physical hardware. To achieve this aim, guest operating system needs to be modified (hypercalls need to be available to the guest operating system) and, of course, unmodified operating systems cannot work properly in a paravirtualized environment;

- **Hardware-assisted Virtualization**: both full virtualization and paravirtualization techniques are totally or partially implemented via software. In order to reduce the overhead introduced by such virtualization methods, since 2005 Intel and AMD started producing CPUs with a new extension of the x86 architecture, called Intel VT-x and AMD-V, respectively. This extension makes the CPU able to support virtualization natively reducing the overhead, increasing performance and reducing the number of changes needed in the guest operating systems;

- **Operating System Level Virtualization**: this method is also well-known as Container-based Virtualization. Figure 1.3 shows the architecture of the Operating System Level Virtualization. On the Host operating system, running

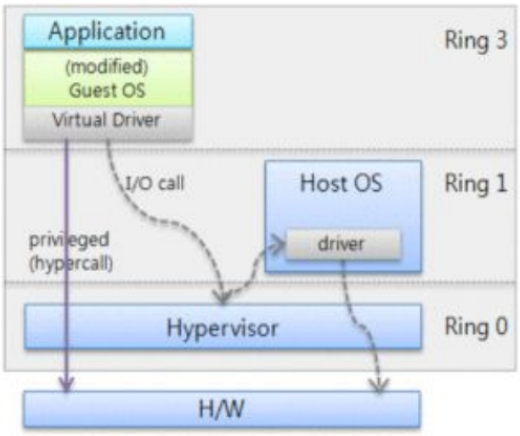

FIGURE 1.2: Paravirtualization architecture

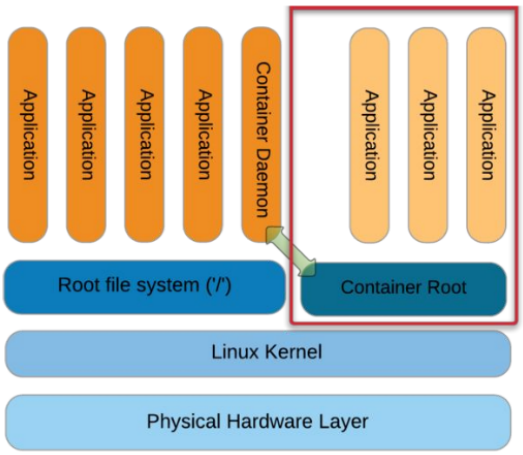

FIGURE 1.3: Operating system level virtualization architecture

on the physical hardware, several applications are running (the orange vertical boxes on the left side). A special application, named Container Daemon, is running as well on the host OS. The daemon communicates with the container where several other applications are running (the light orange vertical boxes on the right side). Applications running inside the container are isolated, and they cannot directly interact with other applications running on the host. The container may look like a real computer from the point of view of programs running on it. In this approach, the virtualization layer runs on the host operating system and then containers are lighter than the Virtual Instance. Examples of applications which can be used to achieve such kind of virtualization are Docker and Singularity;

- **Application Level Virtualization**: this virtualization (Fig. 1.4) is similar to the previous one, but in this solution just one application can be put on its own environment. Different applications work on different environments and all environments are isolated to each others. Applications usually do not interact and conflict with each others and are usually accessible through the network. An example of application used to virtualize applications is Citrix.

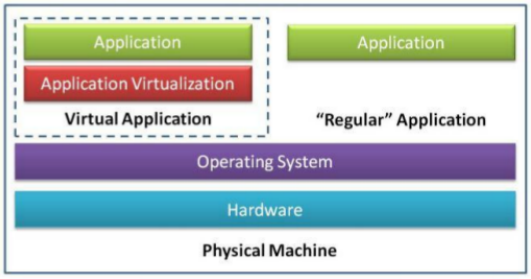

FIGURE 1.4: Application level virtualization architecture

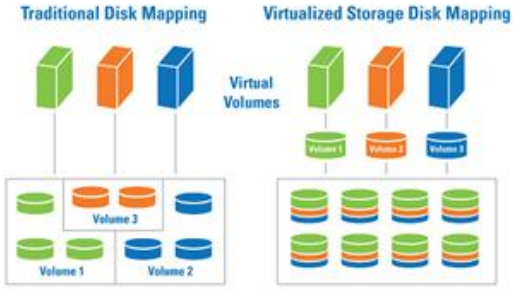

FIGURE 1.5: An example of storage virtualization

The benefits of virtualization are widespread and well-known today and the techniques described above are just related to server virtualization, which is the most familiar type of virtualization. However, virtualization is not just related to servers, but also network, storage and more can be virtualized.

- **Storage Virtualization**: this virtualization hides the physical storage architecture to users and applications running on systems making use of such storage systems. Figure 1.5 shows an example of the storage virtualization. On the left side, the traditional disk mapping is shown. In this architecture, three different servers (green, orange and blue) use three different volumes (having the same colours). Each volume is equipped with a different number of physical disks. In this configuration, users working on each server need to be aware about the physical resources they are using. A user working on the blue host cannot use disks which belong to the orange volume and the same happens for the others. By virtualizing the storage, users actually do not know where their data are stored. Indeed, all users can use all the physical disks belonging to the whole architecture. The new storage management yields better performance as read and write operations are spread across all physical volumes and can be executed in a parallel way, boosting the performance of the systems;

- **Network Virtualization**: this technique can simplify the network management. As shown in Fig. 1.6, the physical network infrastructure is virtualized in two different virtual networks (red and blue). This allows the network administrator to define rules and policy which have to be satisfied when packages are sent across the network. For example, although router C and E

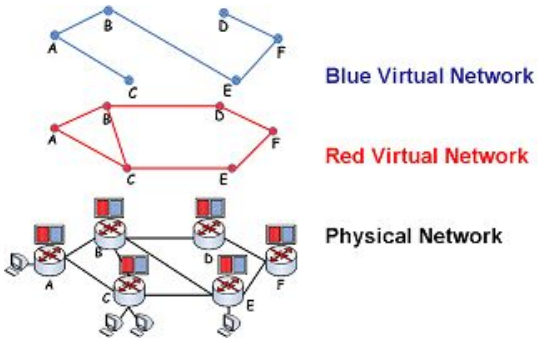

FIGURE 1.6: An example of network virtualization

are physically interconnected to each others, in the blue virtual network both routers cannot send packets directly but the messages need to follow a different route (through routers A and B). This configuration allows the network administrator to split the physical network in subnets and enable or disable communications among authorized or unauthorized nodes belonging to different subnets.

1.7 How Cloud Computing has really changed the life

The way to work for computer science engineers has deeply changed after introducing cloud computing, and it is still changing because new technologies and new services are coming up, bringing along dramatic impact on how we work, live and entertain ourselves. Let's suppose we want to develop an application for validating credit cards. In the traditional enterprise computing, the computer science engineer needs to procure and buy a server enough performing for running the application, pay for the purchase, bring it on site, configure it, setup a network connection, build and deploy the application, than manage the whole infrastructure; after introducing Cloud Computing things got different. Indeed, computer science engineer needs just to buy the virtual server with a minimal configuration, build and deploy the application, pay for what he really uses, and finally manage the virtual infrastructure. Serverless computing has changed again the computer science engineer job because it is just needed to buy the service, build and deploy the application and pay for the function calls. Developers and operators do not need to spend time setting up and tuning auto scaling policy or systems, and all system activities are managed by the service provider. Consumers then can really focus on what they are able to do better, leaving to the service provider all those activities which are behind the business objectives.

1.8 How interest in Cloud Computing has changed

The interest in Cloud Computing is growing more and more. Figure 1.7 indeed shows the number of searches for "cloud computing" made using Google Search Engine. As represented by the blue line, since October 2006, there has been a significant rising in the number of times the words "Cloud Computing" has been searched for. Figure 1.8, instead, shows the regions from where the words have been searched the most [22].

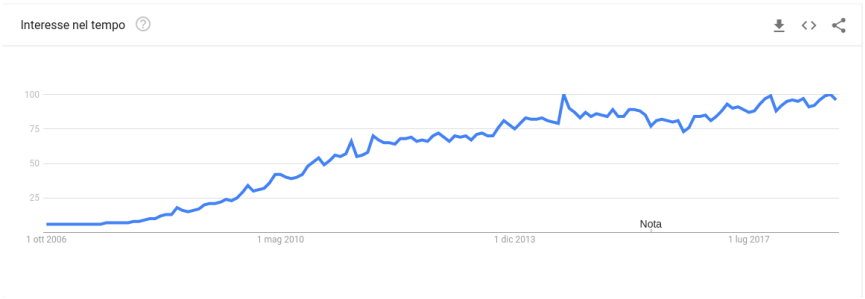

FIGURE 1.7: Number of times the word "Cloud Computing" has been searched for

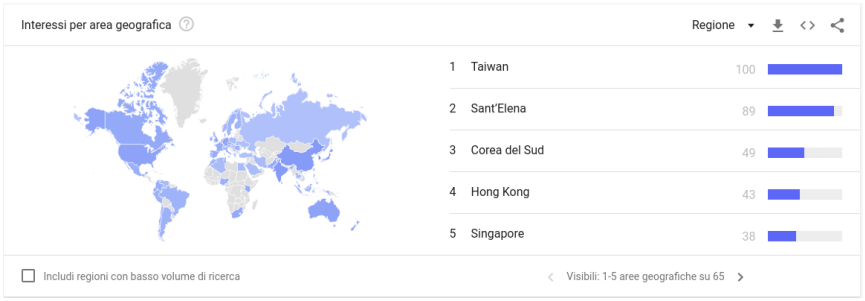

FIGURE 1.8: Regions where the word "Cloud Computing" has been searched for

Furthermore, all forecasts predict an interesting growth in the number of enterprises which decide to move their own applications and infrastructures into the cloud by 2020 with an important rising in the revenue of many cloud service providers. According to Gartner's forecast [23, 24], the worldwide public cloud services market is projected to grow 17.49% in 2019 to total $ 214.3 billions and the growth might rise up over 81% in 2022 when the worldwide public cloud service revenue forecast is $ 331.2 billions (Tab. 1.1).

TABLE 1.1: Worldwide public cloud service revenue forecast (billions of U.S. Dollars).

	2018	**2019**	**2020**	**2021**	**2022**
Business Process Services (BPaaS)	45.8	49.3	53.1	57.0	61.1
Application Infrastructure Services (PaaS)	15.6	19.0	23.0	27.5	31.8
Application Services (SaaS)	80.0	94.8	110.5	126.7	143.7
Management and Security Services	10.5	12.2	14.1	16.0	17.9
System Infrastructure Services (IaaS)	30.5	38.9	49.1	61.9	76.6
Total	**182.4**	**214.3**	**249.8**	**289.1**	**331.2**

TABLE 1.2: Cost per month for buying a cluster of four different virtual instances provided by three different service providers.

	Amazon	**Google**	**Microsoft**
Type	C5d.9x Large	N1-standard-32	D32 v3
Location	London	London	London
Instance Number	4	4	4
Operating System	RHEL	RHEL	RHEL
Virtual CPUs per Instance	36	32	32
RAM per Instance	72 GB	120 GB	128 GB
SSD HD Capability	900 GB	375 GB	384 GB
Cost per Month	**6,441.60 $**	**4,526.57 $**	**6,069.84 $**

1.9 The cloud service provider

To satisfy the rising cloud service demand many companies nowadays offer to their customers a huge broad of services, from software as a service to anything as a service. One of the leader in the cloud service field is doubtless Amazon, the well-known company which provided the first cloud service. But the list is full of other companies, such as Google, Microsoft, IBM, VmWare, Red Hat, Adobe and so on.

1.10 The price of the cloud

Cloud services are based on the pay-per-use model. This allows consumers to pay for what is really used, depending on the amount of resources required and for the time the consumers use those services. Pay-per-use allows consumers to easily adapt the changing business needs without overcommitting budgets. The consumer only pays for the services he consumes, and once he stops using them, there are no additional costs or termination fees. The payment model the cloud is based on is not new, as it is similar to what citizens pay for utilities like water and electricity.

Different service providers use different functions to compute the price to be paid by the consumers, such that the same service might have different costs when provided by different service providers. Table 1.2 shows the cost paid monthly by a hypothetical consumer making use of a specific resource configuration. The table shows the price per month that a customer should pay to use a cluster of four virtual instances, each of them having a specific configuration, provided by three different service providers. The instance configurations are slightly different because all scenarios represent one of the predefined configurations made available by each different service providers. Costs are updated at November 2018.

To encourage the usage of cloud services, many service providers offer free credits for academic purpose. Students can use, after registration, all the provided cloud services, giving them the ability to learn about the cloud platform. Several providers also offer education credits. University faculty can apply for grants for their computer science courses.

To have an idea of how much a consumer should pay to get cloud resources, many service providers usually make available tools for computing the price for a fixed configuration. For Amazon, Google, IBM and Microsoft, the tools can be found at [25–28]. Cloudorado [29] has instead developed and made freely available a price

comparison service of several cloud computing providers. It could be also referred as a price calculator for multiple cloud hosting providers, since the comparison is performed by calculating price for individually set server needs. It currently focuses on Infrastructure as a Service (IaaS) providers.

1.11 Advantages and disadvantages

Deciding whether or not moving an application or an infrastructure to the cloud is not an easy task to deal with, and it should be taken into account carefully. Cloud environments indeed can bring along several advantages but also some drawbacks. Before moving toward to the cloud solution, managers need to understand whether or not it is worth doing it. In the following, many important advantages and disadvantages are described.

1.11.1 More business, less IT

The business growth of a company is not always straight rising up in time, but it goes up and down according to the market request. This is much truer for those startups which are coming into the global market. Thus, the on-premise IT infrastructure needs to be powerful enough to be able to keep step with the business and cope with sudden market demands. To avoid losing customers and money, in the traditional IT, companies usually buy hardware and software more powerful than what is really needed. For the most of the time, however, such large infrastructure is underutilized or not utilized at all. It is easy to notice that from the company perspective this is just a cost. Figure 1.9 describes this scenario. The black line represents the demand amount in a specific time. According to the fluctuations in the market, the demand varies in time. To cope with the sudden rising of the predicted demand, the capacity of the traditional hardware, represented by the black straight line needs to be much larger than what is really needed. This requires a large capital expenditure. Furthermore, if the market does not respect the prediction, all the purchased IT infrastructure becomes soon a fixed asset no more able to build wealth but rather a cost for the company. Worse yet, if the business growth prediction is pessimistic and the actual demand breaks out, the capacity of the traditional IT might not be able to deal with such requests, leading to loss of customers and money. With the cloud, instead, companies can save their money avoiding buying unnecessary infrastructure. Thanks to its elasticity, the cloud fits better market demands (blue line) avoiding the purchase of unnecessary infrastructure. IT managers indeed can start with a small configuration and then enlarge it according to the business growth.

1.11.2 Pay for what you need

As said before, in the traditional IT infrastructure, both hardware and software, after being bought, need to be scaled up according to the predicted business growth. To deal with rising demand, companies need to buy, right from the start, an amount of licences larger than that really needed. When the demand grows more, the amount of available assets might not be enough to fulfil the demand anymore. The existing infrastructure then needs to be scaled up. As time goes by, however, the asset end-of-life time approaches more and more, making not convenient anymore to buy more licences for the existing infrastructure. The infrastructure becomes not able to fulfil the demand with an unavoidable loss in customer and money. Figure 1.10 shows this scenario. Fitting better to the demand and thanks to the pay-per-use model,

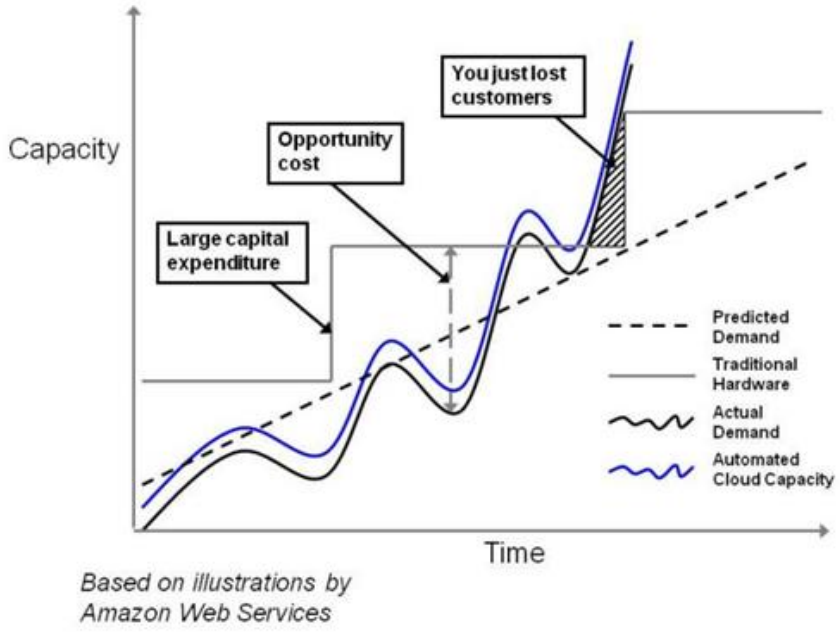

FIGURE 1.9: The cloud infrastructure fits the business growth better than on-premise infrastructure

cloud can reduce the cost for buying licences and the company just pay for what it really needs.

1.11.3 Time to deployment

Cloud can also reduce the time to put a new infrastructure on line. Using cloud solutions, indeed, the total time to deployment is reduced because some traditional activities, such as the procurement, are dropped and many others are reduced, such as installation, configuration and test. Figure 1.11 compares the time to deployment spent with the cloud and without the cloud. With a cloud solution, time to deployment is heavily reduced allowing the company to deliver products in the market sooner. Cloud computing also enables faster comparison among all possible solutions from different providers.

1.11.4 No upfront costs

Being based on the on-demand model, cloud infrastructure allows dropping off the upfront costs. In the traditional IT, indeed, before starting their own business, organizations need to face up a large upfront investment for purchasing all the required hardware and software infrastructure, needed to support the business. The amount of infrastructure an organization should buy is usually predicted with a detailed capacity planning. This infrastructure is provisioned with fixed capacity and its amount is always higher than the predicted market trend (Fig. 1.12 With cloud solution instead, managers can just buy what they really need to start the business. Fitting better the market trend, cloud solution can drop the upfront costs off.

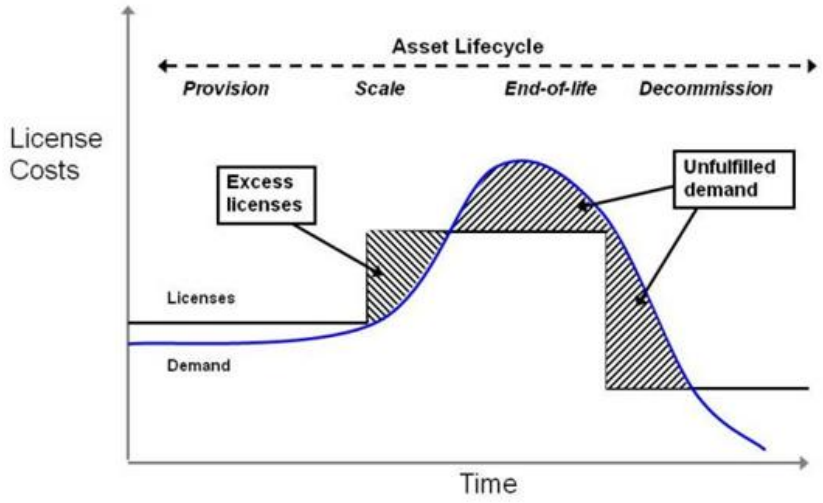

FIGURE 1.10: With the cloud infrastructure, consumers can buy what really needed

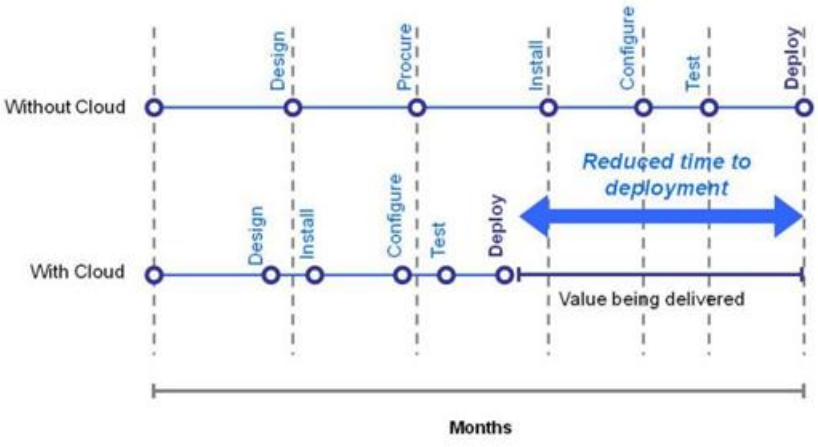

FIGURE 1.11: Comparing the time-to-deployment with and without the cloud

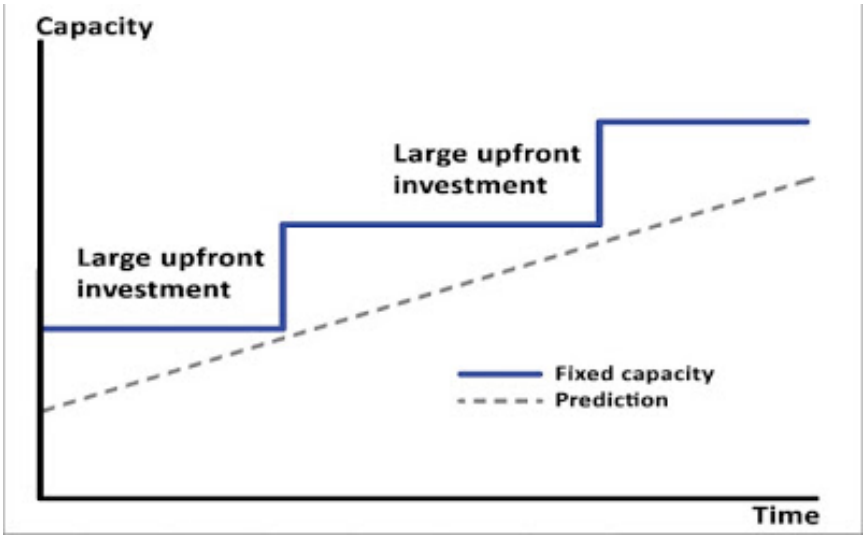

FIGURE 1.12: Traditional IT infrastructure requires large upfront investment

1.11.5 Focusing on the business

In the traditional IT, all the activities related to the management of the IT is in charge of the business himself. Hardware and software need to be bought, managed, updated and replaced when needed. A proper infrastructure management requires knowledge which is often out of the business aim. For many companies, indeed, technology is just used to support the production and not as the business itself. To deal with IT requires enough human resources and these need to be hired and properly trained before being able to manage the infrastructure. With cloud solutions, instead, all the IT management is in charge of the cloud service provider, allowing the manager to focus on its own business instead of spending time and money on something that is out of the business itself.

1.11.6 Scaling for saving costs

Companies using cloud services can shrink and stretch the cloud infrastructure according to the predicted business growth, but such scaling operations can also be made with a more fine-grain frequency, allowing reducing further the cost faced up by the company. Usually the customer demand is not always uniform making the systems well utilized during peak period but idle most of the times when the peak period is over. Fluctuations may occur seasonally, monthly, weekly or even daily. If the IT capacity stays the same along all the period, such as when the organization makes use of the traditional IT, since the entire infrastructure is running all days in a week, the company wastes resources during the normal days. For making cloud solutions more appealing, service providers have developed an auto scaling solution which is able to balance this problem by increasing or decreasing capacities automatically depending upon a pattern. Figure 1.13 shows how it is possible to reduce by 25% off the cloud cost using an auto scaling solution.

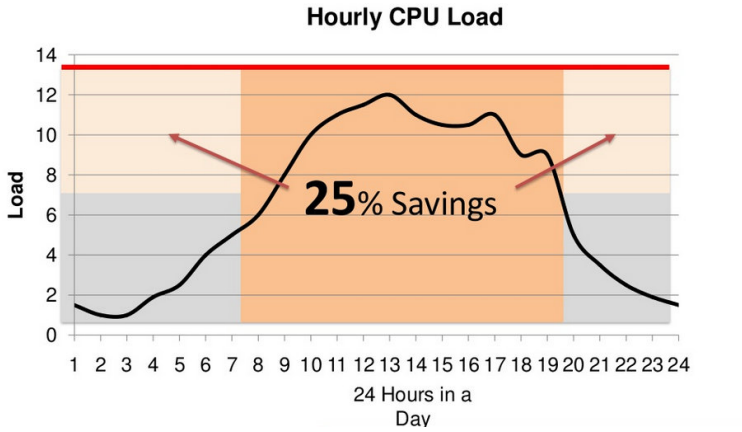

FIGURE 1.13: Cloud computing can save money

1.11.7 No planning

A physical infrastructure is not usually bought straight away, but the procurement business process follows a consolidate sequence of phases which start from the planning and end with the good purchase. In the traditional IT, then, the purchase of the IT needs to be carefully planned enough time in advance. Being based on virtualization, from the customer perspective, cloud infrastructure instead appears to be infinite and immediate, and it can be acquired and released whenever the customer wants and without any planning: the resources seem to be always available.

1.11.8 No electricity and location costs

After being bought, the physical infrastructure needs to be stored. Usually companies reserve one or more rooms to be used as data centre, which need to be equipped with proper solutions to keep all servers safe and secure. Data centre is not a single thing, but rather, a conglomeration of elements, including servers, storage subsystems, networking switches, routers and firewalls, as well as the cabling and physical racks used to organize and interconnect the IT equipment. A data centre must also contain an adequate infrastructure, such as power distribution and supplemental power subsystems, including electrical switching; uninterruptible power supplies; backup generators and so on; ventilation and data centre cooling systems, such as computer room air conditioners. Making use of cloud solutions, organizations get rid of all costs related to building and maintaining the data centre as well as the costs related to energy consumption and those related to safety and security practices.

1.11.9 Meet the economy of scale

Building and maintaining a data centre is not costless. Amortized plants, amortized IT assets, operating and energy are typical costs an organization may have . Some of these costs are always constant and do not change frequently in time. Others are instead variable and increase depending on several factors, such as IT load, number of racks, average rack density or data centre square footage. Others, finally, might instead decrease, in proportion, depending on other factors. For example, a system administrator managing hundreds of systems will hardly get a salary which is thousands of times higher than another system administrator managing tens or even

fewer systems. Based on this observation, managing thousands or even more systems, public clouds appear financially more attractive because a private data centre may not benefit from the same economies of scale.

1.11.10 Better utilization of on-premise infrastructure

Sometimes companies might be concerned about moving their own data to the cloud and prefer to keep them safe inside their own infrastructure, losing all the already mentioned advantages. When security reasons stop the migration towards the public cloud, companies can however take advantages by building a private or a community cloud. A private or community cloud is completely built and managed by the organization or the team of organizations which decided to join together, and this might be hard for those companies with a lack in the know-how, but, on the other hand, private or community clouds offer more flexibility: if a company needs to have things built in a very specific way, with specific hardware, it has control over that. Building a cloud infrastructure over the on-premise system allows the company to achieve a better utilization of the underlying physical architecture. The private cloud offers most of the advantages of the public cloud: self-service and scalability, multi-tenancy, the ability to provision machines and computing resources on-demand, compliance and improved security.

1.11.11 More power for free

In order to provide higher quality services, cloud providers need to keep step with technology, upgrading their systems and replacing old-fashioned systems with new and more powerful ones. The new systems are usually put in production environment and then provided to customers for free, without any raise in costs. This means that customers will have more powerful systems at the same cost.

1.11.12 Cloud is greener

An important aspect that companies should keep in mind is about the impact of a data centre on the environment. Keeping devices on implies energy consumption and then CO_2 emissions which have bad effects on our environment. Making use of cloud solutions, companies can reduce the energy use and carbon footprint of computing. A six-month study conducted by Lawrence Berkeley National Laboratory (Berkeley Lab) and Northwestern University with funding from Google has found that moving software applications from local computer systems to centralized cloud services could cut energy consumption by up to 87 percent — about 23 billion kilowatt-hours. This is roughly the amount of electricity used each year by all the homes, businesses and industries in Los Angeles [30].

1.11.13 All that glitters is not gold

As described before, there are several advantages related to the usage of cloud solutions. Before moving applications and infrastructure to the cloud, however, companies should also carefully think about the disadvantages which are related to such new technology. Here is a small list of possible disadvantages that should be kept in mind:

- Cloud is not always cheaper than on-premise solutions. Indeed, if an application has a regular and predictable requirement for computing services it may be more economical to provide that service in-house;
- Cloud services might not be always available. Some outages might happen and these might make cloud infrastructure inaccessible. Furthermore, working with remote systems implies having a good internet connection. If this goes down, services are no more accessible although they are working properly;
- Data stored in a cloud infrastructure are potentially accessible by unauthorized people, such as technicians working for the service provider, who can steal data and make a not appropriate usage of it;
- Being accessible through the network, cloud services can be accessible by anyone. Hackers can attack computing resources and steal data;
- Migrating applications into the cloud infrastructure might not be easy or cheap and the migration costs might be higher than the money saved using the cloud;
- Deciding to buy cloud services, companies become highly tied to the service provider and all future business decision might be shaped by service provider;
- If the physical infrastructure where cloud services work on is not designed to allow customer applications to work properly, those applications might perform badly. This is actually true for scientific applications which sometimes need to rely on a high performance network infrastructure.

1.12 Costs, benefits and trade-offs

This paragraph summarizes several previous works talking about the costs and benefits which are related to the usage of cloud infrastructure rather than HPC systems. The researches cover both economical and performance aspects which are the two main important aspects a user usually should consider. Indeed, a user who is interested in running a job for getting some results, may be much more interested in getting the results in the fastest possible time or in the cheapest way or further any trade-off between the two. The papers described in this section try to understand in which circumstances a user might prefer HPC infrastructure rather than cloud computing.

1.12.1 Trade-offs from the performance point of view

Running the same application on two or more different infrastructures can yield different performance results. This depends on several factors, many of them are strictly related to the application itself but many others instead are dependent on the architecture where the application is executed on [31]. An application designed to be run on a specific infrastructure might have bad performance when executed in a different one [32–34]. Sometimes, better performance might be achieved just by making small changes in the application [35, 36] or by tuning system parameters (such as HTTP congestion window size or MPI parameters) [37–42]. Most of the times, instead, the decrease in performance is mainly affected by the infrastructure layer, and this is truer for cloud infrastructures, which are usually built on a virtualized layer. Virtual instances, indeed, work on the hypervisor which hides the underlying physical layer. Therefore, the interaction between application and

devices brings unavoidable overheads, which turns into decrease in performance. Applications running on a virtualized environment might suffer from not negligible overhead, as described in [43], which can highly affect the application performance. Nevertheless, performance of a single node on the cloud is similar to traditional clusters, and virtualization brings negligible overhead [44] but when the number of concurrent virtual instances or virtual cores increases, the effect of the virtualized environment might become more perceptible, and might badly affect the application performance even up to 60%, mostly for tightly coupled applications [45]. Many efforts have been done in these years to lessen the effects of virtualization, introducing new technologies [31, 44–48] but several researches show that cloud computing is still not ready for running scientific applications, mostly because of the lack in performance mainly due to the low performance interconnection network used by large commercial cloud infrastructures [31, 32, 34, 43, 44, 46, 49], which have considerably higher latency and lower bandwidth than Infiniband or proprietary interconnections. Other reasons which might contribute to reduce performance in the cloud is that on such infrastructure many virtual instances share the same network and, in general the physical layer. However, in some cases, performance can be improved just by setting up properly the infrastructure layer, as described in [47, 50]. Actually the debate about the utility of cloud computing for running scientific applications is still open and in the recent past it has been also tried to understand if a cluster made of virtual instances could reach the top500 list. This is the question authors in [51] tried to answer running a scientific application on an Amazon cluster made of 126 cores. Results show that performance on a single node on EC2 are similar to that observed on real HPC systems, but the amount of memory and the network infrastructure are not able to keep the performance when the number of nodes increases. The application used for the test is LINPACK, which is the same benchmark tool used to assess the performance of HPC systems for the Top500 list. Authors concluded the paper saying that cloud seems not to be ready to be used for running HPC applications. Anyway, since November 2013, a cluster made of 26,496 Amazon instances reached the 64th place of the prestigious list [52], although nowadays the cluster has been put out of the list (in June 2017, the cluster was in the 438th place).

1.12.2 Trade-offs from the economical point of view

Running applications on the cloud infrastructure is not free of charge and furthermore the cloud price might also be higher than the cost for running the same applications on an on-premise system. A comparison among costs, then, can help researchers to choose the proper infrastructure from an economical perspective. Of course, for a good cost-benefit comparison between cloud and HPC systems, several factors should be taken into account [53] such as costs of servers, network, power, software, cooling, real estate, facility, support and maintenance, computation time, amount resources, amortization and so on.

In several works, described in the follows, many authors have compared the cost for running an application in an HPC system against the cost for running the same application into the cloud. Authors in [33] compare the cost per hour for the Amazon Cloud with the cost per hour for an HPC system, computed as function of the amount of jobs submitted for the execution in a period, the number and the speed of the cores in the HPC architecture, the maintenance cost for the infrastructure but without taking into account other factors such as waiting time, job failure and job setup time. According to this research, cloud seems to be not convenient from the

economical perspective. Authors in [32] instead used a utility function to understand, from the economical point of view, whether cloud computing can be seen as a good trade-off. The utility function represents a class of users who are less or more interested in having results faster and about how much they are willing to pay more. Using these functions, authors showed that there are some cloud configurations which can be considered as interesting for different classes of users. In [44], instead, authors compare the cost for running a suite of applications on the cloud and on an in-house cluster, but without taking into account neither maintenance and training costs nor upfront costs needed to buy the physical infrastructure.

1.13 Wrapping up

Understanding whether or not cloud infrastructure can be a feasible place for running applications from both economical and performance point of view is really important, especially for scientists who are much more interested in running their own scientific applications with a performance as high as possible. As moving an application on a different infrastructure is not effortless, having in advance such knowledge might help scientists to understand if moving the application to the cloud is worth doing it. This is the thread leading this research.

Chapter 2

HPC and Cloud Infrastructure

This chapter describes both physical and virtual infrastructures used along all tests described in this work.

2.1 Marconi

2.1.1 The architecture

Marconi [54] is the new Italian Tier-0 system, co-designed by Cineca and based on the Lenovo NeXtScale platform, that substitutes the former IBM BG/Q system named Fermi. Actually, Marconi is made of three different partitions, named respectively A1, A2 and A3. The first partition, which has been used in all the tests described in this work, is composed of 1512 nodes having each two 18-core Intel Xeon E5-2697 v4 processors, based on the Broadwell architecture and running at 2.30 GHz. Each core has 3.5 GB of RAM for a total of 128 GB per node. All cores in the same processor share a single 45 MB L3 cache. Both processors, instead, share 128 GB RAM. Each processor can connect to the RAM using four channels having a bandwidth of 76.8 GB/s. There are also two QPI links per processor, based on Intel QuickPath Technology, which provide high-speed, point-to-point connections for data transfer inside and outside the processor. All nodes are interconnected by Intel OmniPath Architecture, 100 Gb/s. Since the end of 2016 a new partition, named A2, has been added, equipped with the latest generation of the Intel Xeon Phi product family (Knights Landing). The new partition is composed of 3600 nodes having a single 68-core processor Xeon Phi 7250 CPU (Knights Landing) at 1.40 GHz. Each core has 16 GB/node of MCDRAM and 96 GB/node of DDR4.

Jobs on Marconi are submitted through Slurm [55], an open source job scheduling system for large and small Linux clusters. Using Slurm, users can specify, through a bash script, the task to run, the amount of required resources (number of cores and amount of memory), the wall time, which is the time the job might be left running the most, and finally the queue where the job has to be put on. All partitions offer several queues, each of them having different features. For example, queue named *knl_usr_dbg* can host jobs requiring no more than two nodes and running less than 30 minutes, while *knl_usr_prod* can host jobs requiring even more than 1024 nodes and running up to 24 hours. A queue is not handled like a pure FIFO queue. Each job is assigned a priority index which is computed with a complex formula taking into account many factors such as the waiting time in the queue, the size of the job (core number and amount of memory), the required wall time and furthermore a fair share factor which slows down jobs submitted by users who have almost spent their month-hours. Because of this scheduling policy, the waiting time spent by a job in the queue can be high.

2.1.2 The interconnection network

The characterization of the interconnection network is of paramount importance when a researcher wants to understand how an application based on a communication pattern might behave being run on a system with a fixed interconnection network. In literature, there are several models which can be used to characterize the network layer, but we decided to use pLogP model. An overview about other models can be found in Appendix B, as well as a deep explanation about pLogP model. All parameters in pLogP are measured sending and receiving messages between just two nodes belonging to the same parallel architecture. When the network infrastructure is not homogeneous or symmetric, pLogP can still be applied but the nodes need to be grouped into clusters and the pLogP parameters need to be gathered for all clusters [56].

In Marconi, according to the well-known physical network architecture, a communication between two processes can be classified into intranode and internode communication. With the term intranode communication we refer to a communication between two cores belonging to the same node; with the term internode instead we represent a communication between two cores belonging to two different nodes. The default communication fabric used by the HPC system is *shm:tmi* that makes use of shared memory for intranode communication and TMI for internode communication.

Since the communication on Marconi can be classified into intranode and internode, a complete characterization of the Marconi interconnection network requires to gather all pLogP parameters for both intranode and internode communications.

Table 2.1 shows the pLogP parameters we gathered on Marconi for both intranode and internode communications. All times are expressed in seconds. The values represented in the table have been gathered running the pLogP tool ten times and taking the median value for each message class. We decided to use median instead of mean in order to ignore any outlier. The numerical values nearby the g parameter represent the size of the message. As expected, internode communication is more expensive than intranode.

2.2 Cloud Infrastructure

2.2.1 The architecture

The cloud infrastructure used for our tests is provided by Google. We decided to use Google instead of any other provider because University of Pavia signed an agreement with the famous American company for getting free credits to spend on all cloud services. By using Google Compute Engine console, we instantiated a cluster made of four virtual instances (n1 series). To have a better comparison between cloud and HPC systems, we decided to use a cloud configuration matching as much as possible the node configuration on Marconi system. The following characteristics are related to virtual instances as shown in the Google Console, and not to the real bare metal where cloud infrastructure works on. Each virtual instance has eight virtual CPUs. Each virtual CPU is actually a single hardware hyper-threaded on an Intel Xeon E5 v5 (Broadwell), running at 2.2 GHz, 24 GB of memory RAM and 56 MB of L3 cache [57]. All four virtual machines run on the same geographic zone (us-central-c1) and are interconnected to each others with a Virtual Private Cloud Network [58–60]. Each vCPU has a 2-Gbps outbound cap for peak performance. Each additional vCPU increases the network cap, up to a theoretical maximum of

TABLE 2.1: PLogP Parameters for Marconi Interconnection Network.

PLogP Parameters	Internode	Intranode
L	2.50E-06	4.00E-07
g([0B, 0B])	1.00E-06	6.00E-07
g([1B, 4B])	1.00E-06	6.00E-07
g([4B, 0.5KB])	1.20E-06	6.00E-07
g([0.5KB, 1KB])	1.45E-06	6.00E-07
g([1KB, 2KB])	1.55E-06	6.00E-07
g([2KB, 5KB])	1.80E-06	1.00E-06
g([5KB, 6KB])	2.60E-06	1.50E-06
g([6KB, 10KB])	3.00E-06	1.95E-06
g([10KB, 20KB])	4.70E-06	3.00E-06
g([20KB, 50KB])	1.25E-05	6.20E-06
g([50KB, 100KB])	1.21E-05	1.22E-05
g([100KB, 200KB])	2.99E-05	1.85E-05
g([200KB, 0.5MB])	4.55E-05	3.74E-05
g([0.5MB, 1MB])	1.09E-04	8.59E-05

TABLE 2.2: Google Cloud Virtual Instances configurations.

	Cloud
Number of nodes	3 VM
Cores per node	8 vCPU
CPU	Intel Xeon E5
CPU Architecture	Broadwell
CPU Frequency	2.20 GHz
Memory RAM per node	24 GB
L3 Cache	56 MB
Interconnection Network	Unknown
Operating System	CentOS 3.10 - 64 bit
MPI Version	OpenMPI v3.0

16 Gbps for each virtual machine [A60]. The operating system installed on each virtual instance is a 64-bit CentOS 3.10 equipped with OpenMPI version 3.0. Table 2.2 shows a detailed configuration of the cloud infrastructure.

2.2.2 The interconnection network

After setting up the cluster of virtual instances, we proceeded to get a complete characterization of the interconnection network, as we did on Marconi. As the cloud interconnection network is hidden to the user, before starting gathering pLogP parameters, a preliminary analysis was needed in order to discover 1) if the interconnection network is symmetric, and 2) how the cores (vCPUs) can be grouped together according to the node (virtual instance) they belong to. Using a custom tool, we have been able to classify all communications in two classes, as well as on Marconi: intranode and internode. Here too, with intranode communication we describe a communication between two processes running on two cores belonging to the same

TABLE 2.3: PLogP Parameters for Cloud Interconnection Network.

PLogP Parameters	Internode	Intranode
L	3.41E-05	1.00E-07
g([0B, 0B])	1.42E-05	3.00E-07
g([1B, 4B])	1.42E-05	3.00E-07
g([4B, 0.5KB])	1.42E-05	3.00E-07
g([0.5KB, 1KB])	9.50E-06	6.00E-07
g([1KB, 2KB])	1.77E-05	7.00E-07
g([2KB, 5KB])	2.62E-05	1.30E-06
g([5KB, 6KB])	3.19E-05	2.30E-06
g([6KB, 10KB])	3.19E-05	2.30E-06
g([10KB, 20KB])	4.39E-05	3.10E-06
g([20KB, 50KB])	5.74E-05	5.70E-06
g([50KB, 100KB])	1.87E-04	1.00E-05
g([100KB, 200KB])	3.20E-04	1.91E-05
g([200KB, 0.5MB])	5.10E-04	4.43E-05
g([0.5MB, 1MB])	9.01E-04	9.74E-05

virtual instance; with internode communication we refer instead to a communication between two processes running on two cores belonging to two different virtual instances. After clusterization, we used pLogP for gathering all the required parameters. Table 2.3 shows the pLogP parameters we gathered on the cloud for both intranode and internode communications. All times are expressed in seconds.

As for Marconi, even for the cloud the pLogP parameters have been gathered running ten times the pLogP tool and taking the median value for each message class. Here too, the numerical values nearby the g parameter represent the message size. As on Marconi, even on the Cloud Infrastructure, the internode communication is more expensive than the intranode one.

2.3 Comparing network performance between both infrastructures

As highlighted above, on both architectures the internode parameters are higher than the intranode ones. This means that sending a message between two processes belonging to different nodes (or virtual instance) is more expensive than sending the same message between two processes belonging to the same node. Furthermore, all parameters gathered on the cloud are higher than the corresponding parameters gathered on Marconi. It is also worth noting that on the cloud the ratio between intranode communication and internode communication is much higher than the same ratio on Marconi: an intranode communication in the cloud affects more the application performance than on Marconi. To make the difference more noticeable, let's consider a 6-KB message, which is, as described later, the typical message size used by one of the applications described in this work. According to the pLogP model, described in Appendix B, it is possible to compute the time spent to send a message having a fixed size. The time spent on Marconi for sending up an internode message is equal to 5.10E-06 seconds (2.50E-06 for L parameter plus 2.60E-06 for g parameter), while the same message sent in an intranode communication takes only

1.90E-06 seconds (4.00E-07 for the L parameter plus 1.50E-06 for the g parameter). Then the ratio between internode and intranode on Marconi is almost equal to 2.68. In the cloud instead this ratio is 27.5 (more than 10 times higher than on Marconi). This ratio in the cloud gets equal to 120 when a 4-byte message is sent against 3.5 on Marconi.

Chapter 3

Cross Motif Search and BloodFlow

This chapter describes two different applications which are used in the experiments described in the follows. The first application is a proteomic application, named Cross Motif Search, based on a simple communication pattern. The second instead, named BloodFlow, is a haemodynamics application based on a complex communication pattern. We decided to choose such applications because, from the communication point of view, one is the opposite of the other and can therefore be considered as representative of a wide range of scientific applications. Studying such applications might help to understand the behaviour of many other similar applications. Cross Motif Search and BloodFlow have been compiled and executed first on Marconi and then on the cloud infrastructure, and their performance results are described in chapter 4.

3.1 Cross Motif Search

The recognition of similar motifs in the secondary structures of a protein pair is an important problem in bioinformatics. Due to the computational complexity $O(m^2)$ in the number m of candidate motifs within a protein, exhaustive search for recurring geometrical pattern takes a long time. Furthermore, to process in a reasonable time a large dataset containing several thousands of proteins, such as the PDB [62], a parallel implementation of the algorithm is required. Actually, there are several algorithms in literature [63–67], such as ProSMoS [68, 69], PROMOTIF [70] and MASS [71], but for our research we decided to use Cross Motif Search, which has been designed and developed at University of Pavia.

Cross Motif Search (CMS) [72] is a biological application which is able to search for recurring geometrical patterns in the secondary structures of proteins. A single run of CMS is able to look for similarities between a pair of proteins. The algorithm, instead of considering the topological/biological description, as used by other algorithms, relies on the geometrical description of the structural motifs, which can be simply viewed as line segments. The two most common types of secondary structures, named alpha helix and beta pleated sheet, are represented as simple line segments in a 3D space and then CMS uses the generalized Hough transform [73] to find recurring geometrical patterns. Segment co-occurrences are detected, with a chosen accuracy in 3D position, within the protein structure. After completing the research, the application displays the outcoming similarities by using a 3D visualizer (Fig. 3.1). CMS has been coded according to different parallel paradigms, to expose the best approach for speeding-up protein analysis.

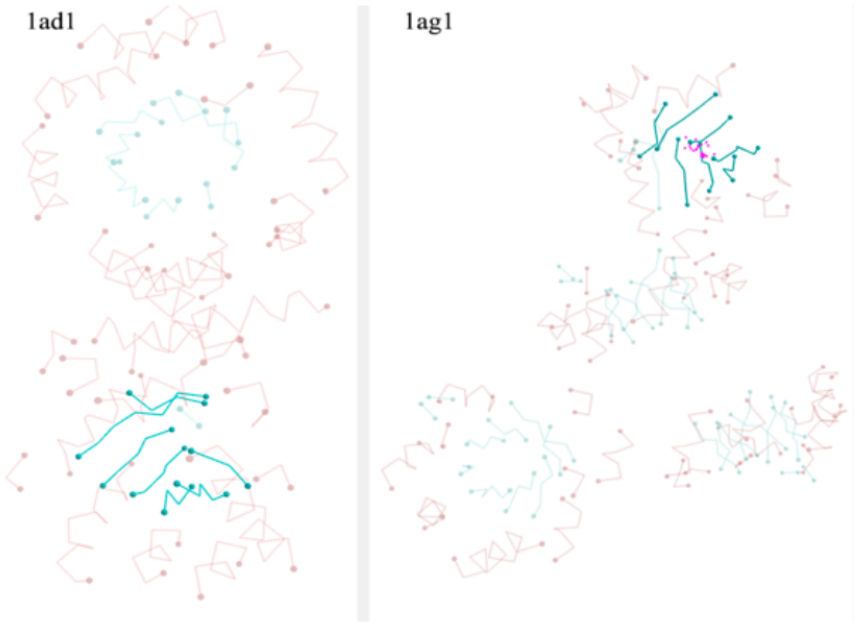

FIGURE 3.1: Secondary structure similarities found by CMS

3.1.1 The OpenMP implementation

The first version of CMS was purely sequential and was able to find recurring patterns of structural elements of a set of proteins in a serial way. In order to exploit the multi-core and many-core architectures and make the computation faster even on a large dataset of proteins, the original implementation was then extended with a new version, called OpenMP-CMS making use of the OpenMP standard [74]. The key idea in OpenMP implementation was to parallelize the inner kernel of a single CMS run, so that searching for the geometrical motifs in a single pair could be distributed on multiple threads, all of them cooperating together in order to find recurring geometrical patterns. Scalability tests, made on a single Intel Xeon server equipped with 32 64-bit cores, revealed that OpenMP-CMS was able to scale up to 8 threads. In those tests, a small dataset containing 200 proteins files was used. The intrinsic limits imposed by the greedy structure of the algorithm was the main factor limiting the application scalability. Nevertheless, as the number of proteins in the set increases, the execution time of the algorithm becomes unmanageable. So, in order to improve performance, a new hybrid implementation was developed with an MPI approach, to benefit from HPC large facilities.

3.1.2 The hybrid implementation

The hybrid solution [75], named Hybrid-CMS, extends the OpenMP version and makes also use of MPI as standard for sending and receiving messages, making the application able to be executed not only on a shared memory architecture but also on a distributed memory architecture. In this implementation, the protein data set is shared at the file system level and the whole problem (which is the computation of the total dataset) is decomposed in small tasks which are sent among several MPI processes. After starting the application, two different sets of processes are created. The first set contains just one process, named master; the other set instead contains

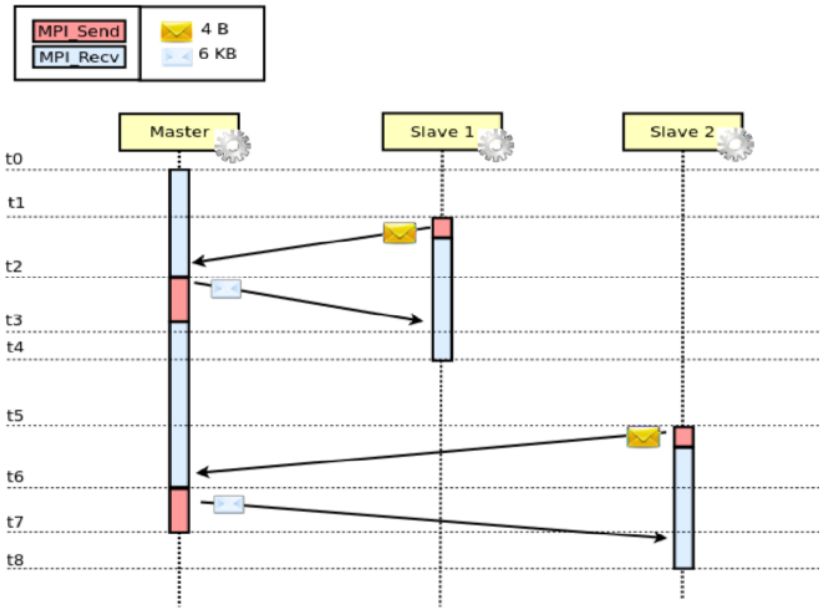

FIGURE 3.2: Communication model in Hybrid-CMS

n-1 processes, named workers. The master process is responsible to select the next task and send it to the workers. Then, the master implements the load balancing policy. Master and workers communicate to each others using send and receive point-to-point MPI functions, such as *MPI_Send* and *MPI_Recv*. At the startup, the master reads the dataset, then generates all possible protein couple combinations. When a worker is ready, it sends a *READY* message to the master, asking for a new task to compute. The master, then, sends to the requiring worker the name of two proteins not compared yet. After receiving the new task, the worker spawns into one, two or more OpenMP threads (depending on the configuration) that cooperate in order to find recurring geometrical patterns in the secondary structure of that given protein pair. The number of threads created by each process is the same for all running processes and it is defined before the application starts. After sending a task, the master waits for the next request coming from any worker. The worker, instead, after completing his job, inquiries again the master for the next task, if any. The application stops when all possible couples have been submitted to the workers for computing. In the follows, we will refer to the notation *x:y* to describe, respectively, the process number and the thread number. So for example, *16:4* means that the application has been executed with 16 MPI processes. Each MPI process, after getting the protein pair, spawns into 4 OpenMP threads cooperating together. This run uses 64 cores to compute the whole dataset.

3.1.3 Communication model in Hybrid-CMS

After starting the Hybrid-CMS implementation, the master process retrieves, from the file system, the list of all XML files belonging the dataset of interest in the current computation. It distributes the workload by assigning couples of proteins to each worker sending messages that contain protein couple names according to the protocol depicted in Fig. 3.2.

The communication pattern is very simple. Indeed, only data exchanges between master and workers are allowed. No communication among workers happen during

the whole execution and no collective operations are used to gather or scatter data. Figure 3.2 shows the communication pattern among the master and two workers. At time t_0, the master invokes the receiving function, waiting until it gets a ready message from any worker. When a worker is ready, at time t_1, it sends the ready message (4 bytes) to the master, asking for the next task to compute, then it invokes the receiving function waiting for the reply. A task is merely the protein pair to be processed. Due to the latency and network bandwidth, the message sent from the worker at time t_1 is completely received by the master at time t_2. When the master receives the message, it returns to the worker a 6-Kbyte message containing a protein pair not already processed. After receiving the message at time t_3, the worker, at time t_4, performs a CMS run on the protein pairs. After completing its own task, the worker sends a new ready message to the master for the next task and the communication cycle begins again. When all protein pairs have been compared, the master broadcasts an *EOF* (end of computation) message to all workers concluding its execution. The number of messages sent and received from master and workers depends on the number of proteins belonging to the entire dataset. Having n protein files, the number of messages sent and received from the master is always equal to

$$N = \frac{n * (n-1)}{2} \tag{3.1}$$

As described in [75], the load balancing algorithm of Hybrid-CMS can be classified as dynamic, synchronous, demand-driven, centralized and one-time assignment.

To avoid useless wasting time and improve the performance, the master is implemented in multi threaded mode. After starting the application, indeed, the master process spawns into two concurrent OpenMP threads. The first one, named primary master, waits for the requests coming from the workers; the secondary master instead behaves as any other worker, but differently from that, it communicates with the primary master by using the shared memory instead of sending and receiving MPI messages through the network.

3.1.4 The protein dataset

The large dataset, used to study scalability, efficiency and speed-up of Hybrid-CMS, contains 1,549 protein files, for total number of combination equals to 1,198,926, according to the formula 3.1. Each protein in such dataset belongs to a distinct superfamily in the Protein Data Bank [62], which nowadays contains more than 145,000 biological macromolecular structures. To build the dataset, we selected from the Structural Classification Protein (SCOP) [76] the first member of each superfamily. The first member is defined as the *first* in lexicographic order. Then, we sorted (again lexicographically) the resulting subset of proteins, selecting the first 1549 as the large dataset used for our tests. A complete list of all proteins can be found in [74].

Selecting the dataset in such way, we can almost assure that the Q score (basically a biological measure of the similarity between two proteins) of any two proteins in the dataset is quite low. And this is coherent with the original aim of CMS that is uncovering previously unseen similarities in unfamiliar proteins.

3.2 BloodFlow

BloodFlow is a tool which can be used by surgeons and doctors for getting support during study and analysis of the cardiovascular systems in order to run simulations

of patient specific haemodynamic of an aorta through computational fluid dynamic analysis.

The application is able to simulate biomedical problems linked the cardiovascular system [77, 78]. In order to simulate the detailed three-dimensional blood flow of a patient specific vessel during a whole heartbeat, this tool relies on a Navier-Stokes partial differential equation system, which is solved by using numerical approximations. For the numerical solutions of the Navier-Stokes partial differential equation system, the application makes use on LifeV [79, 80], an open source parallel library, written in C++ and MPI, which is able to solve the differential equation system using the Finite Element Method [77] for the numerical approximation. The core component of LifeV is Trilinos [81] that is used in the library to perform parallel linear algebra operations. Some difficulties are linked to the computational mesh made up to few millions of elements, the moderate Reynolds number (up to few thousands in the ascending aorta) and the possibility of turbulent flows, in particular in diseased cases. For all these reasons, such a type of simulations is computationally expensive and may take up to several hours also in a High Performance Computing context. More in details, the unsteady Navier-Stokes equations — for which the numerical solution generally consists in the velocity and the pressure of the fluid in each point of the computational mesh — are discretized both in space and in time resulting, for the most common biomedical applications, in tens of millions of spatial degrees of freedom and up to thousands of time steps for each heartbeat.

3.3 Communication model

Describing the application communication model is not easy because many components are involved and each of them uses its own pattern to send e receive data. As described above, the application in mainly based on LifeV, a parallel library written in C++ for the approximation of Partial Differential Equations (PDEs) by the finite element method in one, two and three dimensions. For an exhaustive explanation of LifeV, a good work for a deep explanation is [82]. Parallelism in LifeV is achieved by domain decomposition strategies. In a typical simulation, the main steps involved in the parallel solution of the finite element problem using LifeV are the following:

1. All the MPI processes load the same (not partitioned) mesh.
2. The mesh is partitioned in parallel using ParMETIS. At the end each process keeps only its own local partition.
3. The degrees of freedom (DoF) are distributed according to the mesh partitions. By looping on the local partition, a list of local DoF in global numbering is built.
4. The Finite Element (FE) matrices and vectors are distributed according to the DoFs list. In particular, the matrices are stored in row format, for which whole rows are assigned to the process owning the associated DoF.
5. Each process assembles its local contribution to the matrices and vectors. Successively, global communication consolidates contributions to shared nodes (at the interface of two subdomains).
6. The linear system is solved using an iterative solver, typically either a Preconditioned Conjugate Gradient (PCG). The preconditioner runs in parallel. Ideally, the number of preconditioned iterations should be independent of the number of processes used.

7. The solution is downloaded to mass storage in parallel using HDF5 for post-processing purposes.

On each of these steps, communication among processes is performed using MPI as communication standard. A deeper overview about the MPI functions used in communication will be done in chapter 4.

3.3.1 The datasets

Data computed by BloodFlow are stored in a flat file containing information about vertices and tetrahedra. On all our tests, three different datasets have been used, named respectively *Test1*, *Test2* and *Test3*. The first two datasets are examples of real-size study case; *Test3*, instead, is characterized by a smaller mesh. More in detail, the *Test1* mesh is made of 6 million tetrahedra corresponding to 24 millions degrees of freedoms (DoF); *Test2* mesh is made of 3 millions tetrahedra and 12 millions DoF; and *Test3* is made of about 600,000 tetrahedra and 2.4 millions DoF. For the purpose of this paper, the small dataset has been used just for profiling activities while the two larger datasets have been used to study application scalability and performance on different architectures.

Chapter 4

Profiling and tracing activities

Before starting to move both applications to the cloud, an extensive in-depth analysis has been done on both applications. We used several profiling and tracing tools in a black-box way, helping us to spot bottlenecks (intrinsic and external factors) limiting the application performance. Removing such points by applying small adjustment could improve the application performance and scalability. This chapter describes all the results we got running and profiling both applications on Marconi (which is the infrastructure where the applications have been designed to be executed on). The aim of profiling and tracing activities is dual: on one hand we can spot bottlenecks which, once removed, improve the application performance and on the other hand we can get all needed information to build the analytical model, that is described in chapter 4.

4.1 Profiling Cross Motif Search

On Cross Motif Search, profiling and tracing activities have been done not just for harvesting application-related parameters, which will be described and used later in chapter 8, but also for spotting bottlenecks limiting the application performance.

4.1.1 Spin Time and Overhead Time

During our tests, we noticed that, varying the number of threads, cooperating together in order to find recurring geometrical patterns in the secondary structure of a given pair protein, *Spin Time* (that is the wait time during which the CPU is busy because of a synchronization between threads) and *Overhead Time* (that is the time the system takes to deliver a shared resource from a releasing owner to an acquiring owner) varied too accordingly. Both factors were close to zero when the number of threads was equal to 1 and increased with the rising of those. As a consequence, *CPU Time* (the total time spent by all threads) was greater when the protein pair was computed by more threads, and this effect was more evident when the protein couple took less time (because the impact of *Spin Time* and *Overhead Time* was greater). For example, the execution of a couple of proteins with 4 concurrent threads took a CPU Time equals to 4.24 seconds, with a *Spin Time* equals to 18.81% and an *Overhead Time* equals to 6,36% of the *Elapsed Time* (that is the time required to look for recurring geometrical patterns in a single protein pair). Running the same protein pair with a single thread, *Spin Time* and *Overhead Time* fell down close to zero and the *CPU Time* was equals to 2.23 seconds. This analysis explains the behaviour described in the follows, where the pure MPI implementation (256:1) performed better than any other hybrid one. Then, it seems to be quite clear that the OpenMP component is simply a burden, rather than an enhancement.

4.1.2 Heap Contention

We also observed that using OpenMP, the *lib_malloc* function took more *CPU Time* and represented a bound for the application. In the pure MPI version, instead, that function took less time. This increase in the time was due to the high number of heap contentions among the threads. As described in [83], indeed, *malloc* functions allocate a block of memory in the heap. As the heap is shared among all threads, it is necessary to add synchronization to gate access to the shared heap. This is a well-known problem and several solutions have been already described. For example, it is possible to use *kmp_malloc* and *kmp_free* functions in order to maintain a per-thread heap attached to each thread utilized by OpenMP and avoid the usage of the lock that protects access to the standard system heap. In [84], instead, is described *ssmalloc* that is a locality-conscious memory allocator, which can somehow solve the heap contention problem. Unfortunately, using these functions is not always possible, because *malloc* are also used inside external procedures, where there is no control and no possibility to modify source code. Here too, the OpenMP component seems to be a burden, rather than an enhancement.

4.1.3 Wasted Time spent in external function

Using profiling tools, we also discovered a high number of remote cache accesses producing several low level cache misses. Most of them were introduced by the *trim* function in the *boost* library (used in *CrossMotifSearchChain.cpp* at row 50). As explained in *boost* documentation [85], *trim* algorithm is used to remove trailing and leading spaces from a string, where space is recognized using given locales. A locale [86] is an immutable indexed set of immutable facets used for parsing and formatting of all data. Internally, a locale object is implemented as if it is a reference-counted pointer to an array of reference-counted pointers to facets: copying a locale only copies one pointer and increments several reference counts. To maintain the standard C++ library thread safety, both the locale reference count and each facet reference count are updated in thread-safe manner. This places a huge overhead in memory contention, to preserve coherency, and this is paid even if the feature is not used within the application.

4.1.4 Variance in the task computation time

The last interesting aspect we observed profiling the application is that the time spent by the workers to complete a task is not always the same for all protein couples, because it depends on several factors such as geometrical tolerances, maximum size of the motif searched by CMS, similarity score of the proteins in the pair, and size of the proteins in the pair. Indeed, as shown in previous tests [87] the fastest protein pair (*2b1y.xml* and *2hep.xml*) took just 0.000030 seconds, while the slowest one (*1k32.xml* and *1bgl.xml*) took 946.978223 seconds. The average time was 0.2318196 seconds with a variance of 5.96 seconds and a standard deviation equals to 2.44 seconds, ten times greater than the mean time. Although it is not possible to predict the completion time of a task, in [88] we showed that there is a good correlation $\rho = 0.653$ between the size of the protein pair (as the product of the number of secondary structures in both protein files) and the elapsed time. As described in chapter 5, we exploited this observation to increase the load balancing factor.

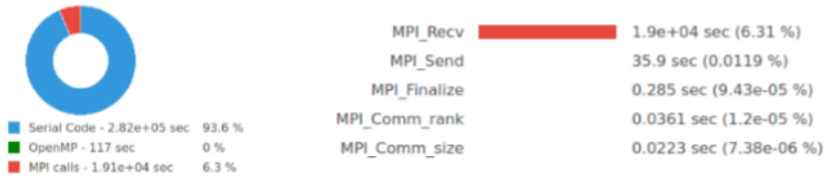

FIGURE 4.1: Time spent by CMS in serial code and inside MPI functions

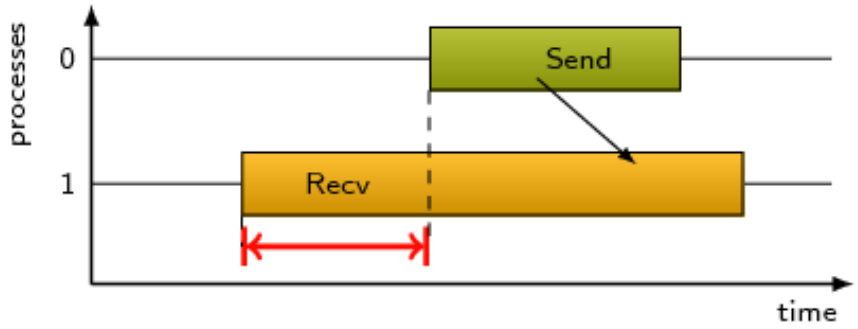

FIGURE 4.2: Time spent by CMS in serial code and inside MPI functions

4.1.5 MPI Communication

Profiling activities showed that Cross Motif Search spends 93.68% of the total execution time inside the application source code and the remaining 6.32% inside MPI calls (Fig. 4.1). The effect of the interconnection facility is not a dominant factor for this application. However, the most active MPI function is the *MPI_Recv* call, where our application has spent nearly all the time, leaving a very small percentage for *MPI_Send* and an even more negligible percentage inside other MPI calls. Due to the amount of time spent inside the MPI functions and the amount of data exchanged among processes, our application cannot be defined as communication-intensive.

Nevertheless, the time spent inside the MPI functions is not representative for the real time spent in communication. This is due to a problem known as *Late Sender* (Fig. 4.2). As described above, indeed, the communication model of Cross Motif Search makes just use of *MPI_Send* and *MPI_Recv* functions, which block process execution until data are completely sent or received. This means that, due to the coarse-grained parallelism of our application (a protein computation might take even several seconds), the master might wait for a long time inside the receiving function, making the time spent inside the MPI functions not representative for measuring the actual communication time, because it includes also the synchronization time. This is the reason why the time-based metrics provided by the profiler are likely to be almost useless for measuring the time spent in communication.

According to the formula 3.1, the profiled amount of messages exchanged among master and workers is as the same as that expected: *n*(n-1)/2* 4B messages sent from the workers to the master and the same amount of 6 KB messages sent back from the master to the workers, where *n* is the number of protein files belonging to the dataset.

4.2 Profiling BloodFlow

As described in chapter 3, instead of using *Test1* and *Test2* datasets, to profile BloodFlow we decided to use a third dataset, named *Test3*, a smaller dataset for a faster execution. The following paragraphs summarize the factors which might affect the application performance, regardless the physical infrastructure where the application is executed on.

4.2.1 Vectorization

The first analysis we did on the application was to make a vectorization analysis to understand the relevance of the floating-point processing, the efficiency of the compiler in detecting and generating vectorized loops, without any action by the programmer, and the degree of architectural optimization possible through AVX-like instruction set. Surprisingly enough, the profiling tools showed that the utilization of the Floating Point Unit is quite low, just 0.11% of CPU time. Moreover, a high percentage (64.49%) of floating-point instructions are scalar, 34.35% of instructions are 128-bit packed FP instructions and just 1.15% are 256-bit packed FP instructions. From the execution point of view, the vectorized loops represent just the 16.3% (32.26 seconds) of the total CPU time (198.09 seconds). The number of the vectorized loops is 63 against a total of 507 loops. Almost half of the vectorized loops (33) have been vectorized using AVX2 instruction set; 27 instead have been vectorized using AVX instruction set, and the last three loops have been vectorized using SSE (one loop) and SSE2 (two loops) instruction sets. There is no gain in 256-bit support, because there is apparently not enough fine-grain data parallelism.

These results highlight that several loops cannot be auto-vectorized, probably because of various factors, among which non-contiguous memory accesses, plus data dependencies. Furthermore, because of the limited number of 256-bit packed FP instructions (just 1.15% of the total floating-point instructions), we speculate that there is no advantage in running the application on Knights Landing architecture making use of AVX512 Instruction set.

4.2.2 Memory footprint

Running the application with the small dataset and using 16 MPI processes running on 16 different cores, the mean quantity of memory used by each process was 1156.97 MB with a peak of 1258.04 MB. Reducing the number of running processes and then the cores number used, the memory allocated by each process grows up. Indeed, running the application with just 4 MPI processes, the mean quantity of memory required by each process becomes 3637.39 MB with a peak of 3787.31 MB. This amount of memory is higher than the RAM available on each core on Marconi, then it seems to be hard to run the BloodFlow on Marconi with a small number of concurrent cores because of the large amount of memory RAM required by the application, unless users decide to reserve a memory size larger than the default size. This can be done according to the billing policy but the application execution becomes more expensive.

4.2.3 MPI Communication

After starting the application, all concurrent MPI processes work together in order to compute the entire dataset. The typical concurrent communication pattern used

TABLE 4.1: Mean number of times each function has been invoked by BloodFlow running on Marconi on different runs.

P2P Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_ISend	3,476	7,007	13,277	24,503
MPI_RSend	17,378	28,578	51,441	84,097
MPI_Send	16,502	24,192	36,824	57,049
MPI_Recv	37,356	59,777	101,514	166,029
MPI_Wait	13,720	18,915	25,596	34,305
MPI_WaitAll	9,266	9,182	9,246	9,195
Collective Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_Barrier	14,241	14,191	14,244	14,518
MPI_BCast	420	776	1,490	2,926
MPI_AllGather	1.67	3.40	9.18	12.89
MPI_AllGatherv	510	510	508	510
MPI_AllToAll	1	1	1	1
MPI_Reduce	15	14	13	12
MPI_AllReduce	34	34	34	34
MPI_Scatter_Reduce	5,769	5,588	5,477	5,516
Other MPI functions	2,192	2,192	2,192	2,192

by the application is an all-to-all communication to scatter and gather data across all processes and this is done by using both point-to-point and collective operations. Indeed, many MPI functions are used by the application.

Table 4.1 shows the complete list of MPI functions (point-to-point as well as collective) used to distribute messages. The numerical values represent the number of times each function has been invoked using four different configurations (from 8 to 64 concurrent MPI processes). These data are related to a single MPI process. This means that, to get the total amount of times each function has been invoked by all processes, the relative value needs to be multiplied by the number of concurrent MPI processes. For example, the total number of *MPI_ISend* invoked by all eight processes is equals to *3,476 * 8 = 27,808*.

As shown in Tab. 4.1, the number of times each point-to-point function is invoked by each MPI process grows when the number of processes rises. For collective operations, instead, the number of times each collective operation is invoked by each MPI process is almost the same for all configurations, except for the function *MPI_BCast*, which increases exponentially. Another important aspect which is worth noting is that the collective function that has been invoked the most in all execution is the *MPI_Barrier*.

Table 4.2 shows the time spent by all processes inside point-to-point and collective functions. It is worth noting that the time depicted in Tab. 4.2 is just the time spent inside each MPI function and should not be considered as the time our application spends in communications, as we shown in [89]. This time might overestimate the real communication time because it also includes synchronization time among processes, network congestion and resource contentions among the processes, making the time spent inside the MPI functions not representative for measuring the communication time.

As shown in Tab. 4.2, the functions where the application spends more time

TABLE 4.2: Time (in seconds) spent by BloodFlow running on Marconi inside each MPI function on different runs.

P2P Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_ISend	0.14	1.04	5.78	22.91
MPI_RSend	2.34	7.76	29.58	130.35
MPI_Send	0.49	2.56	13.58	64.87
MPI_Recv	1.95	6.71	33.65	127.54
MPI_Wait	24.03	62.13	152.52	332.99
MPI_WaitAll	2.27	5.40	15.91	39.54
Collective Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_Barrier	61.72	117.39	218.69	319.90
MPI_BCast	1.19	2.45	5.74	13.32
MPI_AllGather	1.67	3.40	9.18	12.89
MPI_AllGatherv	0.00	0.01	0.05	0.09
MPI_AllToAll	0.10	0.09	0.14	0.22
MPI_Reduce	0.00	0.01	0.06	0.15
MPI_AllReduce	99.24	145.98	211.47	304.66
MPI_Scatter_Reduce	25.00	43.60	59.79	88.75
Other MPI functions	1.17	4.13	16.60	51.02
Observed Time	**221.31**	**402.67**	**772.73**	**1,509.18**

are *MPI_Wait*, *MPI_Barrier*, *MPI_AllReduce* and *MPI_Reduce_Scatter*, and such time grows rapidly when the process number rises. The wide overhead introduced by MPI might reduce performance and application scalability, which will be shown in chapter 6.

To better understand the impact of the MPI functions on each run, Fig. 4.3 shows the ratio among the time spent inside the MPI functions and the time spent outside them. When 64 concurrent MPI processes are used, the time spent inside MPI functions is more than a quarter of the elapsed time, and the ratio rises up when the process number increases.

Another important set of application-related parameters is the amount of data sent and received across all MPI processes. Table 4.3 shows the mean message size (in bytes) sent by all processes on each single MPI operation. To compute the total amount of data sent/received on each operation, the mean size, showed in Tab. 4.3, has to be multiplied by the number of times each operation has been invoked (Table 4.1).

According to the results described above, the application can be defined as tightly-coupled and the MPI communications might affect the application performance. Because of this, we observed that the application seems to not be able to be run in the cloud, where some commercial cloud infrastructure are built on Ethernet interconnections, having much higher latency and lower bandwidth that Infiniband or OmniPath Architecture.

We can also speculate that the application performance will get worse when the application is executed on Marconi's Knight Landing partition because of the higher MPI latency measured on the new Marconi A2 partition for two main reasons: lower CPU frequency, worse intranode and internode latency.

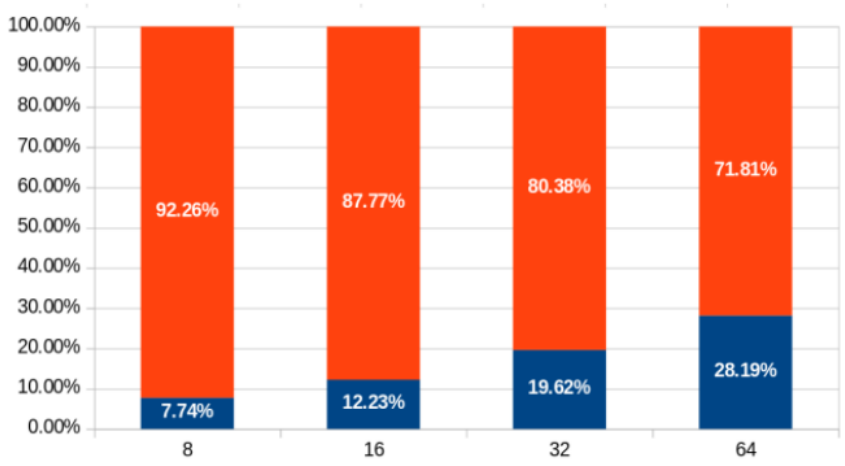

FIGURE 4.3: Time spent by BloodFlow inside and outside MPI functions

TABLE 4.3: Mean message size (in bytes) sent by each process running on Marconi on each single MPI operation.

P2P Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_ISend	5,693	1,824	553	160
MPI_RSend	43,946	21,909	9,393	4,010
MPI_Send	498	293	159	81
MPI_Recv	21,193	10,807	4,889	2,083
MPI_Wait	0	0	0	0
MPI_WaitAll	0	0	0	0
Collective Functions	**8 MPI**	**16 MPI**	**32 MPI**	**64 MPI**
MPI_Barrier	0	0	0	0
MPI_BCast	11,297	6,211	3,276	1,708
MPI_AllGather	107	207	401	793
MPI_AllGatherv	38,784	67,502	127,140	240,845
MPI_AllToAll	131	262	524	1,048
MPI_Reduce	8	8	8	8
MPI_AllReduce	18	25	36	57
MPI_Scatter_Reduce	37	71	138	272

4.2.4 Time consuming functions

Profiling activities also revealed that CPU time is highly consumed by the function *LID* inside the package *Epetra_BlockMap*. This function, that returns local ID of a global ID, takes 418.032 seconds corresponding at 12.43% of the total CPU time (3363,989 seconds). The most time-consuming loop, instead, is inside the function *ConstructFilledGraph* in *Ifpack_IlukGraph* package, that performs the actual construction of the geometry of the mesh. It is a scalar loop consuming 7.4% of the total CPU time. Both functions belong to the Trilinos library.

4.2.5 I/O Bound

The Input / Output does not represent a bound for our application. Indeed, the time spent by all processes waiting for an I/O operation is less than 0.02% of the elapsed time. Each process reads files for an amount of 45.2 MB of data and writes files for an 25.7 KB, just for the log files.

Chapter 5

Application optimization

The profiling process ended with a list of possible point where appropriate updates might increase the application performance reducing the wasted time. By removing such bottlenecks, the application might be executed better not just on the HPC infrastructures but also on the Cloud, which is the main aim of this thesis. In the follows, we describe all the activities we decided to do in order to make both applications more performing.

5.1 Cross Motif Search

Cross Motif Search is a small application making just use of a few external components. Being also co-developed by the author of this thesis, source code update did not require too much time, so we decided to optimize the source code where needed according to the results we got after profiling.

Profiling activities revealed that many intrinsic and external factors make the pure MPI implementation more performing than the hybrid solution, which makes use of OpenMP. Advanced OpenMP optimizations (such as a better memory management) do not seem to change the pattern in a major fashion: the extent of the change is in any case limited to the calls in the application code, while external functions cannot be directly controlled.

For this reason, we decided to keep the current system level software architecture and replace just the *trim* function with an analogous, lighter one, allowing us to reduce the *Elapsed Time*. It is worth noting that poor performance in a trivial function call was only exposed by running the application in a thread-sensitive environment.

Figure 5.1 compares the execution time obtained fixing to 256 the number of the threads simultaneously running, before and after replacing the heavy function (blue and red lines). The results show a not-negligible improvement in performance, but here again, one more time, the fastest execution remains the pure MPI implementation.

5.1.1 Introducing the new selection policy

As described in chapter 3 the load balancing policy used by Hybrid-CMS selects the next protein pair to be processed and assigns it to a free worker process. As also described in chapter 4, there is a great variance in the time spent by each process to compute a single protein pair. When a worker process completes the computation of a received protein pair, it sends a message to the master process requiring another task. If all protein pairs have already been sent to the other processes, the requester waits before finishing off its own execution, doing nothing, until all other processes complete their tasks. When all processes have completed their own computation, all workers and master can release the reserved computing resources (CPU) and

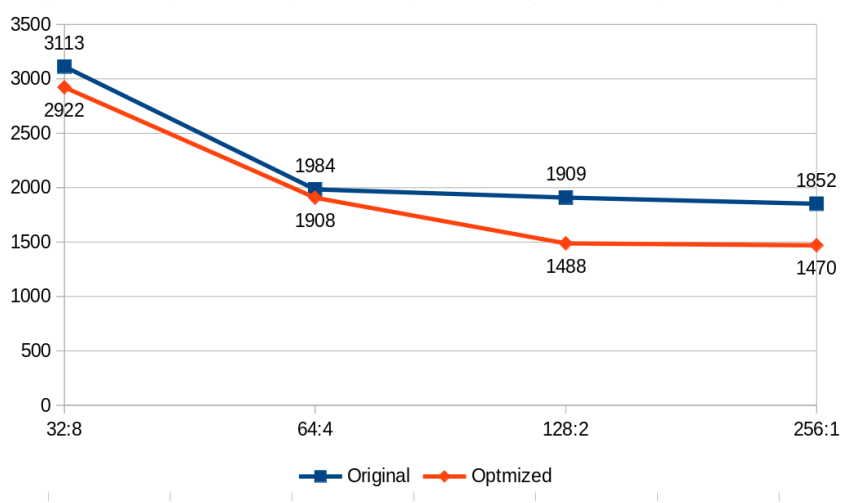

FIGURE 5.1: Cross Motif Search scalability comparison

the application ends. It is clear that the time spent by a worker waiting for the completion of the other workers is just wasted time. So, it is desirable that such time is as small as possible. Reducing the wasted time does not affect the time to complete a single task, but the whole time required to compute the entire dataset, which is the main aim of Cross Motif Search. So, because we are interested in reducing the time required to compute the entire dataset and not the time required for computing just a single protein pair, we aimed to reduce the wait time spent by the processes waiting for other processes, introducing a new load balancing scheme: this turns out to be the Longest Job First policy. The Longest Job First policy has been introduced as a new selection policy used for choosing the next protein pair to send to the worker processes.

5.1.2 Longest Job First: a new policy for selection tasks

The original selection policy used by the hybrid implementation of Cross Motif Search was named Randomly Selected Policy: the next protein pair to send was randomly selected. Using this policy, we discovered [87] that the Global Load Balancing factor [91] was equals to 0.64597: the closer this value to 1, the better the load balancing. Varying the order the protein pairs are sent with, Global Load Balancing changes as well, getting better or even worse. Then the execution time (the time spent by the application to compute the entire dataset) is tightly dependent to the order the protein pairs are sent to the worker processes. This observation led us to introduce a new selection policy, named the Longest Job First: instead of sending protein pair in a random way, the protein pairs are sent according to its expected completion time: protein pairs requiring more time are sent before then the others requiring less time. Even though it is very difficult to get in advance a precise estimate of the elapsed time required by a process to compute a pair of proteins, in [92] it has been noticed that there is a good correlation among the size of the protein pair (as the product of the number of secondary structures in both protein files) and the elapsed time, as described in chapter 4. Then, we decided to modify the application by preparing the list of protein pairs sorted according to the product of the number of secondary structures in both protein files, and then to apply the Longest Job First policy. Running the application with the new selection policy, we measured a

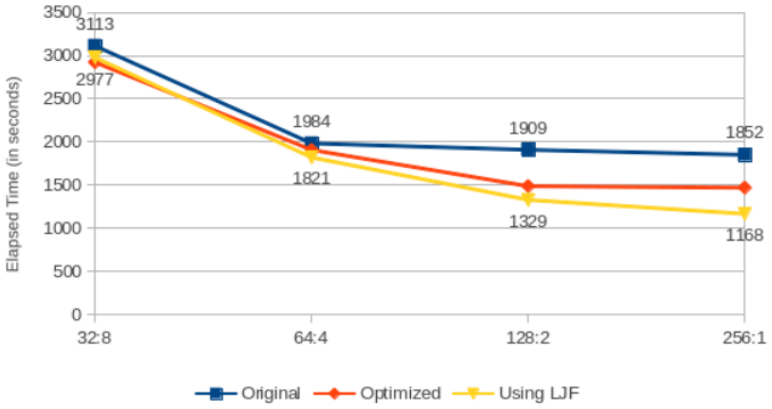

FIGURE 5.2: Cross Motif Search scalability comparison after introducing Longest Job First policy

Global Load Balancing factor of 0.99976 [88], which is higher than that observed with the previous policy. We also implemented a third different policy to submit protein pairs, the Shortest Job First policy, but even this approach yields worse results. The Global Load Balancing factor indeed was equals to 0.57905.

5.1.3 Application performance after introducing Longest Job First policy

Figure 5.2 compares the time spent by Cross Motif Search using different configuration. The blue line represents the original implementation of the hybrid CMS; the red line instead is the elapsed time we got running the optimized version; the yellow line instead represents the time spent by CMS after introducing the Longest Job First policy. Here again, one more time, the pure MPI solution (represented by the run labelled with *256:1*) yields the best result.

5.1.4 Side effects of the Longest Job First policy

The load balancing algorithm used by Hybrid-CMS for delivering protein pairs is on-demand. Anticipating the computation of the longest protein pairs (and then deferring the shortest protein pairs to the end of the execution) yields an increase in communication between master and workers processes at the end of the application execution. This behaviour might affect the performance of the application when the number of MPI processes simultaneously running is high and the interconnection bandwidth is not enough large to support the increase in communication. Moreover, if the application makes use of just one master process for delivering proteins pairs, the time spent by each worker waiting for the next protein pair might increase sensitively. These side effects have been confirmed by the profiling.

5.1.5 When LJF is winning

The Longest Job First policy is just one of the several policies that can be used for selecting tasks. Others can be found in [93–96]. The effectiveness of each policy is tightly connected to the specific application domain and a policy achieving good performance in a context might yield bad results in a different context. In this paragraph, we want to summarize all factors and constraints that might make the Longest Job First policy winning:

- There is a large variance of the elapsed time during the computation of each possible task;
- All tasks are independent: there is no communication between two processes computing two different tasks;
- All tasks have the same priority: a task is never more important than any other;
- All tasks are known before the computation starts: in Cross Motif Search, indeed, the dataset is read at the startup and all possible combinations among all protein files are known when the application starts;
- The aim is reducing the total computation time and not the time of a single task;
- The tasks cannot be suspended or interrupted;
- Even though a precise estimation of the elapsed time required by a process to complete the task is difficult or impossible, a heuristic about the presumed time required to complete the task has to be known.

5.1.6 The location-aware implementation

When we described cloud infrastructure and HPC system in chapter 2, we classified communications in intranode and internode, depending on the node where both processes involved in the communication belong to. Internode communications are more expensive than the intranode ones, because the messages are sent through the fabric while the intranode communications are usually solved through shared memory. It becomes clear that reducing the internode communications might bring along better performance most over for those architectures, such as the cloud, where the ratio between intranode and internode communication is higher. In order to reduce the time spent by the application for sending and receiving messages, we have identified a new method for distributing the tasks among the processes that minimize the internode communication increasing the intranode one. This new improvement exploits the coarse-grain granularity of all tasks and makes Cross Motif Search application able to be successfully executed on the Cloud Infrastructure. The new, last implementation adopts a new task selection policy which takes into account both predicted completion time of a protein pair and the location of the worker within the virtual cluster.

In this new Cross Motif Search implementation, named location-aware CMS [90], before starting the protein comparison, master and workers establish a handshake, which is the process of negotiation between two participants (master and workers) before they start sending useful information. Each worker sends to the master the position where it is running. This information is stored in a table and afterwards used by the master to understand whether or not the worker belongs to its same instance. After completing the handshake phase, the computation phase starts. As in the previous version, each worker sends to the master a 4-byte message for getting the next task. The master, instead of sending back the next protein pair, inquires about the position of the requiring worker. If the worker belongs to the same virtual instance as the master, the protein pair having the lowest predicted computation time is sent back. Otherwise, the master returns a protein pair having the highest predicted computation time.

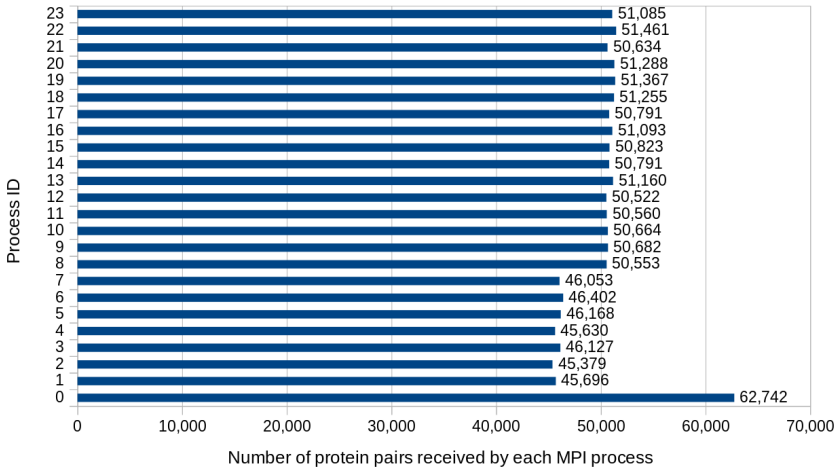

FIGURE 5.3: Number of protein pairs received by each MPI process before introducing LJF

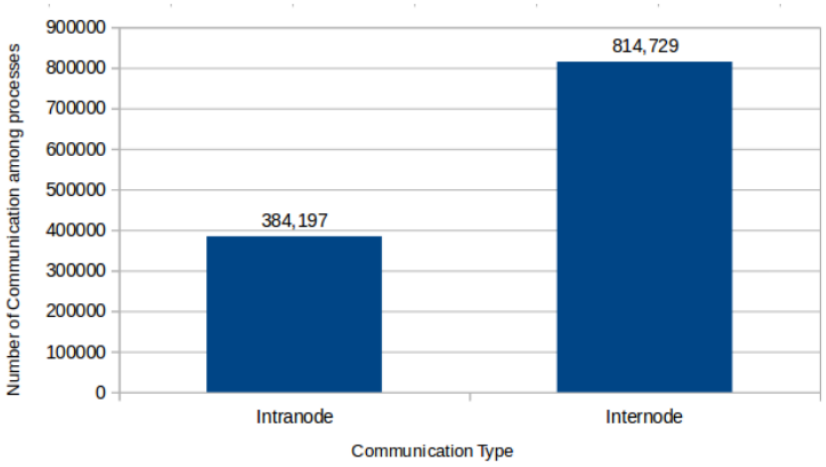

FIGURE 5.4: Total number of intranode and internode messages sent by CMS master before introducing LJF

The rationale of the policy is to reduce as much as possible the communication overhead, by placing short, frequent tasks close by to the sending master. As in the previous implementation, the master spawns into two threads: the first, primary master, is responsible to the communication; the secondary master instead communicates with the primary master by shared memory and always computes the protein pair having the lowest predicted computation time, as the other intranode workers.

Before introducing the locality-awareness policy, the tasks were almost uniformly distributed across all processes. Figure 5.3 shows the number of tasks received by each process in a previous test. As the scheduling algorithm is on-demand, it is impossible to predict in advance how many protein couples a process might receive and this amount changes in different runs, but it is easy to note that the tasks were distributed quite uniformly among all processes.

By summing up the amount of tasks received by each worker according to the infrastructure configuration, it is possible to compute the amount of intranode and internode communications, showed in Fig. 5.4.

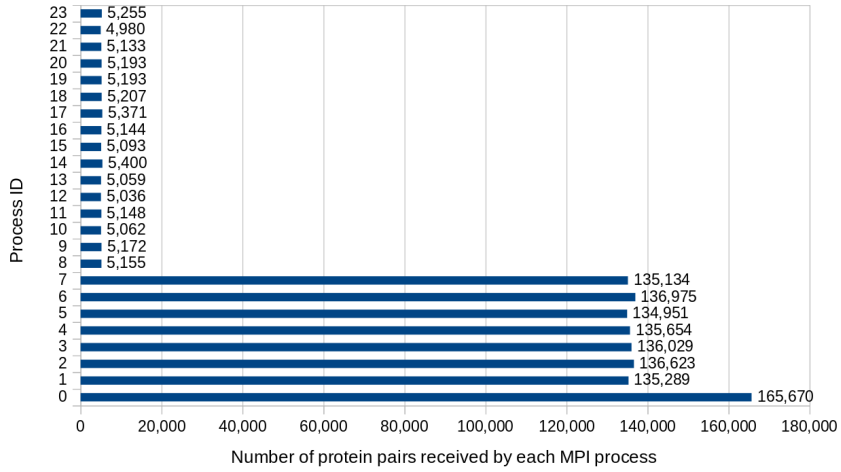

FIGURE 5.5: Number of protein pairs received by each MPI process after introducing LJF

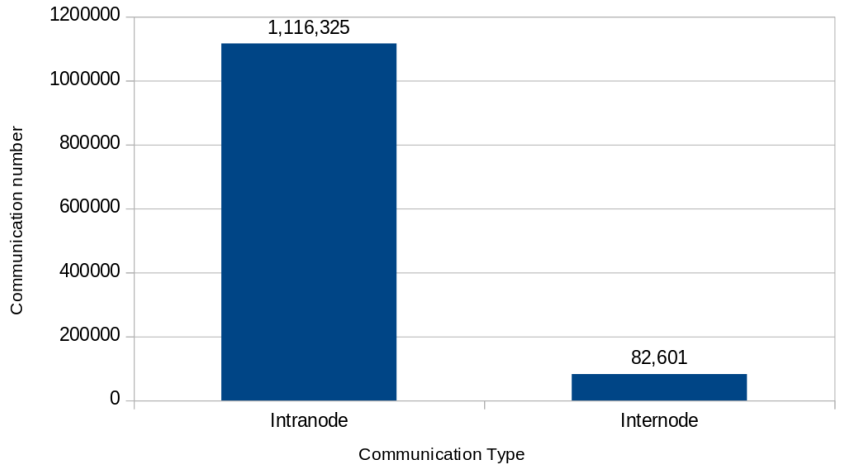

FIGURE 5.6: Total number of intranode and internode messages sent by CMS master after introducing LJF

The number of internode communication is more than twice than the intranode one. This behaviour might be considered negligible for applications running on a real HPC system, such as on Marconi, where the ratio among intranode and internode communication is quite low, but for the Cloud Infrastructure the unbalance among intranode and internode communication might compromise the application performance.

With the location-aware policy, the internode communications have been heavily reduced, as shown in Fig. 5.5. Using the new selection policy, indeed, the number of intranode communication is much higher than the intranode one, as shown in Fig. 5.6. In the optimized implementation, the intranode communication count is more than 13 times higher than the internode one.

5.1.7 Global Load Balancing Factor of the location-aware implementation

The new selection policy changes completely the strategy used by the master to send tasks to the workers and this might bring to the conclusion that an imbalance among

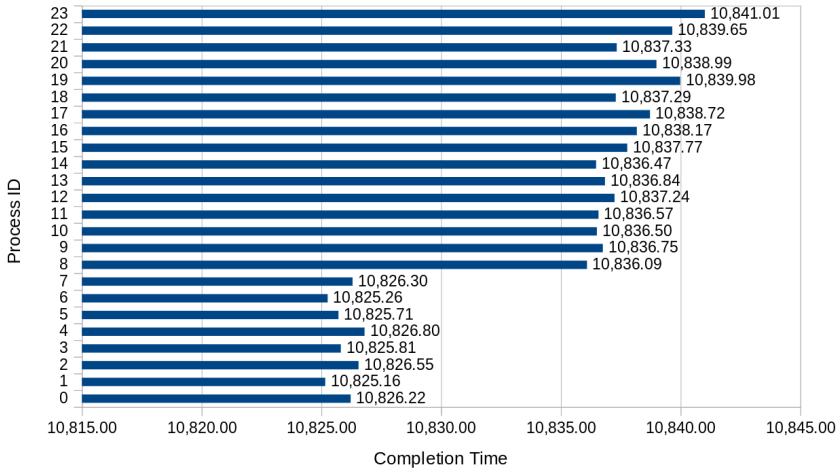

FIGURE 5.7: Completion time of the 24 MPI processes used to run CMS in the location-aware implementation

processes might arise. To compare the effectiveness of the optimized implementation with respect to the previous one, we computed the Global Load Balancing Factor for the last implementation, taking into account the time for each process to complete its execution.

Figure 5.7 shows the final section of the execution of all 24 MPI processes used in our test. Depending on the time last task was received by each process, the completion time can be different. However, also in the location-aware implementation, all MPI processes are well balanced, and the Global Load Balancing Factor is still very close to the ideal value:

$$LB = \frac{avg(T_p)}{max(T_p)} = \frac{10833.8835}{10841.0137} = 0.9993423 \tag{5.1}$$

5.2 BloodFlow

The profiling process we did on BloodFlow highlights that the application is not able to exploit all the features provided by the physical architectures. Improving application performance is then possible but it requires an extensive activity which might involve many libraries, such as the Trilinos Library, which are beyond our scope. The optimization process is then not effortless and might take a huge amount of time. As the aim of this thesis is not the application optimization but just understanding whether or not this application can be successfully executed on the cloud, we decided to leave the application as it is and go further, building the analytical model, described in the chapter 7, to get insight about how the application might perform in the cloud.

Chapter 6

Application Scalability

6.1 Cross Motif Search

6.1.1 Scalability and performance on Marconi

To study scalability and performance of Cross Motif Search, we used the last optimized version of CMS based on the Longest Job First policy and the location-awareness, described in chapter 5. Being OpenMP a burden rather than an enhancement, as many time highlighted by the results showed in chapter 5, we also decided to get rid of such technology, studying scalability and performance just for the pure MPI solution.

Figure 6.1 shows the time spent by the pure-MPI solution varying the number of MPI processes from 16 to 256. As shown, the scalability of CMS is very high: doubling the number of concurrent processes, the execution time is halved.

6.2 BloodFlow

6.2.1 Scalability and application performance on Stampede and Comet

In previous works [97–99], scalability and application performance have been first measured on Stampede and Comet and using the largest dataset described in chapter 3. As the authors were not interested in the biomedical significance of the results, instead of considering the total application execution time, in order to study application performance and scalability, they decided to run only the first 10 time steps of the 1000 whole simulation (each step extends over 1 ms); in the following, *execution time* is the average of a single time step. Since most common clinical applications need at least of three simulated heartbeats, the total necessary execution time is at least three thousand times longer than what we refer as *execution time* in this thesis. The execution time is split over the three most costly operations performed during a time step: the assembly of the finite element matrix (assembly); the building of the preconditioner of the linear system (preconditioning) and the solution of the preconditioned linear system (solving).

Figures 6.2 and 6.3 show the application scalability on both platforms and using different core numbers. Results about the time spent by application running on Stampede and using 48 and 96 cores are missing because the application experienced a crash during the runs. According to the results we got profiling and tracing the application we speculate that the reason is mainly due to the high amount of memory required by the application when a small number of CPU are used.

Comparing figures 6.2 and 6.3, BloodFlow seems to have the same asymptotic behaviour on both architectures but it stops scaling at 384 cores on Comet while on Stampede it is able to scale up to 768 cores. On both architectures, increasing more

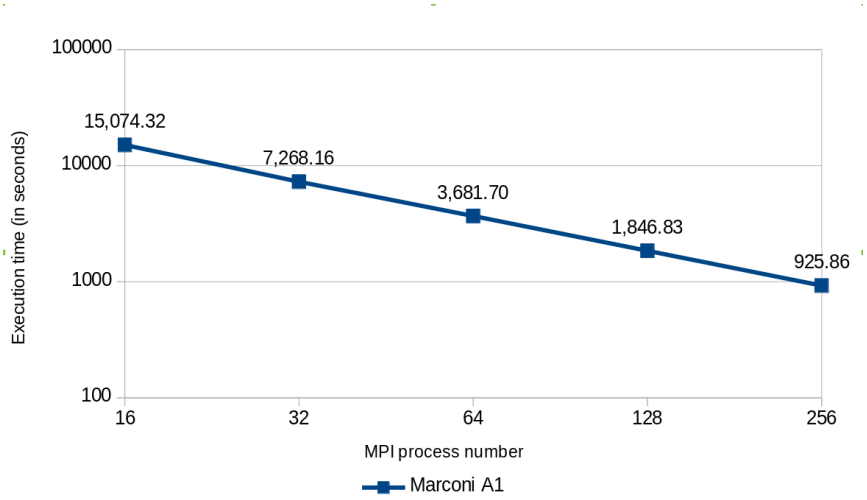

FIGURE 6.1: Cross Motif Search Scalability on Marconi

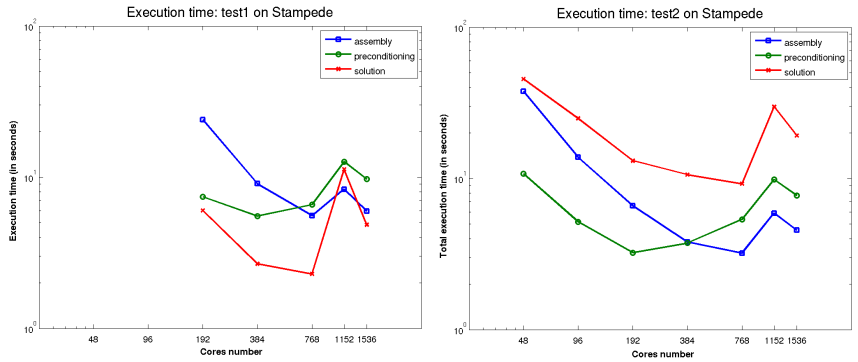

FIGURE 6.2: BloodFlow Scalability on Stampede

the core numbers beyond these limits, the execution time rises up. Furthermore, we did not observe any advantage in using the AVX2 instruction set on Comet over AVX only on Stampede. Even this behaviour can be justified according to the results we got profiling the application: the application showed a limited number of 256-bit packed FP instructions (just 1.15% of the total floating-point instructions). Furthermore, computation on Stampede is faster than on Comet; apparently, this might be explained by the fact that the CPUs in Stampede have a faster clock (2.7Ghz) than in Comet (2.5Ghz), as shown in Tab. 6.1.

6.2.2 Scalability and application performance on Marconi

Figure 6.4 shows the scalability results we got running the application on Marconi, using the largest two datasets and varying the number of concurrent MPI processes from 36 to 2,304. Although the results seem to be more regular than on Stampede and Comet, even on Marconi the application showed the same behaviour: the application is able to scale very well up to a specific configuration. Increasing more the core number, the execution time rises up.

Figure 6.5 compares all three different architectures [101]. Although the CPU clock frequency on Marconi is lower than on Stampede and on Comet, as shown in

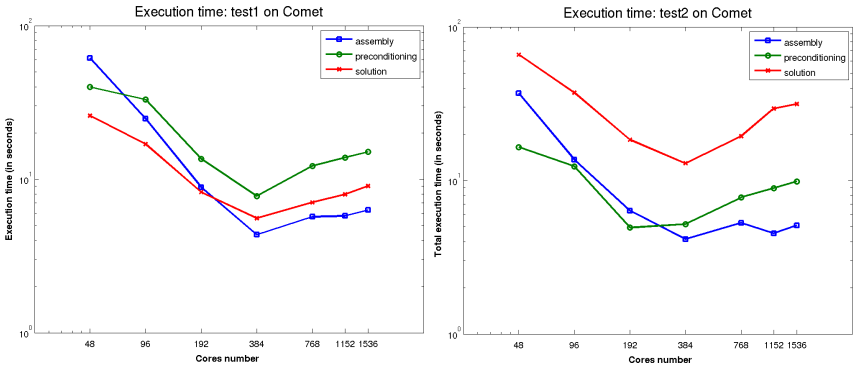

FIGURE 6.3: BloodFlow Scalability on Comet

TABLE 6.1: HPC system configuration comparison.

	Marconi A1	**Stampede**	**Comet**
Nodes	1,512	6,400	1,94
Processor	Intel Xeon E5-2697	Intel Xeon E5-2680	Intel Xeon E5-2680
CPU Frequency	2.30 GHz	2.70 GHz	2.05 GHz
Architecture	Broadwell	Sandy Bridge	Haswell
Procs per node	2	2	2
Cores per proc	18	8	12
RAM per core	3.50 GB	4.00 GB	5.30 GB
Network	100 Gb/s		

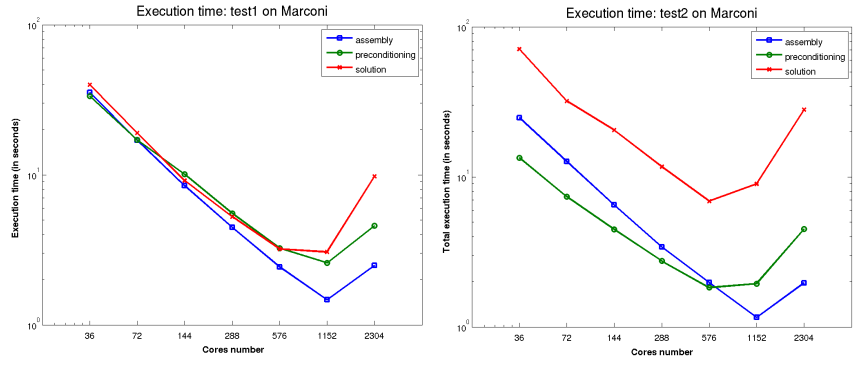

FIGURE 6.4: BloodFlow Scalability on Marconi

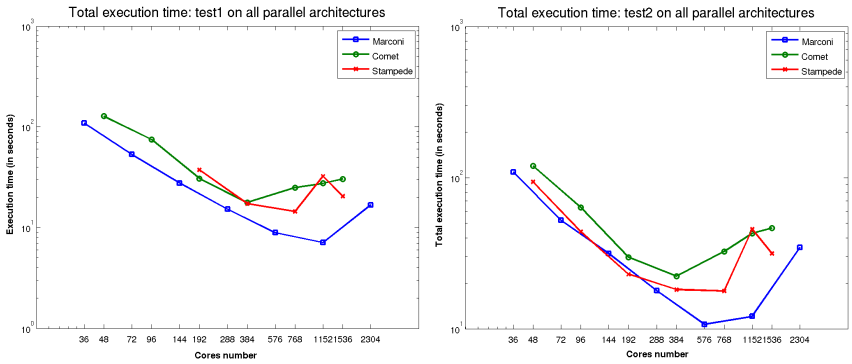

FIGURE 6.5: BloodFlow Scalability all three HPC systems

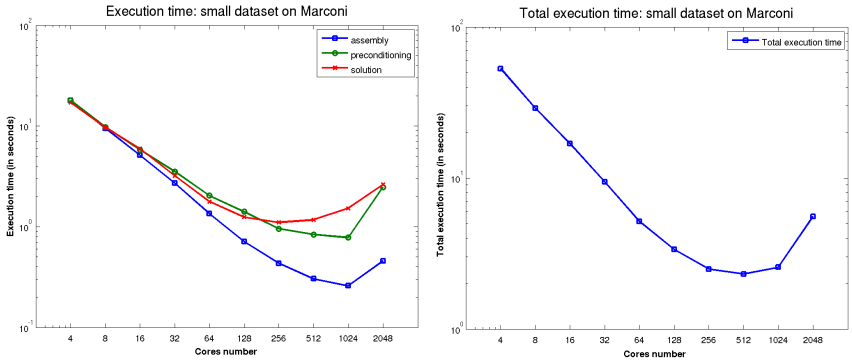

FIGURE 6.6: BloodFlow Scalability on Marconi using the small dataset

Tab. 6.1, the application performs better on Marconi, where the application is able to scale up to 1,152 cores and 576 cores for *test1* and *test2* respectively.

Figure 6.6 shows the mean execution time obtained for each main phase (on the left) and the total execution time (on the right) when the smaller dataset has been used, varying the number of concurrent MPI processes from 4 to 2048.

The behaviour is similar to that observed using the two largest dataset. Indeed, even using the small mesh, the application seems to be able to scale up to fixed point, 512 cores, then, increasing more the concurrent process number, the total execution time rises up. Clearly, the problem size (dimension of the mesh, in *test3* it is 1/5 of the smallest between *test1* and *test2*) has no influence on scalability.

The first attempts to run the application on Marconi using the small dataset and just 4 processes did not end well. The application indeed crashed after being executed. A behaviour similar to those observed running the application on Stampede and using 48 and 96 cores. The explanation seems to be strictly related to the amount of memory available in the system. As shown in chapter 4, running the application with a small number of concurrent cores requires an amount of memory RAM higher than the amount of actually available in the system, making the application unable to run. Anyway, the billing policy on Marconi [100] allows the user to reserve a higher amount of RAM but with a higher cost. Then, just for the first run shown in 6.6 (4 MPI processes), the amount of memory used for that run has been 4.5 GB per process (instead of 3.5 GB for the other runs).

6.2.3 Factors making Marconi more efficient

The previous paragraph showed that although Marconi relies on less powerful processors, the application performs better than on the others. Table 6.1 briefly compares the configuration of all three different HPC infrastructures, but we spotted four important factors making Marconi more efficient than the other two systems:

- **Lower MPI latency**: on Marconi the intranode MPI latency is 0.5 microseconds (against 1.03 microseconds for Stampede and Comet) while the internode MPI latency is 1.1 microseconds (against 1.97 microseconds for Stampede and Comet);
- **Higher Bandwidth**: on Marconi the interconnection bandwidth is 100 Gb/s against 56 Gb/s on Stampede and Comet;
- **Larger RAM for each node**: on Marconi each core has got up to 3.5 GB against 2.0 GB for Stampede (Comet instead has got a higher amount of RAM, that is 5.3 GB);
- **Larger L3 cache**: on Marconi all cores in the same node share a 45 MB L3 cache against 20 MB for Stampede (no information about Comet);

6.2.4 Understanding the lack in scalability

The reason of the lack in scalability for BloodFlow, now, is quite clear. As described in chapter 4, the application communication model is quite complex and the time spent inside the MPI functions gets higher when the number of concurrent MPI processes rises up. Then, increasing the number of MPI over a specific configuration, the elapsed time starts to grow. What has been observed studying the application scalability is compatible with that observed profiling the application with a different number of concurrent MPI processes. Indeed, as shown in chapter 4, increasing further the number of concurrent MPI processes, the number of invoked functions and the time spent inside of them rises up heavily, becoming quickly higher than the time spent by all processes in computation, and when this happens, there is no convenience in running the application with a higher number of cores. It becomes clear that the interconnection network plays a major role.

Chapter 7

Building the analytical model

As already described, the main aim of this thesis is to understand if moving two different scientific applications (BloodFlow and Cross Motif Search), respectively based on a complex and on a simple communication pattern, is worth doing it or not from both economical as well as performance perspective. When considering the porting of an MPI application from a fabric-based infrastructure to the cloud environment, the pattern of communication as well as the interconnection network are of paramount importance, because both of them are the most important factors limiting the performance of many parallel scientific applications. As moving any application to a different infrastructure is not effortless and can take a huge amount of time, instead of moving straight away CMS and BloodFlow to the cloud, we decided to do a deep analysis to predict the application performance and understand if the porting is worth doing it.

In this chapter we want to build an analytical model which is able to describe the complexity of the communication pattern where the applications are based and then provide insights about how the applications will behave being run on a different platform. By looking at such models, analysts may understand whether or not it is worth moving the application on that platform. To build a proper model, parameters describing the communication model (application-related parameters) as well as the network infrastructure (network-related parameters) need to be gathered. Application-related parameters describe the application and its execution, and they are always the same regardless the infrastructure layer where the application is executed. When the communication model is well-known, application-related parameters can be gathered by looking up the documentation. Other times, instead, most over when application has been designed and built by others, information about the communication pattern used by processes to send and receive messages has to be harvested in a black-box manner, using, for instance, profiling and tracing tools. For gathering network-related parameters, several models can be used, as described in Appendix B, but for our experiments we decided to use pLogP [102] model, which has already been used in chapter 2 to characterize the Marconi and Cloud interconnection network. Differently from the application-related parameters, which are always the same regardless the infrastructure where the application is executed on, the network-related parameters are strictly related to the network infrastructure and need to be gathered on all different network layers. Table 7.1 shows the list of application-related parameters and network-related parameters we decided to use to build the analytical model. It is worth noting that all these parameters can be gathered by knowing the communication model or by inspecting the application execution using profiling tools. All application related parameters for both applications are described in chapter 3, while the network related parameter for Marconi and the cloud interconnection networks are described in chapter 2.

By combining together both parameter sets, an accurate analytical model can be

TABLE 7.1: Application-related parameters required to build the analytical model.

Application-related parameters
Number of concurrent MPI processes
MPI functions used by the application
Number of times each MPI function is invoked
Message size sent/received
How these parameters change varying the number of concurrent processes
Network-related parameters
Number of concurrent MPI processes
Intranode and internode communication number
Network latency (L parameter, see PLogP)
Network Bandwidth (g parameter, see PLogP)
PLogP model for collective functions

built and this can help to understand how much the interconnection layer affects the application performance. Extending the pLogP model (see formula B.2) for both intranode and internode communications and for all message sizes, the analytical model for computing the time spent to send a message in a point-to-point communication can be built as follows:

$$\begin{gathered} T = T_{intra} + T_{inter} \\ \text{where} \\ T_{intra} = n_{intra} * (L_{intra} + g_{intra}(M)) \\ T_{inter} = n_{inter} * (L_{inter} + g_{inter}(M)) \end{gathered} \tag{7.1}$$

where n is the number of times the sending function has been invoked, L and g are the pLogP parameters, representing respectively the latency and the minimum time interval between consecutive message transmissions or receptions, and M is the message size.

The model described so far can be applied just for point-to-point functions. For collective functions instead that model can be still applied, but it has to be extended, according to the implementation of the collective functions (see Table B.1). To be clearer, let's suppose we are interested about building the analytical model for computing the time spent by an application to perform the broadcast operation. Let's also suppose that the number of concurrent MPI processes is N and they are grouped into k different clusters. As described in Appendix B, the way the message is sent among processes depends on the function implementation. To make the things easier, we can suppose, without losing in generality, that the broadcast operation is implemented with the linear algorithm. This means that the root process sends the message to each of the *N-1* receivers, one at time. This means that the number of intranode communications is *1/k* of the total communication, while the internode communication is *(k-1)/k* of the total communications. According to the pLogP model for the broadcast implementation, the analytical model for the broadcast function is the follows:

$$
\begin{aligned}
T &= T_{intra} + T_{inter} \\
T_{intra} &= n_{intra} * ((N-1) * g_{intra}(M) + L_{intra}) \\
T_{inter} &= n_{inter} * ((N-1) * g_{inter}(M) + L_{inter}) \\
n_{intra} &= \frac{1}{k} * n \\
n_{inter} &= \frac{k-1}{k} * n
\end{aligned}
\tag{7.2}
$$

where M is the message size, N is the process number, n is the number of times the broadcast function has been invoked, k is the number of clusters. As an application usually makes use of several collective functions, for each of them the corresponding analytical model needs to be built. At the end, all analytical models are put together into the final formula:

$$
PCT = \sum_i n^i_{intra} * f^i_{intra} + \sum_i n^i_{inter} * f^i_{inter} \tag{7.3}
$$

where f^i is *i-th* MPI function, n^i_{intra} is number of times the *i-th* MPI function has been invoked by a process for intranode communication and n^i_{inter} is number of times the *i-th* MPI function has been invoked by a process for internode communication.

The Predicted Communication Time (PCT) formula is able to compute the predicted communication time for an application running on a target platform.

After having described how to build in general the analytical model to compute the predicted communication time, the following paragraphs show the corresponding values for the application-related parameters and network related parameters for both applications gathered on Marconi and on the cloud infrastructure.

7.1 Cross Motif Search

7.1.1 Application-related parameters

The communication model where Hybrid Cross Motif Search relies on, is very simple. As already described in chapter 3, the application makes use of just two primitives (*Send* and *Receive*), and there is no communication among workers. Regardless the number of concurrent processes used to compute the whole dataset, the total amount of messages exchanged between master and workers is always the same, and, as shown in the formula 3.1, it can be computed in advance, as it depends just on the number of protein files belonging to the dataset. Table 7.2 shows the application related parameters we got profiling the application on Marconi and using 24 concurrent MPI processes (three node with 8 cores). Here we represent only one configuration (24 cores) because the others were very close to this, as the number of messages exchanged among processes is always the same and the small variation in the time spent inside the MPI functions can be considered as negligible and does not affect the application-related parameters used to build the analytical model.

7.1.2 Network-related parameters

To build the analytical model and then predict the communication time spent by the application, both application-related parameters and network-related parameters need to be combined together. Differently from the application-related parameters, which depend only on the application and are always the same given a fixed

TABLE 7.2: Application-related parameters required to build the Cross Motif Search analytical model.

Number of concurrent MPI Processes	24		
MPI functions used by the application	*Send / Receive*	**Message size**	**Message number**
		4 B	1,198,926
		6 KB	1,198,926
How these parameters change varying the number of concurrent processes	The number of messages exchanged among master and workers is fixed as it does not depend on the number of concurrent processes		

TABLE 7.3: Network-related parameters gathered on Marconi.

Number of nodes	3	
Number of cores per node	8	
	Internode	**Intranode**
Number of 4-byte messages	82,601	1,116,325
Number of 6-KB messages	82,601	1,116,325
L	2.50E-06	4.00E-07
g(4-B)	1.00E-06	6.00E-07
g(6-KB)	2.60E-06	1.50E-06

configuration, the network-related parameters are strictly dependent on the underlying infrastructure layer and need to be gathered for all different infrastructures where the application runs.

In chapter 2, we showed all network-related parameters for different message sizes gathered on Marconi and on the Cloud, while chapter 5 gave a description about the number of intranode and internode messages are sent using the location-aware implementation, when the application was executed with 24 processes running on three different nodes. Although the scheduling algorithm is on-demand and then the number of tasks received by a worker might change in different runs, in order to build the analytical model we can suppose such number as invariant and always the same regardless the number of concurrent processes, but supposing fixed the number of clusters.

Tables 7.3 and 7.4 summarize all network related parameters required to build the Cross Motif Search Analytical Model for both infrastructures.

7.1.3 Cross Motif Search Analytical Model

According to the formula 7.3, the analytical model for Cross Motif Search for both infrastructure can be built as following:

TABLE 7.4: Network-related parameters gathered on Google Cloud.

Number of nodes	3	
Number of cores per node	8	
	Internode	**Intranode**
Number of 4-byte messages	82,601	1,116,325
Number of 6-KB messages	82,601	1,116,325
L	3.41E-05	1.00E-07
g(4-B)	1.42E-05	3.00E-07
g(6-KB)	3.19E-05	2.30E-06

$$PCT_{Marconi} = n_{inter} * f_{inter}^{send} + n_{intra} * f_{intra}^{send} =$$

$$\begin{gathered} = n_{inter} * [L_{inter} + g_{inter}(4B)] + n_{inter} * [L_{inter} + g_{inter}(6KB)] + \\ + n_{intra} * [L_{intra} + g_{intra}(4B)] + n_{intra} * [L_{intra} + g_{intra}(6KB)] = \\ 82,601 * (2.50E-06 + 1.00E-06) + \\ + 82,601 * (2.50E-06 + 2.60E-06) + \\ + 1,116,325 * (4.00E-07 + 6.00E-07) + \\ + 1,116,325 * (4.00E-07 + 1.50E-06) = 3.95\ s \end{gathered} \tag{7.4}$$

$$PCT_{Cloud} = n_{inter} * f_{inter}^{send} + n_{intra} * f_{intra}^{send} =$$

$$\begin{gathered} = n_{inter} * [L_{inter} + g_{inter}(4B)] + n_{inter} * [L_{inter} + g_{inter}(6KB)] + \\ + n_{intra} * [L_{intra} + g_{intra}(4B)] + n_{intra} * [L_{intra} + g_{intra}(6KB)] = \\ 82,601 * (3.41E-05 + 1.42E-05) + \\ + 82,601 * (3.41E-05 + 3.19E-05) + \\ + 1,116,325 * (1.00E-07 + 3.00E-07) + \\ + 1,116,325 * (1.00E-07 + 2.30E-06) = 12.57\ s \end{gathered} \tag{7.5}$$

Summarizing, the predicted time spent by Cross Motif Search in communication is equal to 3.95 seconds on Marconi and 12.57 seconds on the cloud.

7.1.4 Speculation of the execution on the cloud

Analysing the analytical model, many interesting observation can be raised:

- The analytical model does not depend on the number of concurrent MPI processes, as the whole amount of messages sent and received depends only on the number of protein file belonging to the dataset. Keeping fixed the dataset and varying the number of concurrent MPI processes, the communication is always the same;
- The impact of the communication on the Cloud is three times higher than the communication on Marconi;

TABLE 7.5: PLogP model for collective operations used by BloodFlow.

MPI Function	Algorithm	pLogP Model
Barrier [115]	Double Ring	$T = 2*P*(L+g(0))$
Broadcast [115, 124, 125]	Linear	$T = L + (P-1)*g(m)$
AllGather [115]	Ring	$T = (P-1)*(L+g(m))$
AllToAll [115]	Pairwise exchange	$T = (P-1)*(L+g(m))$
Reduce [115]	Flat Tree	$T = L + (P-1)*g(m)$
AllReduce [115]	Recursive Doubling	$T = (log_2P + 2)*(L+g(m))$
ScatterReduce	Ring	$T = (P-1)*(L+g(m))$

- The predicted communication time we got using pLogP is not comparable to the time spent by the application inside the MPI functions, revealed by the profiler, as the pLogP measures only the real communication time, while the time spent inside MPI functions can be affected by many factors such as synchronization time among processes, network congestion and resource contentions among the processes, making the time spent inside the MPI functions not representative for measuring the communication time;
- If compared with the CPU time, which is shown in Fig. 6.1, the predicted communication time for both architecture is very small and does not affect at all the application performance even being run on the cloud infrastructure.

7.2 BloodFlow

7.2.1 Application-related parameters

By using tracing tools, we have been able to gather all the required application-related parameters varying the number of concurrent processes from 8 to 64. All of them have been widely described in chapter 4.

7.2.2 Network-related parameters

The pLogP parameters describing Marconi and Cloud network infrastructures have already been gathered and presented in chapter 2. As also showed in chapter 4, BloodFlow relies not only on the point-to-point functions but also on the collective operations, and both of them should be kept into account during prediction. Predicting the communication time for collective operation is not so immediate, because it is important to consider the internal implementation of each functions, as described in Appendix B. Different algorithms, indeed, yield different performance and the number of intranode or internode messages exchanged in a collective function can be different depending on the implementation. Table 7.5 shows a small list of collective function, used in the prediction. The algorithm name nearby each MPI function represents the name of the implementation of the function. A wider list of collective functions can be found in Appendix B.

Although here we don't want to go through a complete description of the function implementation, Tab. 7.6 shows, for each collective function, the amount of intranode and internode communication, supposing to use cluster made of k nodes each of them equipped with c cores.

TABLE 7.6: Amount of intranode and internode communication for each MPI function on Marconi.

	Internode	**Intranode**
MPI_Isend	(k-1)/k * N_{MPI_Isend}	1/k * N_{MPI_Isend}
MPI_Rsend	(k-1)/k * N_{MPI_Rsend}	1/k * N_{MPI_Rsend}
MPI_Send	(k-1)/k * N_{MPI_Send}	1/k * N_{MPI_Send}
MPI_Barrier	k/c * $N_{MPI\_Barrier}$	(c-k)/c * $N_{MPI\_Barrier}$
MPI_Bcast	(k-1)/k *N_{MPI_Bcast}	1/k * N_{MPI_Bcast}
MPI_Allgather	k/c * $N_{MPI\_AllGather}$	(c-k)/c * $N_{MPI\_AllGather}$
MPI_AllgatherV	k/c * $N_{MPI\_AllGatherV}$	(c-k)/c * $N_{MPI\_AlGatherV}$
MPI_AllToAll	(k-1)/k *$N_{MPI\_AllToAll}$	1/k * $N_{MPI\_AllToAl}$
MPI_Reduce	(k-1)/k *N_{MPI_Reduce}	1/k * N_{MPI_Reduce}
MPI_AllReduce	(k-1)/k *$N_{MPI\_AllReduce}$	1/k * $N_{MPI\_AllReduce}$
MPI_Scatter	1/k * $N_{MPI\_Scatter}$	(k-1)/k *$N_{MPI\_Scatter}$

7.2.3 BloodFlow Analytical Model

We are now ready to build the analytical model for computing the Predicted Communication Time for BloodFlow. By combining the data showed in Tab. 2.1, representing the pLogP parameters on Marconi, in Tab. 4.3, representing the mean message size for all point-to-point and collective operations, and in Tab. 4.1, representing the number of times each function is invoked, we can estimate the actual time spent by our application for all communications. Just for example, we show how to compute the predicted communication time only for the MPI_Reduce function, giving for the others just the final result.

$$\begin{aligned} PCT_{MPI\_Reduce} = n^{inter} * f^{inter}_{MPI\_Reduce} + n^{intra} * f^{intra}_{MPI\_Reduce} = \\ \frac{(k-1)}{k} * N_{MPI\_Reduce} * ((k*c-1) * g^{inter}(m) + L^{inter}) + \\ \frac{1}{k} * N_{MPI\_Reduce} * ((k*c-1) * g^{intra}(m) + L^{intra}) \end{aligned} \tag{7.6}$$

where k is the number of nodes, c is the number of cores per node, g and L are the pLogP parameters, m is the message size, N is the number of times the function has been invoked. Supposing to have a cluster made of 4 different nodes k each of them equipped with 8 cores c, the predicted communication time for the MPI_Reduce on Marconi can be computed as follows:

$$\begin{aligned} PCT_{MPI\_Reduce} = [(4-1)/4 * 34 * ((4*8-1) * 1.20E-06 + 2.50E-06)] \\ +[1/4 * 34 * ((4*8-1) * 6.00E-07 + 4.00E-07)] = \\ 0.00117385s \end{aligned} \tag{7.7}$$

The total predicted communication time can then be computed summing up the predicted communication time for all collective functions.

Tables 7.7 and 7.8 show the predicted communication time for BloodFlow when it is executed on Marconi and on the Cloud. Times are expressed in seconds.

TABLE 7.7: Predicted communication Time in seconds for different runs of BloodFlow on Marconi

	8 MPI	**16 MPI**	**32 MPI**	**64 MPI**
Predicted Communication Time	1.89	5.07	8.56	35.14

TABLE 7.8: Predicted communication Time in seconds for different runs of BloodFlow on Google Cloud

	8 MPI	**16 MPI**	**32 MPI**	**64 MPI**
Predicted Communication Time	27.22	71.55	180.32	518.51

7.2.4 Speculation of the execution on cloud

Comparing tab. 7.7 and 7.8 representing the predicted communication time on Marconi and on Cloud, it is easy to notice that on Cloud the values are not only higher than the corresponding predicted values on Marconi, but also they increase much more quickly than on Marconi. This suggests that application scalability might be quite reduced and the breakpoint we observed on Marconi at 512 cores (fig. 6.4) can come before in the cloud, even at 64 cores or fewer. This outcome led us to state that for all applications based on a complex communication pattern such as BloodFlow, Cloud Infrastructure might not be a competitive choice for running such applications, from a performance point of view.

Chapter 8

Validation

To evaluate the correctness of the model and verify our speculations, we moved and ran both applications on the cloud infrastructure. The following two paragraphs show the result we got running Cross Motif Search and BloodFlow on the cloud and give a comparison with the results we got running both applications on Marconi.

8.1 Cross Motif Search

Figure 8.1 compares the performance results obtained running the application on both architectures. Although the cloud infrastructure was set-up to be very close to the HPC configuration, Marconi is slightly superior in raw performance. The reason is mainly due to the overhead introduced by the communication, the virtualization layer and to the interfering jobs being run on the same physical bare-metal. Accordingly our speculation, the outcome we got highlights that Google Cloud Infrastructure can be a good choice for running parallel applications having a small, but not negligible, communication based on master/worker model, such as our CMS application.

8.2 BloodFlow

Figure 8.2 compares the scalability of BloodFlow when it runs on Marconi and on the Cloud Infrastructure. As shown, on the cloud our application seems to be able to scale up 32 cores. On Marconi, instead, the application was able to scale up to 512 cores as shown in Fig. 6.6.

The loss in scalability is mainly due to the increase in the time spent inside the MPI function when the core number grows up, as highlighted by the profiling activities. Figure 8.3, indeed, shows the time spent inside and outside MPI functions varying the MPI process number. Using 32 virtual cores, the time spent inside all MPI functions is almost equal as the time spent outside, and when the core number grows up further, the time spent inside the MPI function is much higher than the time spent in the rest of the application.

According to our speculation, Cloud environment cannot be considered as a feasible place for running scientific applications based on a very complex communication pattern, because the interconnection network plays an important role in the application performance.

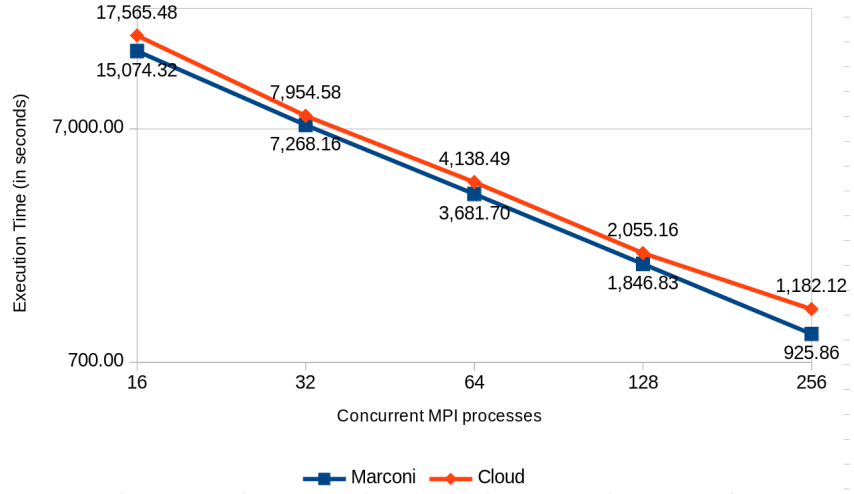

FIGURE 8.1: Comparing Cross Motif Search Scalability on Marconi and on Cloud Infrastructure

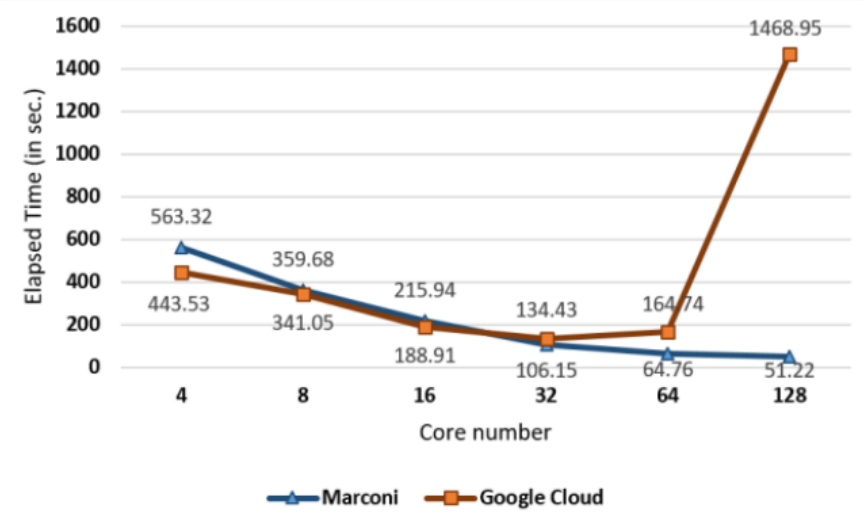

FIGURE 8.2: Comparing BloodFlow Scalability on Marconi and on Cloud Infrastructure

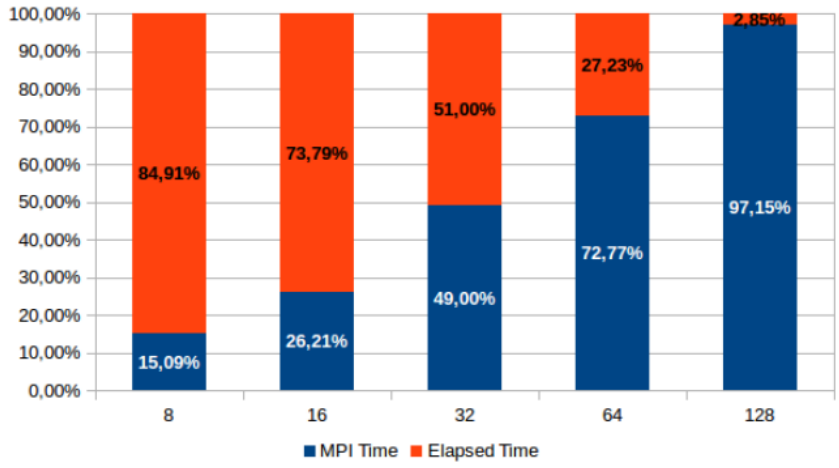

FIGURE 8.3: Time spent inside and outside MPI functions by BloodFlow

Chapter 9

The evaluation model

9.1 Performance and Economical comparison

According to the results shown in figures 8.1 and 8.2, it is clear that Cross Motif Search is able to scale very well even on the cloud but BloodFlow performs worse as it stops scaling at 32 cores, much before than on the HPC system.

This comparison highlights that scientific applications based on a complex communication pattern, such as BloodFlow, might meet several troubles being run on the cloud while those applications based on simple communication model, like Cross Motif Search, are good candidates to be executed on a cloud environment, because of the small impact of the interconnection network. In conclusion, cloud computing does not seem to be yet a convenient place for running scientific application from the performance perspective.

To understand if Cloud Computing can be convenient at least from the economical perspective, we estimated the cost for running a virtual instance on three different cloud platforms provided respectively by Google, Amazon and Microsoft. Each virtual instance runs a Red Hat Enterprise Linux distribution and is equipped with 8 cores, 16 GB of memory RAM and 100 GB of Hard Disk. All virtual instances have also been built on a physical cluster based in London. Table 9.1 shows the cost per hours for all three providers. The table shows that Microsoft is slightly more expensive than the other two providers but Google wins the comparison as the Amazon billing policy is less convenient because, for example, the cost is computed by hours and not by seconds as in Google.

Even though Google is the cheapest solution, it is still more expensive than Marconi, where the cost per hour for 8 cores is two times lower than Google. And even using preemptible instances (that is instances that can be stopped if other tasks require accessing to those resources) the cost, which is dropped by half, stays still too much higher than on Marconi. In conclusion, not even from the economical perspective cloud seems to be convenient.

TABLE 9.1: Cost per hour for running a scientific application on three different cloud infrastructures.

Service Provider	**$ per hour**
Google Cloud	0.41
Amazon AWS	0.41
Microsoft Azure	0.53

9.2 Building the evaluation function

Looking at the results showed in the previous paragraph, cloud computing does not seem to be a convenient place for running scientific application, neither from the performance perspective nor from the economical one. But the landscape might be different if the comparison takes into account other factors making cloud computing so appealing for the users [7–9, 15]. For example, jobs on HPC systems are not usually executed on-the-fly, but they are put in a queue until the required resources are available and afterwards are scheduled. Sometimes, depending on the number of the jobs waiting in the queue and the amount of the physical resources, the waiting time might be quite long. In the cloud infrastructure, instead, there is not any waiting time as the resources are immediately available. In our vision, a fair comparison between cloud and HPC infrastructures should take into account not just performance and economical aspects but also waiting time, job failure, job setup time, maintenance time as well as the user preferences. A time-sensible user might be willing to pay a bit more for getting the results sooner and then the chosen architecture will be different according to its preferences. Choosing the right infrastructure can be essentially seen as a multi-attribute decision-making problem. A proper model based on all these attributes might help researchers to understand which platform might be the best one depending on the user preference, the execution time, the cost for computing and the expected waiting time in the queue. The selected architecture might not be the highest performing one, nor the most affordable, but that one which optimizes the utility function describing the model.

To measure the goodness of each platform using several attributes, we built a utility function based on the weighted geometric aggregation function [128]. The attributes taken into account by the formula are user preference, execution time, cost for computing, expected waiting time in the queue for HPC system and virtual instance startup time for the cloud. The following formulas describe the utility function we adopt for measure the convenience to use the HPC infrastructure U_M rather than the Cloud system U_C for running any application.

$$U_M = \left(\frac{T_M + W_M}{max(T_M + W_M, T_C + S_C)}\right)^{\lambda} * \left(\frac{C_M}{max(C_M, C_C)}\right)^{(1-\lambda)} \tag{9.1}$$

$$U_C = \left(\frac{T_C + S_C}{max(T_M + W_M, T_C + S_C)}\right)^{\lambda} * \left(\frac{C_C}{max(C_M, C_C)}\right)^{(1-\lambda)} \tag{9.2}$$

In formulas 9.1 and 9.2, T_M and T_C are the elapsed time spent by a fixed job running respectively on Marconi and on the Cloud, while C_M and C_C are the cost for running that job respectively on the HPC system and on the cloud. All four parameters are already available as the have been assessed running Cross Motif Search and BloodFlow on both infrastructures. The parameter λ instead is the user preference. Its value ranges between 0 and 1. Having λ=0 means that the user is more sensible to the cost (and then the user would like to have the results with a lower cost, without being interested about the time to completion for getting such results); on the opposite, λ=1 is the preference of a user who is mainly interested to minimize the time to completion and then optimize the turnaround time. The last two parameters, W_M and S_C are respectively the waiting time a job has to wait to be executed on the HPC system (waiting for all required resources are available) and virtual instance startup time, which is the time a cluster of virtual instances takes to get ready to run a job. Both parameters are still unknown and have to be assessed (as described in chapters 10 and 11) to be put into the evaluation model. Using the utility function,

it is then possible to set turnaround time as a criterion for assessment. The model is then used to evaluate all runs of Cross Motif Search and BloodFlow applications on both Marconi and Cloud. If U_C is lower than U_M, users should choose Cloud as the best platform for running the applications, else Marconi is the best one.

As already mentioned, the utility function takes the time spent by the applications to be executed on the cloud and on Marconi and their relative costs. Furthermore, it requires *Waiting Time* and *Virtual Instance Startup Time* which need to be characterized. For this reason, in order to get both parameters, the following chapters 10 and 11 make a characterization of both times.

Chapter 10

Marconi workload characterization

The results described in this chapter refer to the jobs which have been submitted on Marconi during eight months, from the 23rd of January (when SLURM was open to the scientific community) up to the 26th of September 2018 (when A1 partition was closed). To submit a job, users should declare the amount of resources (number of cores and amount of memory RAM) required to run the job, the maximum time limit within the job has to complete its execution and the queue where the job has to be put for execution. When the required resources are available and according to the scheduling policy, the job is started. The job then runs until it completes its execution. Jobs can end in a regular way or with an error. For example, jobs running beyond the time limit are killed by the system with the signal *Timeout*, even the execution is not completed yet.

10.1 Jobs and Partitions on Marconi

10.1.1 Submitted, Queued and Started jobs

During the observed period of time, the amount of jobs submitted on Marconi has been equal to 2,121,429, but 51,449 jobs have been almost immediately cancelled by the user or by the system without being put in the waiting queue. The remaining 2,069,980 jobs, then, have been put on the queue even though for a small amount of time before being executed. Furthermore, 130,837 queued jobs have been cancelled by the users or the system after being put in the waiting queue but before being run, so that even this small subset of jobs is ignored from our analysis. At the end, the number of jobs executed on Marconi during the observed period of time is equal to 1,939,143 (see Table 10.1).

All the results described in this chapter take into account only started jobs, since only all jobs inside such subset yielded significant results from the user perspective.

10.1.2 Jobs per Partitions

All started jobs have been submitted on all three different partitions (as shown in Fig. 10.1) but that one which has been used the most is *A1* (based on the Broadwell architecture), with the 52.65% of the total jobs, followed by *A2*, with 525,239 jobs, and by *A3* with 346,729 jobs. A small amount of jobs has been submitted for the execution on the *sys* partition which is not accessible to the end users but only by technicians for running management applications.

As we are mainly interested to understand for which run of our target applications cloud computing can optimize the turnaround time, we focused our analysis on just those jobs submitted on *A1* partition, being this the partition used for running our applications. Furthermore, such partition exposes a finer-grain parallelism

TABLE 10.1: Submitted, queued and started jobs on Marconi.

Submitted jobs		**2,121,429**
Before queueing	Cancelled by the user	34,562
	Cancelled by the system	13,119
	Failed	3,710
	System Error	58
Queued jobs		**2,069,980**
After queueing	Cancelled by the user	125,261
	Cancelled by the system	414
	Failed	4,242
	System Error	920
Started jobs		**1,939,143**

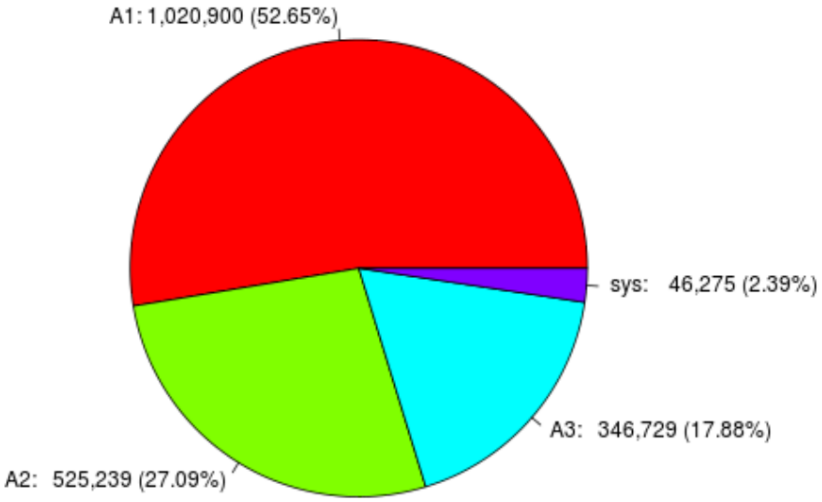

FIGURE 10.1: Amount of jobs started on each Marconi partition

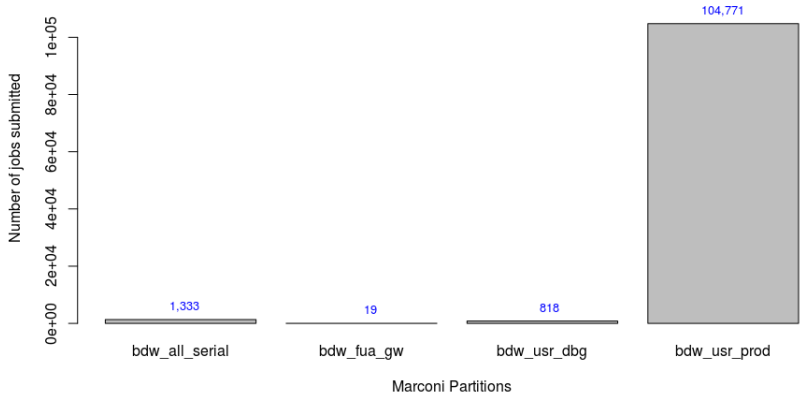

FIGURE 10.2: Amount of jobs started in the queues of Marconi A1 partition

as it allows the users to reserve the exact number of core they need, and this can be smaller than the amount of cores belonging to a node. Then the reserved resources and the allocated resources are always the same, differently from the other partitions where the allocated resources is always a multiple of the core number in a node. Being the cost proportional to the allocated resources, *A1* partition guarantees that the cost corresponds to the amount of resource really required and used. For these reasons, the following analysis is restricted to those 1,020,900 jobs which have been submitted and started on *A1* partition.

10.1.3 Jobs and Queues on Marconi A1

Inside *A1* partition, jobs are submitted on seven different queues, but that one used the most is the *bdw_usr_prod* queue, with 615,996 jobs, corresponding to the 60.34% of the total jobs started in *A1* partition, as shown in Fig. 10.2 .

10.2 Submissions by date and time

10.2.1 Submissions by period of time

The number of jobs submitted every day is quite variable, indeed it ranges between 698 and 23,884, with a mean of 4,150 jobs per days, a median of 3,383 and a standard deviation equal to 2,762. Figure 10.3 shows the number of jobs submitted on each day of the observed period of time. Red horizontal lines represent the first and the third quartile, while the blue line is the median value.

10.2.2 Submissions by hours

The highest number of jobs is submitted between 9 AM and 1 PM (267,946 jobs) (Fig. 10.4, but users submitted their works all day long, as a confirm that Marconi is an HPC system used by many organizations all over the World.

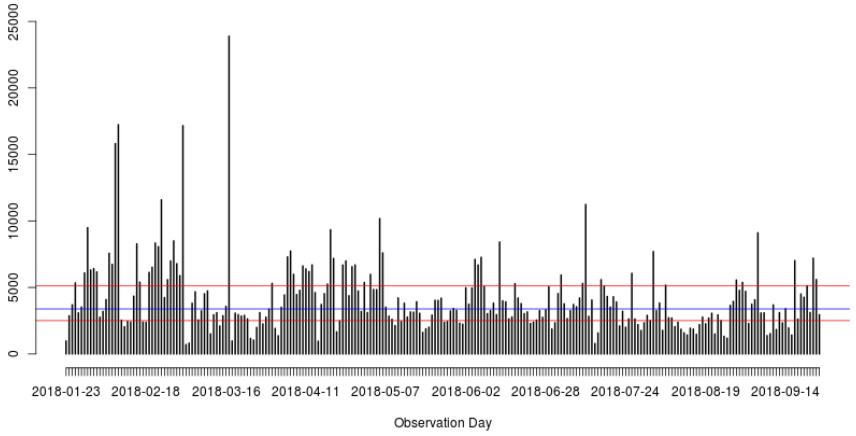

FIGURE 10.3: Amount of jobs per days submitted on Marconi

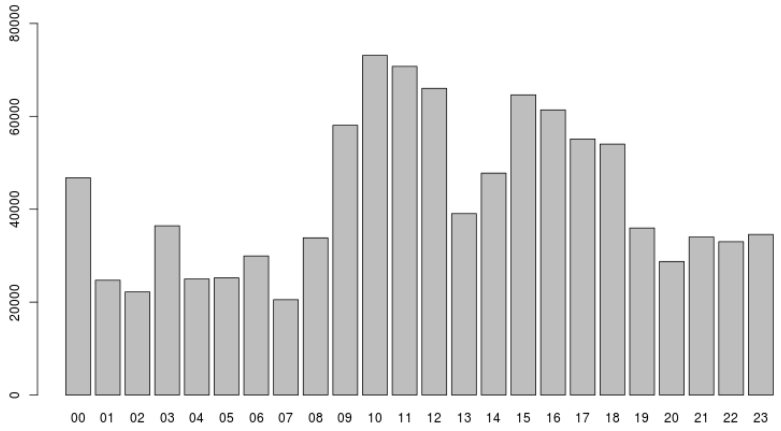

FIGURE 10.4: Amount of jobs per hours submitted on Marconi

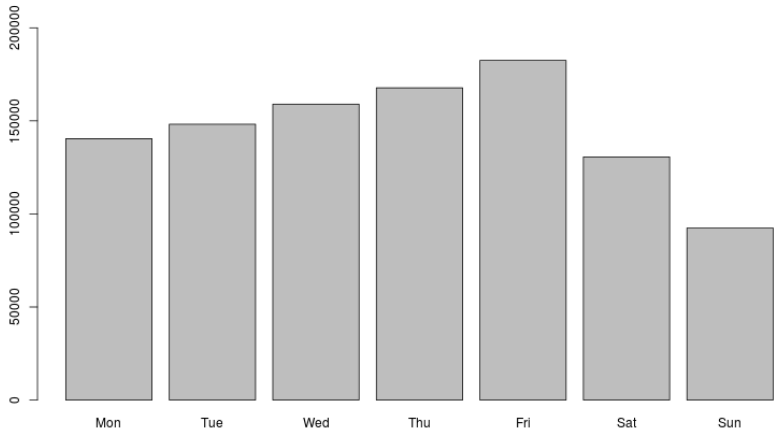

FIGURE 10.5: Amount of jobs per weekday submitted on Marconi

10.2.3 Submissions by week day

From a day perspective, instead, Friday is the day when the highest number of jobs is submitted. As expected, Saturday and Sunday recorded instead the lowest number of submitted jobs, as shown in Fig. 10.5.

10.2.4 Inter-arrival time

Jobs on Marconi are submitted frequently. Figure 10.6, indeed, shows the time in seconds between two consecutive jobs. As shown in the figure, 64.81% of the started job has been submitted after one second or less from its previous, while almost 92% of total jobs has been submitted within a minute. There are some jobs coming after several minutes from the previous. The highest delay has been recorded by the job 13496, that came after 35,078 seconds (more than 9 hours) after its previous. Median and mean values are respectively 0 and 20.81 seconds, with a standard deviation equal to 108.30.

10.2.5 Inter-delivery time

From the user perspective, an interesting aspect to evaluate is how much the HPC system is able to deliver completed jobs. All the jobs submitted on *A1* partition have been taken into account on this analysis, without making any difference between jobs having completed their execution successfully and those having not. The inter-delivery time is highly dependent of many factors, such as the inter-arrival time, the length of each job and the amount of resource available, but the closer the inter-delivery time to the inter-arrival time, the higher the system throughput. Figure 10.7 the cumulative inter-delivery time for all jobs. As shown, only 38.05% of the executed jobs have been delivered after less than 1 seconds from the previous job, but 90.52% have been delivered after less than one minute.

Figure 10.8 shows the inter-delivery time (in seconds) for each executed job. Median and mean time are respectively 4 and 20.88 with a standard deviation equals to 80.85 seconds. All of these values are not too much far from the corresponding

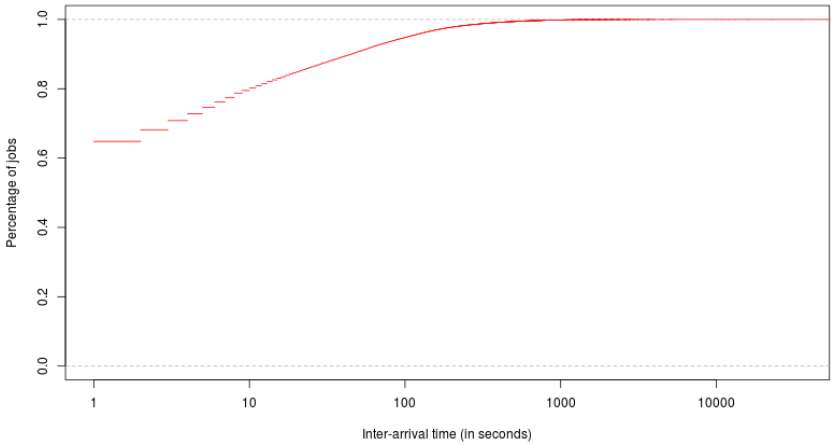

FIGURE 10.6: Inter-arrival time of the jobs submitted on Marconi

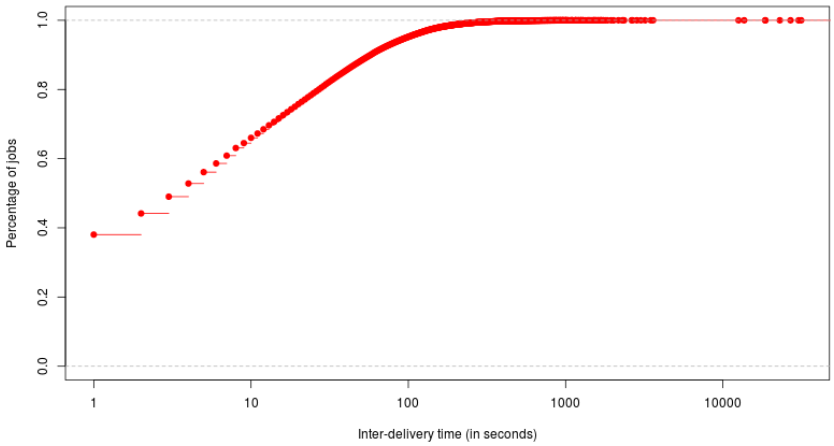

FIGURE 10.7: Inter-delivery time of the jobs submitted on Marconi

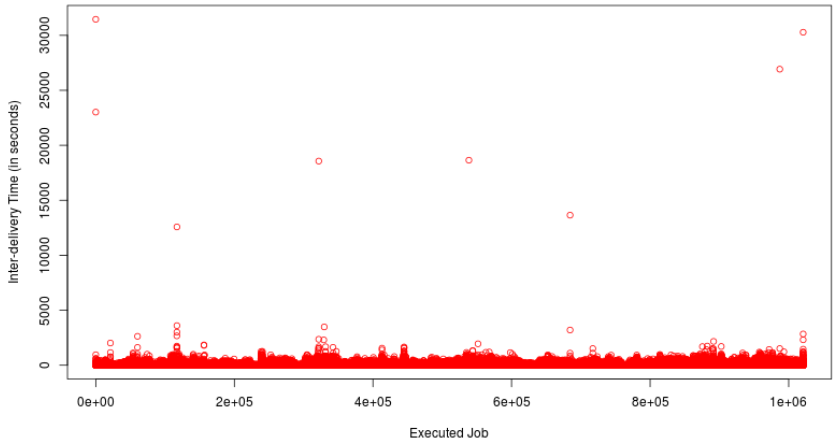

FIGURE 10.8: Inter-delivery time of the jobs submitted on Marconi

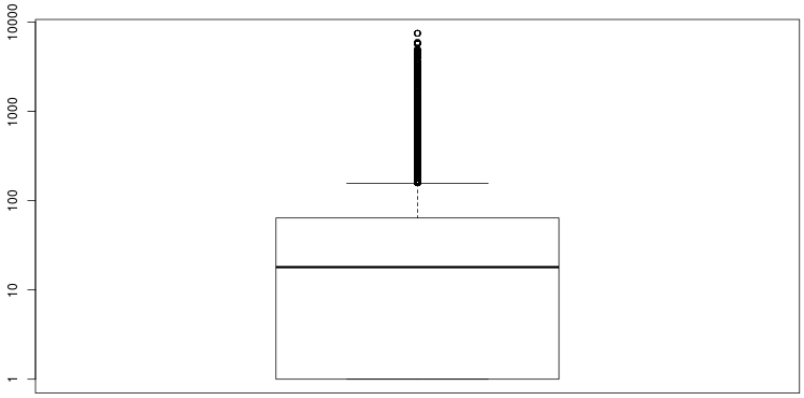

FIGURE 10.9: Number of required concurrent cores

values we got in the inter-arrival analysis. This means that inter-arrival and inter-delivery are quite balanced and in a pure FIFO queue, jobs should not wait too long before being run.

10.3 Job Geometry

10.3.1 Number of required cores

A huge amount of jobs has been submitted to be run in serial. Indeed, 341,833 jobs, corresponding to a third of the started jobs, have used only a single core while 83% of the jobs has been executed with 64 or fewer cores. The remaining jobs, instead, have used up to 7,488 concurrent cores. The median value is 18 while the mean is 58.67, as shown in Fig. 10.9.

Figure 10.10 shows the cumulative distribution function representing the number of jobs requiring less than or equal to a fixed CPU number. The y-axis shows

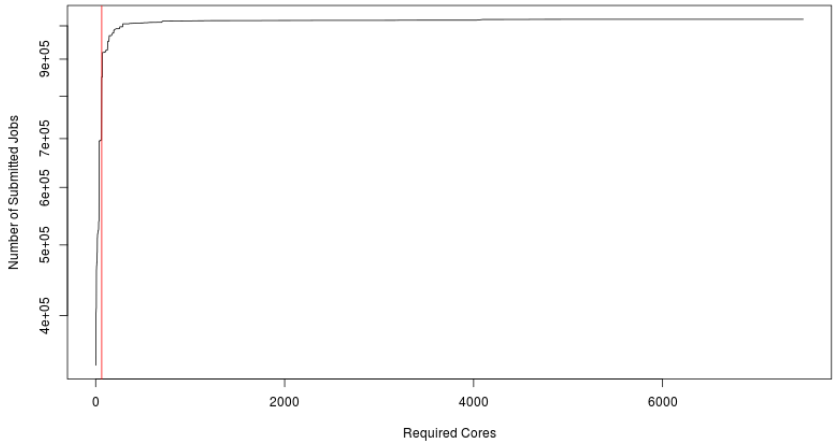

FIGURE 10.10: Jobs requiring less than or equal to a fixed CPU number

the number of submitted jobs, the x-axis instead shows the number of cores. The vertical red line represents 64 cores.

10.3.2 Required core number per queue

A closer look to the job geometry per each queue (Fig. 10.11) shows that the queue named *dbw_usr_prod* is not only the most used but also the queue where jobs required the highest amount of cores (up to 7,488). All other jobs being run on the other queues, instead, always required less than 800 cores. Furthermore, the queue *bdw_meteo_prod* seems to be completely unbalanced as there is a high number of jobs running in serial mode. Indeed, for such queue, minimum value as well as first quartile and median are equal to 1.

10.4 Job Execution Time

As described at the beginning of this chapter, we are characterizing only those jobs which have been successfully submitted on *A1* partition and have been put in a queue for the execution. After a while, according to the Marconi scheduling policy and the resource availability, the job is executed until an event occurs. Such event can be the normal execution completion or some error such as *Timeout*, *Cancelled* or *Failed*.

10.4.1 Job States

Although 82.77% of the started job have regularly completed their execution (844,975 jobs), there is a significant amount of jobs which were cancelled by the user (26,557 jobs) or by the system (28 jobs). More than 80 thousand jobs failed their execution, likely due to an application problem, and more than 40 thousands (4% of the total jobs) instead were killed by the system because the *Time Limit*, defined by the user at submission time, was smaller than the time required to complete the job execution. Jobs killed for timeout represents a big deal for the users as they waste computation time without producing any significant result. A minor percentage of jobs is instead

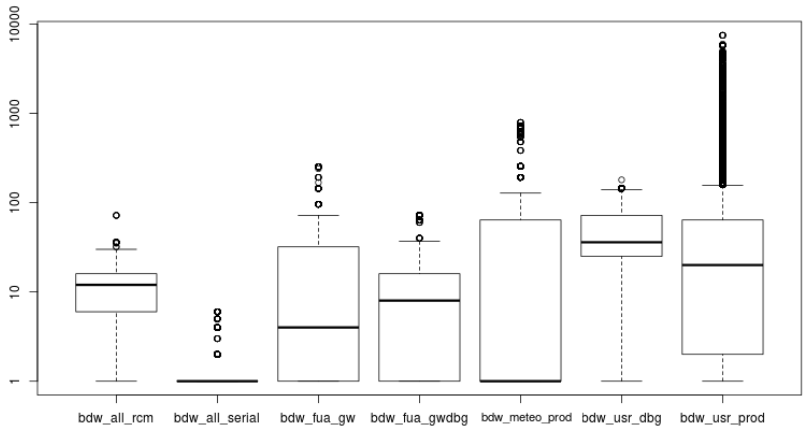

FIGURE 10.11: Number of cores required by processes belonging to the queues

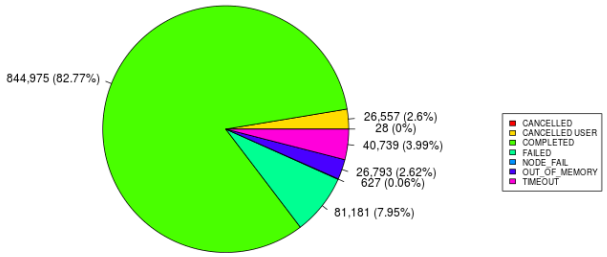

FIGURE 10.12: Completed jobs grouped by state

represented by jobs which failed their execution due to a node fail or for needing a higher amount of memory. Figure 10.12 shows all job termination states, the number as well as the percentage of the jobs that have completed their execution with such state.

10.4.2 Job Elapsed Time in general

As showed in the previous paragraph, just 844,975 jobs have completed their execution correctly. A half of all completed jobs terminated their execution in less than 43 seconds, while 80% stayed running for less than 1,400 seconds (almost 23 minutes), as shown in Fig. 10.13. Only a negligible percentage of jobs (0.06%) took more than 24 hours to complete its execution, with a maximum elapsed time equals to 417,311.

As also showed in Fig. 10.14, the median elapsed time is equal to 43 seconds with a mean of 2,385 seconds.

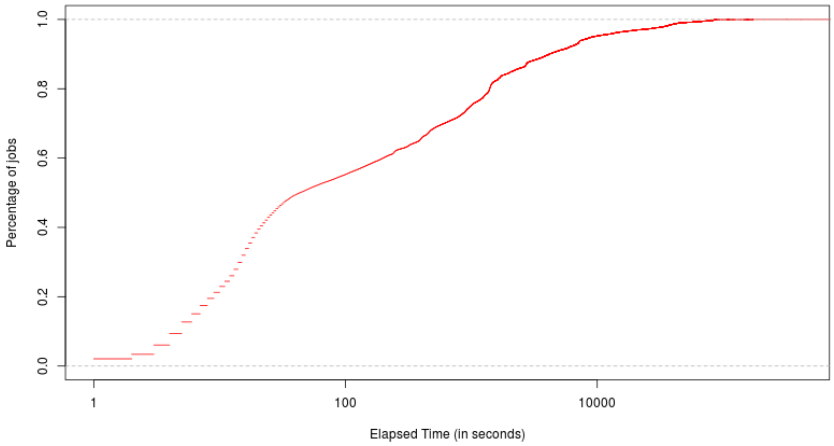

FIGURE 10.13: Cumulative Job Elapsed Time

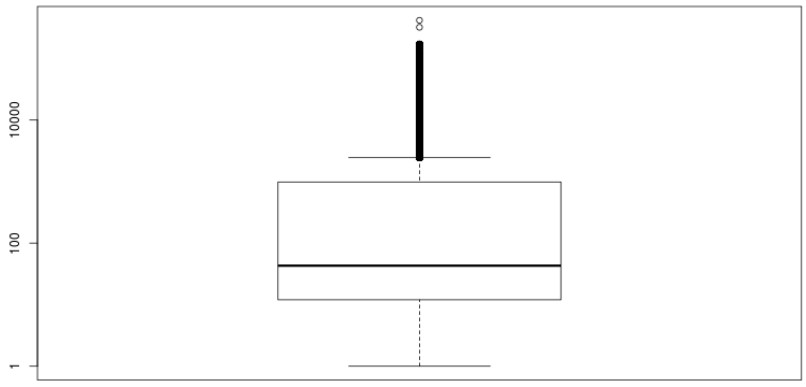

FIGURE 10.14: Job Elapsed Time

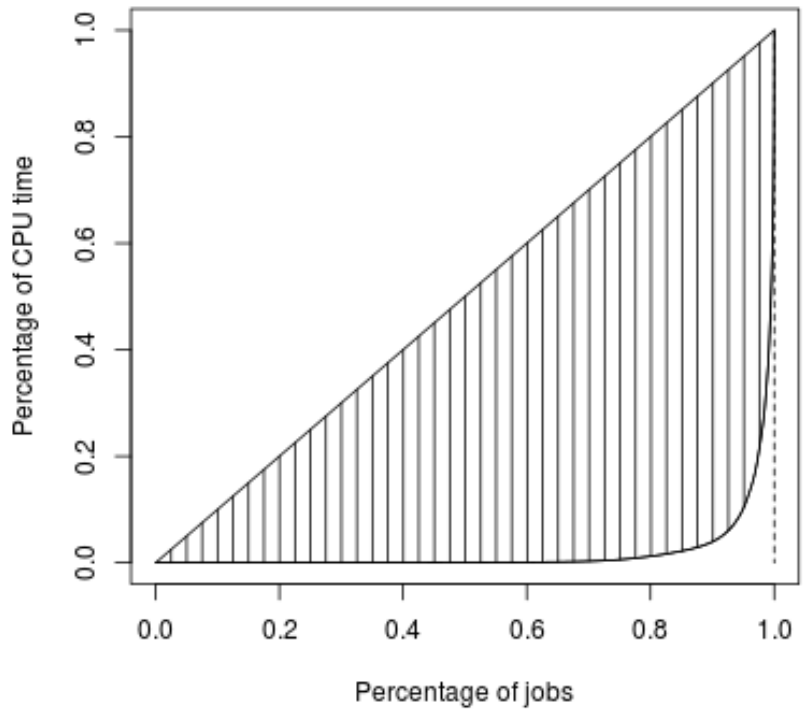

FIGURE 10.15: Lorenz curve

10.4.3 Gini index and Lorenz curve

An interesting perspective to be evaluated is how equally the resources (that is the usage of CPUs) are used by the jobs. Figure 10.15 shows the Lorenz curve which describes how the CPU time is distributed among jobs. The x-axes shows the percentage of jobs executed on Marconi, the y-axes instead shows the percentage of CPU time used by a job. Each point belonging to the rounded curve in the bottom of the picture represents the percentage of CPU Time that has been used by a percentage of jobs. The 45-degree line is the line of equality and corresponds to the perfect equality distribution, while the dashed vertical line corresponds to the perfect inequality distribution. As shown in the figure, the curve is very close to the inequality line, this means that a huge amount of resources (CPU time) are used by a small amount of jobs. Indeed, further analysis reveals that 93.18% of the whole jobs uses only 6.82% of the CPU Time.

Our distribution also showed a very high Gini coefficient, equals to G = 0.96. Gini index is a statistical measure of distribution to study inequality within a distribution. It usually comes with the Lorenz Curve as it provides an analytical description of the inequality level. The coefficient ranges from 0 to 1. The closer the value to zero, the more equally the distribution. The value 1 represents a perfect inequality.

10.4.4 Job Elapsed Time by Queue

As showed in figure 10.16, under the elapsed time perspective almost all queues are similar to each others. Two of them, instead, *bdw_all_serial* and *bdw_fua_gw*, show

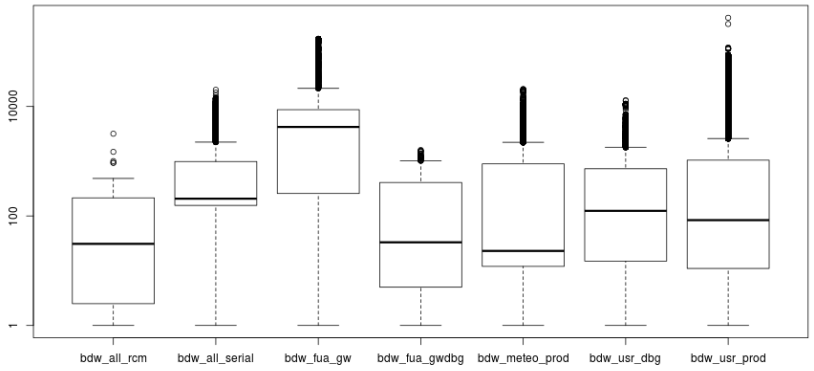

FIGURE 10.16: Elapsed Time per Queue

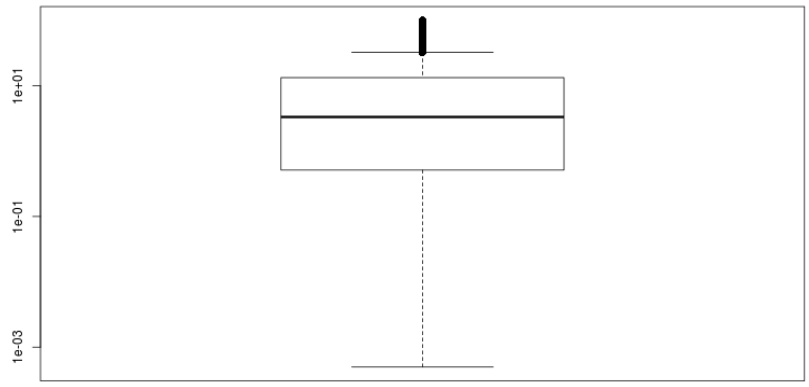

FIGURE 10.17: Accuracy between Time Limit and Elapsed Time

a large collection of jobs with an higher elapsed time, although the longest job has been submitted in the *bdw_usr_prod*.

10.4.5 Accuracy between Time Limit and Elapsed Time

As already described, if the time required by a job to complete its execution is larger than the time limit defined by the user at submission time, the job is killed by the system. To avoid killing jobs for timeout, users usually define a higher time limit, which, according to the scheduling policy, might affect the time spent by the job in the waiting queue. Figure 10.17 shows the accuracy of the time limit respect the elapsed time. The accuracy factor has been computes as follows:

$$Accuracy = ElapsedTime * 100/TimeLimit \tag{10.1}$$

This value ranges between 0 and 100. The closer to 100, the higher the accuracy.

As represented in Fig. 10.17, the median is equal to 3.3, while the mean is equal to 11.74, and this behaviour is similar if compared to the results of each queue (Fig.

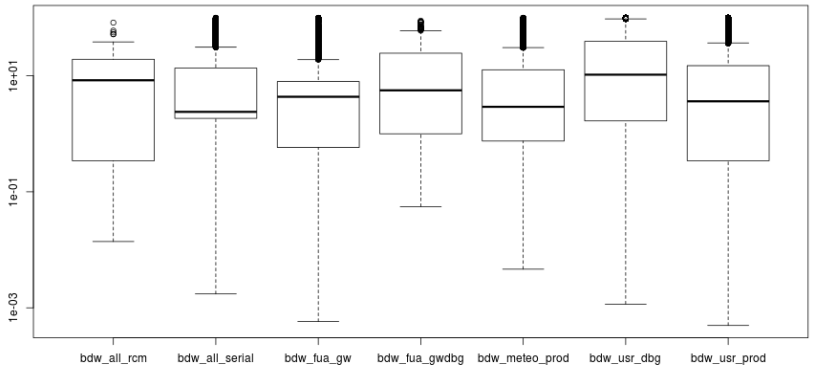

FIGURE 10.18: Accuracy by Queue between Time Limit and Elapsed Time

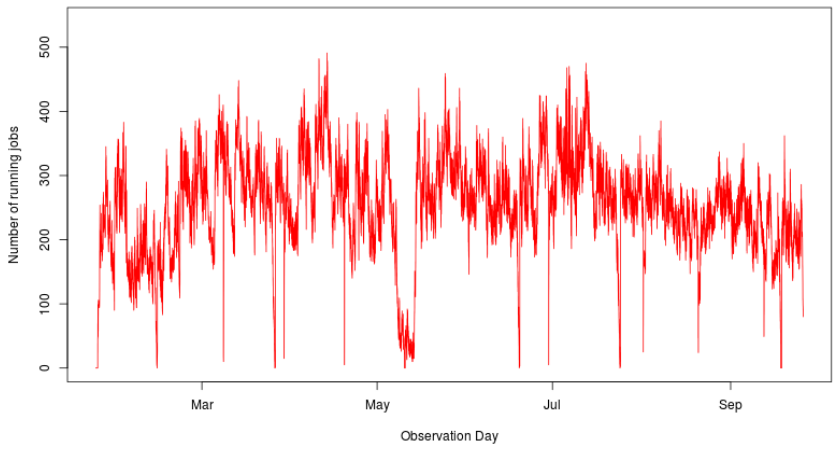

FIGURE 10.19: Running Jobs per day

10.18). For all queues, the median values ranges between 2.39 (for *bdw_all_serial* queue) and 10.44 (for *bdw_usr_dbg* queue). This result shows that Marconi users do not always know in advance the time an application might spend being run on the HPC system and, to avoid the job is killed, they use a time limit which is much higher the elapsed time.

10.4.6 Number of Running jobs

The highest number of jobs concurrently running on the HPC system is 491, but the trend goes up and down as shown in Fig. 10.19. The time when the number of concurrent jobs is zero corresponds to the time when Marconi was put in maintenance.

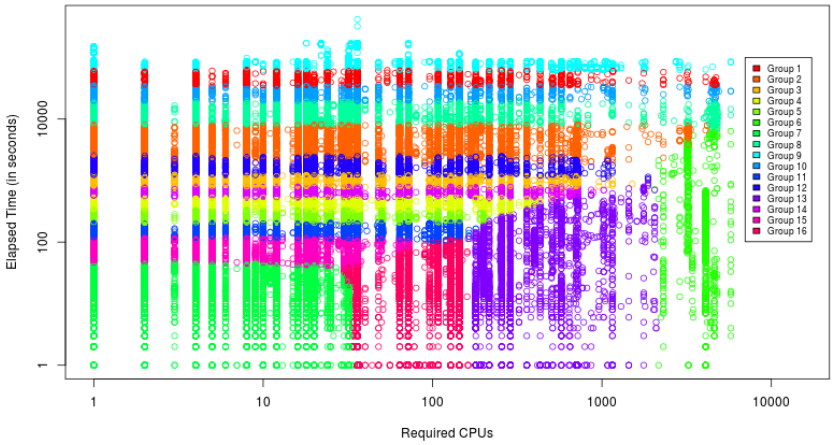

FIGURE 10.20: Job Clusterization

10.5 Job Clusterization

A trivial way to classify jobs is according to the queue where the jobs have been submitted. Anyway, as shown in the previous paragraphs, inside each queue the variance of the jobs, their geometry, the elapsed time and the waiting time is very high. To get sets of more homogeneous jobs, we decided to clusterize jobs according to their geometry and the time spent running. We decided to use k-means as partitioning method for job clusterization. Instead of fixing priori the number of clusters, we iterated k-means method several times, until the covariance coefficient was not lower than 1.1 for all clusters. The covariance coefficient (cc) for the*i-th* cluster is computed as follows:

$$cc_i = \frac{sd(J^i_{ElapsedTime}) + sd(J^i_{CPUNumber})}{mean(J^i_{ElapsedTime}) + mean(J^i_{CPUNumber})} \tag{10.2}$$

where *sd* is the standard deviation function, *mean* instead is the mean function and J is the set of jobs belonging to the *i-th* cluster.

Our test shows that the ideal cluster number is 16. Figure 10.20 shows all clusters found running k-means method on the workload. Table 10.2, instead, shows the number of jobs belonging to each cluster and the corresponding covariance coefficient.

Table 10.2 shows that the largest cluster is also that having the highest covariance factor. Jobs belonging to this cluster are characterized by a higher diversity but, as shown in Fig. 10.20, all of these jobs (dark green circles) exhibit a small amount of required CPUs (less than 34) and also a very small elapsed time (not higher than 46 seconds). Table 10.3 shows, for each cluster, the interval of the CPU required by the jobs belonging to each cluster and the interval of their elapsed time.

TABLE 10.2: Covariance Coefficient Value for each cluster.

ID	Job Number	Covariance Coefficient
1	11,200	0.1678488
2	81,175	0.4429892
3	45,725	0.1932386
4	41,561	0.2196228
5	30,803	0.2362751
6	1,836	0.3503713
7	256,963	0.8988784
8	21,305	0.2922092
9	5,642	0.2662690
10	9,961	0.1785923
11	33,553	0.3376443
12	63,532	0.1992133
13	18,884	0.7526654
14	24,117	0.1981165
15	39,642	0.3964350
16	159,076	0.4780405

TABLE 10.3: CPU number interval and Elapsed Time Interval for each cluster.

ID	CPU Interval	Elapsed Time Interval
1	1 - 4752	34387 - 62258
2	1 - 4176	1864 - 8176
3	1 - 1872	735 - 1234
4	1 - 512	282 - 534
5	1 - 216	200 - 335
6	2048 - 7488	1 - 6643
7	1 - 34	1 - 46
8	1 - 5760	6960 - 19590
9	1 - 5760	62227 - 417311
10	1 - 5760	18892 - 34880
11	1 - 180	103 - 208
12	1 - 1872	1035 - 2593
13	162 - 2088	1 - 920
14	1 - 1024	473 - 803
15	1 - 72	39 - 117
16	30 - 162	1 - 111

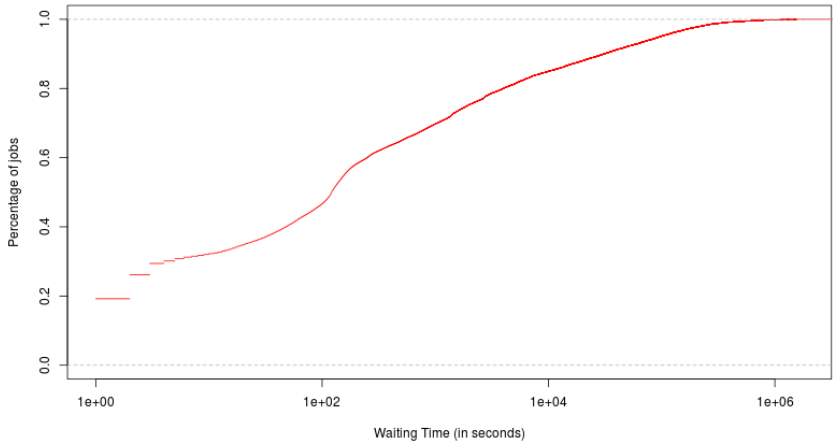

FIGURE 10.21: Cumulative Job Waiting Time

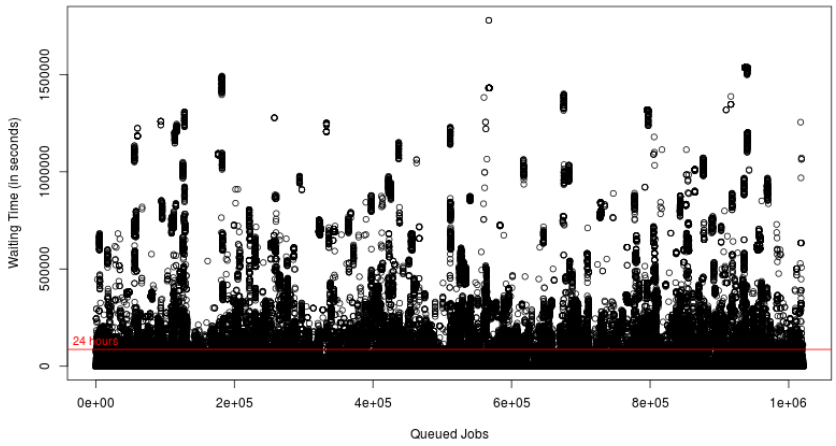

FIGURE 10.22: Job Waiting Time distribution

10.6 Job Waiting time

10.6.1 A global perspective

An overview on the waiting time of all jobs started on A1 partition shows that the time spent by each job waiting to be executed is not negligible as it can last even several days. Indeed, 56,754 jobs, corresponding to 5.56% of the total submitted jobs, had to wait at least 24 hours before being run. Only 19.21% of the jobs, instead, has been executed almost immediately, while almost fifty percent of the jobs waited at least two minutes before being run (Fig. 10.21).

Figure 10.22 shows for each queued jobs, the time spent waiting for being run. Horizontal red line represents a wait time of 24 hours. The median waiting time is equal to 121 seconds, with a mean equals to 18,227 seconds, a maximum value equals to 1,779,294 seconds (more than 20 days) and a standard deviation equals to 82,339.37 seconds.

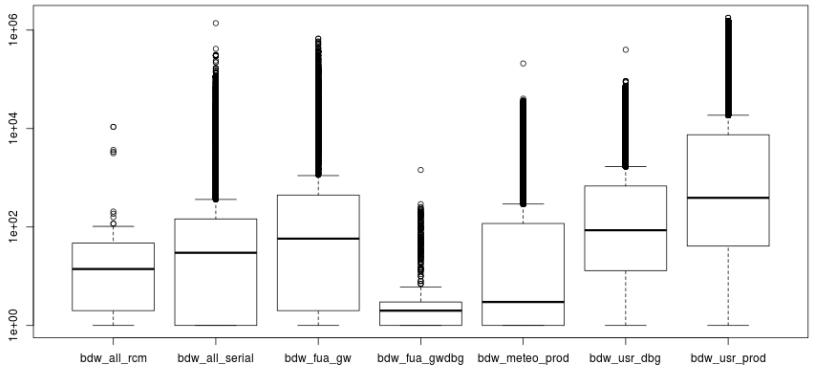

FIGURE 10.23: Job Waiting Time per Queue

TABLE 10.4: Job Waiting Time per Queue.

Queue	min	1st quart.	median	mean	3rd quart.	max
bdw_all_rcm	0	2	14	212	47	10,838
bdw_all_serial	0	1	30	2,414	146	1,382,508
bdw_fua_gw	0	2	58	9,206	446	674,984
bdw_fua_gwdbg	0	1	2	16	3	1,439
bdw_meteo_prod	0	1	3	365	118	209,874
bdw_usr_dbg	0	13	86	2,191	681	398,366
bdw_usr_prod	0	42	392	29,48	7,486	1,779,294

10.6.2 Waiting time by queues

A deeper analysis on the waiting time shows that many jobs are affected by the waiting time regardless the queue where each job is submitted. The queue where the waiting time affects the most the job execution is the *bdw_usr_prod*, which is also the most used. Inside such queue, the median waiting time is 392 seconds, while the third quartile and the mean value are respectively equal to 7,486 and 29,481 seconds.

But even for the other queues the maximum waiting time is however not negligible being always over than 1,439 seconds as shown in Fig. 10.23 and in Table 10.4.

10.6.3 Waiting time by clusters

We also did a similar analysis using clusters we found using k-mean technique. As described in Fig. 10.24 and in Table 10.5, the median value for all clusters ranges between 3 seconds (cluster 7) and 92,094 seconds (cluster 1).

10.6.4 Correlation between Job Geometry and Job Waiting Time

Figure 10.25 shows the correlation between the number of required cores for each job and the job waiting time. Although for a fixed geometry the waiting time spreads across several values, it is possible to notice a small correlation (ρ = 0.20) between core number and waiting time. Indeed, the maximum waiting time goes down

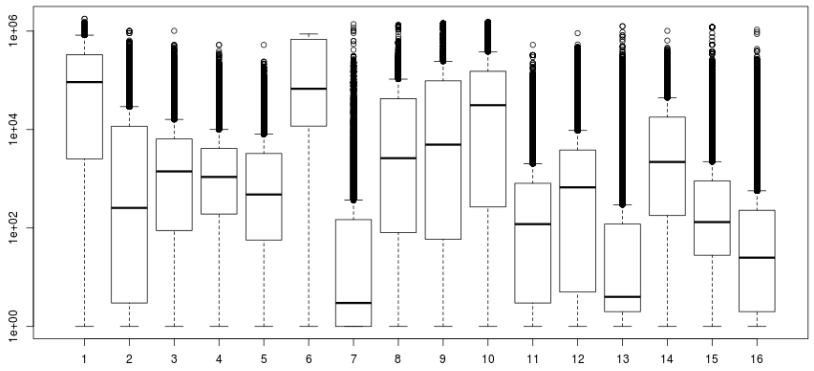

FIGURE 10.24: Job Waiting Time per Cluster

TABLE 10.5: Job Waiting Time per Cluster.

Queue	min	1st quart.	median	mean	3rd quart.	max
Cluster 1	0	2,537	92,094	219,940	332,596	1,779,294
Cluster 2	0	3	256	21,624	11,680	1,008,409
Cluster 3	0	89	1,413	15,821	6,475	1,011,275
Cluster 4	0	192	1,089	12,145	4,127	523,31
Cluster 5	0	57	479	7,045	3,258	523,31
Cluster 6	0	11,795	67,414	276,079	677,283	873,641
Cluster 7	0	1	3	1,537	148	1,382,508
Cluster 8	0	81	2,620	40,195	42,378	1,318,36
Cluster 9	0	59	4,940	97,509	97,442	1,431,388
Cluster 10	0	268	31,233	149,690	152,391	1,522,522
Cluster 11	0	3	120	4,250	810	523,317
Cluster 12	0	5	670	22,854	3,840	907,381
Cluster 13	0	2	4	3,389	120	1,256,129
Cluster 14	0	179	2,196	22,78	17,882	1,011,712
Cluster 15	0	28	132	7,348	910	1,200,751
Cluster 16	0	2	25	1,183	229	1,065,234

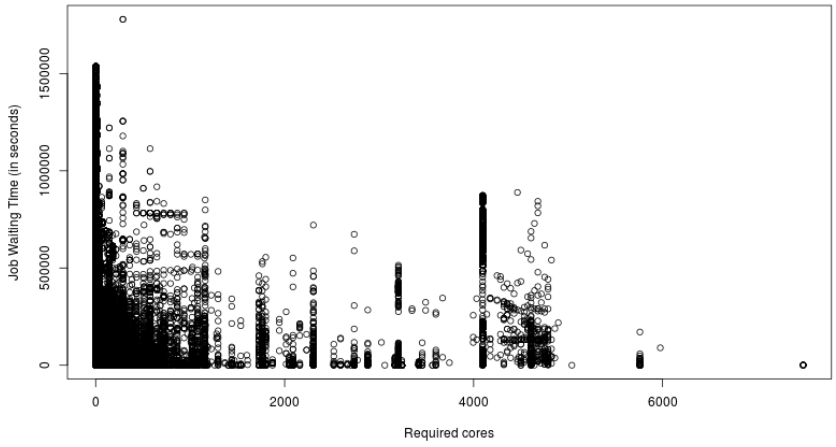

FIGURE 10.25: Correlation between Required Cores and Job Waiting Time

when the number of required cores rises up. The scheduling policy of Marconi then favours larger jobs, using the smaller ones to fill the unused cores.

10.6.5 Correlation between Time Limit and Waiting Time

If Marconi, from one side, favours those larger jobs, on the other side, instead, jobs submitted with larger Time Limit might wait a longer time. This behaviour is shown in Fig. 10.26, which shows the correlation between the Time Limit and the Waiting time. In the figure, it is possible to notice that the maximum waiting time for a fixed class of jobs having the same Time Limit rises up when the time limit grows.

The correlation between Time Limit and Job Waiting Time is slightly higher than the correlation between Job Geometry and Job Waiting Time ($\rho = 0.27$).

10.6.6 Ratio between Elapsed Time and Waiting Time

In previous paragraph we characterized the waiting time without taking into account the elapsed time. Instead of considering the absolute waiting time, it is worth studying the time spent by a job waiting in the queue respect the time spent by the same job to complete its execution. Indeed, a waiting time of five minutes might appear to be a small amount of time if compared with a job taking several hours to be completed, but the same waiting time might be considered too high if the job elapsed time is just a few minutes. We define the relative waiting time as follows:

$$RelativeWaitingTime = WaitingTime/ElapsedTime \tag{10.3}$$

This value is always greater than or equal to 0. Better values are closed to 0. The higher the relative waiting time, the greater the impact of the waiting time respect to the elapsed time.

As shown in Fig. 10.27, more than 50% of all executed jobs has been waiting in the queue for almost the same time spent in computation, while the 80% of all executed jobs has been waiting in the queue for almost 15 times the time spent by the job in computation.

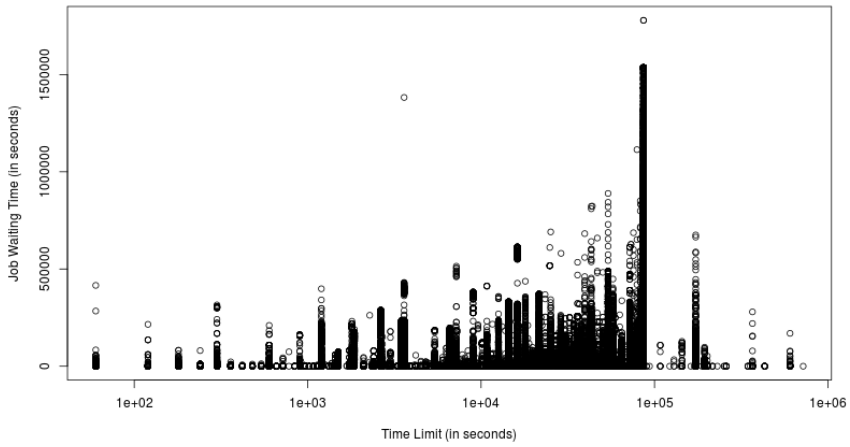

FIGURE 10.26: Correlation between Time Limit and Job Waiting Time

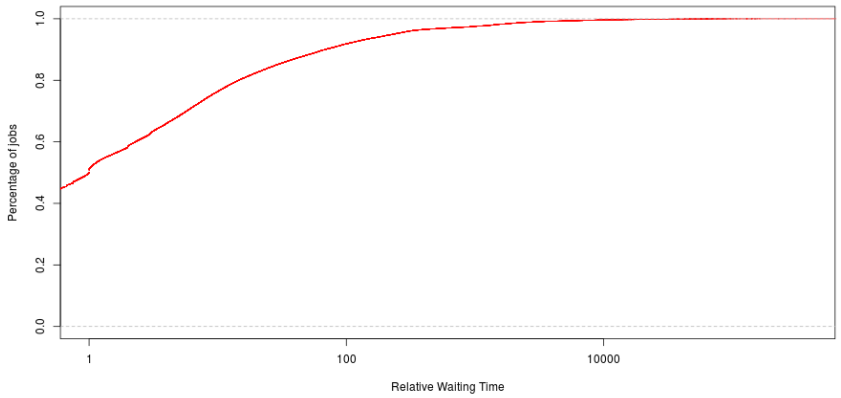

FIGURE 10.27: Cumulative Relative Waiting Time

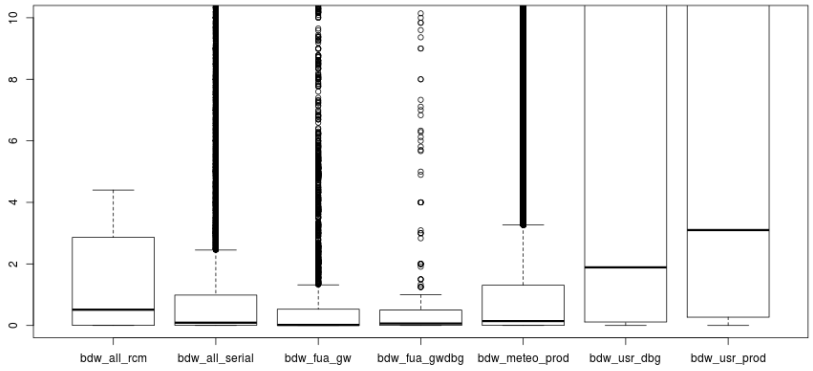

FIGURE 10.28: Cumulative Relative Waiting Time per Queue

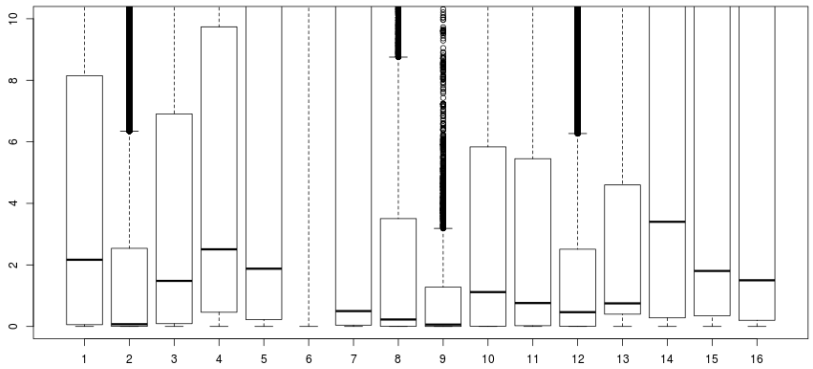

FIGURE 10.29: Cumulative Relative Waiting Time per Cluster

A synthetic analysis on the relative waiting time shows a median value equals to 1, first and third quartile equal respectively to 0.1 and 8.8, a mean equals to 203.5 and a standard deviation of 2,651.25.

10.6.7 Relative Waiting Time by Queues and by Clusters

So far we have grouped all jobs in two different ways: according to the queue where the jobs have been submitted and according to the clusters, discovered using k-means method, where the jobs belong to. As the main aim of this work is to characterize the waiting time the jobs might experience on an HPC system like Marconi, we studied how the relative waiting time changes depending on both classifications. Figures 10.28 and 10.29 show how the relative waiting time is distributed respectively on all queues and on all clusters. Tables 10.6 and 10.7, instead, show the numerical values of such distribution, in particular the minimum and the maximum value, the first and the third quartile, and finally, the median and the mean value.

On both representation, the mean value is almost always greater than the third quartile. This highlights that the distribution is badly affected by some outlier making the mean value and the maximum value much higher than the median value.

TABLE 10.6: Relative Job Waiting Time per Queue.

Queue	min	1st quart.	median	mean	3rd quart.	max
bdw_all_rcm	0.00	0.00	0.51	100.97	2.62	3,671.00
bdw_all_serial	0.00	0.01	0.09	82.76	0.99	230,418.00
bdw_fua_gw	0.00	0.00	0.02	1,032.68	0.53	95,233.00
bdw_fua_gwdbg	0.00	0.00	0.06	1.128	0.50	52.00
bdw_meteo_prod	0.00	0.00	0.14	13.57	1.31	3,4899.00
bdw_usr_dbg	0.00	0.11	1.89	122.91	14.16	46,260.50
bdw_usr_prod	0.00	0.30	3.10	314.60	23.60	377,678.05

TABLE 10.7: Relative Job Waiting Time per Cluster.

Queue	min	1st quart.	median	mean	3rd quart.	max
Cluster 1	0.00	0.06	2.17	5.50	8.15	42.96
Cluster 2	0.00	0.00	0.07	6.21	2.54	248.46
Cluster 3	0.00	0.09	1.48	15.84	6.91	1,015.34
Cluster 4	0.00	0.46	2.51	29.57	9.74	1,461.78
Cluster 5	0.00	0.23	1.88	25.82	12.24	2,528.10
Cluster 6	0.00	81.32	708.14	2,341.14	3,387.19	69,066.80
Cluster 7	0.00	0.04	0.50	409.09	11.00	313,432.00
Cluster 8	0.00	0.01	0.23	3.38	3.51	133.06
Cluster 9	0.00	0.00	0.06	1.33	1.28	20.95
Cluster 10	0.00	0.01	1.12	5.27	5.84	64.79
Cluster 11	0.00	0.02	0.76	28.31	5.45	4,120.61
Cluster 12	0.00	0.00	0.47	14.39	2.51	507.77
Cluster 13	0.00	0.40	0.80	381.70	4.60	37,7678.50
Cluster 14	0.00	0.28	3.40	37.16	27.68	1,809.86
Cluster 15	0.00	0.35	1.80	99.38	12.53	20,702.60
Cluster 16	0.00	0.20	1.50	284.07	13.87	166,114.00

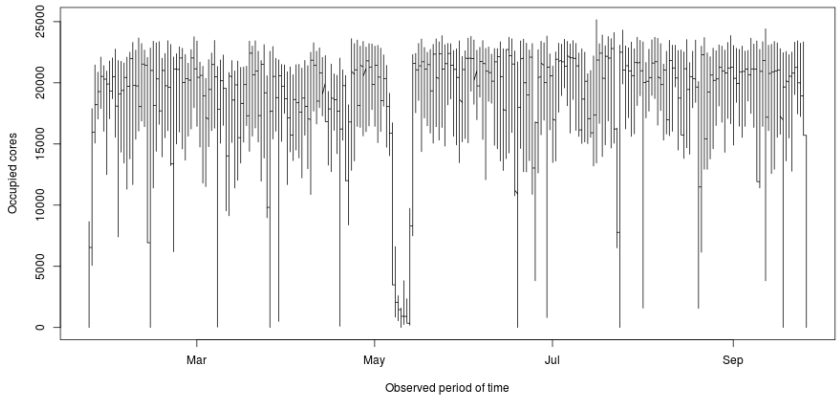

FIGURE 10.30: Number of cores concurrent occupied by running jobs

For this reason, median value can better describe the waiting time because it is insensitive to the presence of the outliers.

By looking at the Tables 10.6 and 10.7, it is possible to notice that the median of the relative waiting times for queue-oriented distribution ranges between 0.02 and 3.10, while on the cluster-oriented distribution the presence of the cluster 6 makes the spread higher than the other distribution (from 0.06 to 708.14). Making a drill down cluster 6 (having the highest median), we observed that all jobs belonging to *bdw_usr_production* queue are characterized by a large number of required cores (between 2048 and 7488) and an elapsed time which is always lower than 6643 seconds but with a very high waiting time (up to 873,641 seconds). In Fig. 10.20, these jobs cover the dark green area on the bottom-right side of the picture.

10.7 Total system Utilization

During the observed period of time, Marconi has been working almost all the time, except for some small intervals where the system was put closed for maintenance. Figure 10.30 shows the number of concurrent cores busy during the whole observed time. The maximum value is 25,153, corresponding to 97% of the total number of core belonging to the HPC infrastructure, which is 25,920, as described in chapter 2.

Anyway, to compute the real total system utilization, maintenance periods of time, which made Marconi closed, need to be taken in account. The analysed workload contains all jobs submitted from January 2018, the 23rd, at 9.00 AM to November 2018, the 26th, at 7.00 PM, for a total of 5,914 hours. According to the system log, the HPC infrastructure was closed for maintenance for 53 hours. Table 10.8 shows the time when the infrastructure has been completely closed. These times are comparable with the job execution trend shown in Fig. 10.19 and 10.30, where the lines fall down to zero when the system was closed and the jobs stopped. However, this amount takes into account just the time when the infrastructure has been completely closed to the users and no node was accessible. Partial closing are instead not computed because they affected only a small number of nodes, even just one, and users would have anyway submitted jobs for being executed in the working nodes. As the number of cores belonging to A1 was 25,920, the total core-hours can be computed as

TABLE 10.8: Marconi's interruptions.

Day	From	To	Lenght
02/13/2018	08.30.00	13.00.00	04.30.00
03/08/2018	09.30.00	13.00.00	03.30.00
03/26/2018	08.30.00	15.00.00	06.30.00
04/19/2018	08.00.00	16.00.00	08.00.00
05/10/2018	12.00.00	18.00.00	06.00.00
06/19/2018	10.00.00	15.00.00	05.00.00
06/29/2018	14.00.00	17.30.00	03.30.00
07/24/2018	08.30.00	16.30.00	08.00.00
11/18/2018	10.00.00	18.00.00	08.00.00

$$TotalCoreHours = (5,914 - 53) * 25,920 = 151,917,120 \tag{10.4}$$

.

To compute the Total System Utilization we used the following formula :

$$TSU = CoreHourUsed * 100/CoreHoursAvailable \tag{10.5}$$

According to the workload, core-hours used in the observed period of time was equal to 112,247,824, then the TSU is equal to

$$TSU = 112,247,824 * 100/151,917,120 = 73.89\% \tag{10.6}$$

As the core-hours available in the period does not take in account the single core or the subset of cores closed for maintenance, as said above, such Total System Utilization can be considered as a lower bound.

Chapter 11

Virtual Instance Startup and Stop Time

The evaluation model described in chapter 9 takes not only the waiting time, which has been characterized in the previous chapter, but also the virtual instance startup time. Many works [103–105] have already studied Virtual Instance Startup Time and its relations with other factors such as the time of the day, operating system image size, instance type, data centre location and number of instances requested at the same time. Those researches have stated that the average virtual machine startup time, which is the time a user should wait before having the first successfully ssh login, on a single core Linux system running on the Amazon EC2 infrastructure is about 90 seconds, while a similar system equipped with 16 virtual cores takes 146 seconds. The following analysis instead covers the startup time and the stop time measured running single virtual instance built on a physical infrastructure provided by Google, hosted in the West US (*us-west2-a*) equipped with Centos 7 as operating system, 50 GB of virtual disk and using different virtual hardware configurations. Both times have been measured starting and stopping virtual instance from a custom tool written using the Google SDK. As the starting and stopping commands block until the operation is completed, measuring the waiting time spent by the commands gives an assessment of the time spent for starting and stopping the virtual instances.

Figures 11.1 and 11.2 show respectively the startup time and the stop time measured varying the number of virtual cores from 1 to 16 and the amount of RAM memory from 3.75 GB to 60 GB. For each configuration, the virtual instance has been started and stopped ten times. The first picture shows the startup time for all five configurations. The startup time seems not to be heavily affected by the virtual instance configuration. Indeed, the median startup time observed using 16 cores is not too much higher than that one measured using a smaller configuration. For all configurations, the median startup time is about 10 seconds. Even the stopping time seems to be not badly affected by the configuration as the median value for each configuration is almost the same and it does not go beyond 21 seconds.

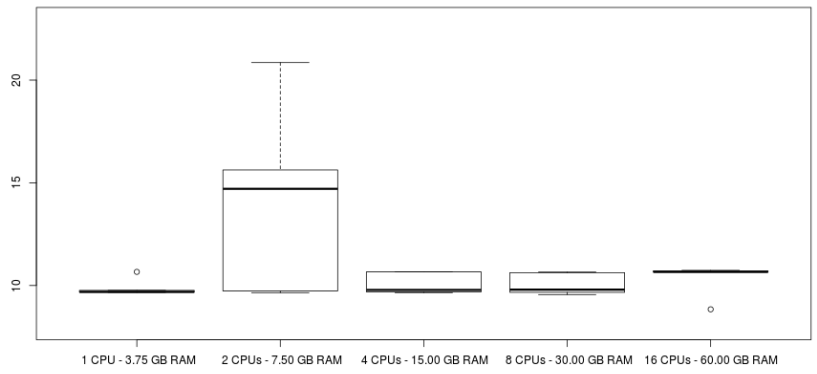

FIGURE 11.1: Virtual Instance Startup Time

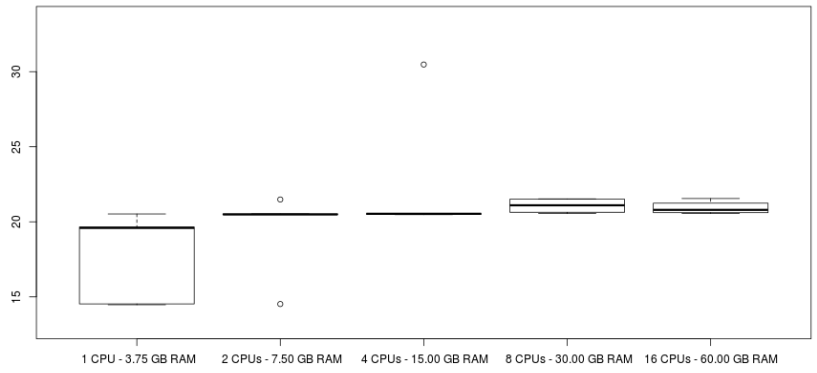

FIGURE 11.2: Virtual Instance Stop Time

Chapter 12

Applying the evaluation model on both applications

The utility function described in chapter 9 takes the time spent by the applications to be executed on the cloud and on Marconi and their relative costs. Furthermore, it requires the Virtual Instance Startup Time and the Waiting Time. Previous two chapters gave us a measure of the time spent by a virtual instance to get ready to start the application. The startup time seems not to be dependent to instance configuration. Then, according to the previous analysis, we can fix to 10 the startup time S_c in the utility function.

For the parameter W_M, which is the job waiting time measured on Marconi, instead of using a static value for all runs, we decided to identify the cluster where each run of Cross Motif Search and BloodFlow might belong to, according to the job geometry. Then, median relative waiting time RW_M for the selected cluster is chosen as factor to determine the waiting time to put into the utility function for a fixed run. The job waiting time W_M then can be easily computed as follows:

$$W_M = RW_M * T_M \tag{12.1}$$

where T_M is the job elapsed time. To be clearer, let's consider the first run of Cross Motif Search. The application took 15,074.32 seconds using 16 cores. According to the job clusterization defined in Table 10.3, the run might belong to the cluster number 8, where the median relative waiting time for all job in that cluster is 0.23 (see Table 10.7). Then for this run, the waiting time W_M is equal to 15,074.32 * 0.23 = 3,467.09 seconds. The last run of BloodFlow, instead, took 51.22 seconds using 128 cores. Then this run belongs to the cluster number 16 (see Table 10.3), having a relative waiting time equals to 1.50 (see Table 10.7). Then for such run, the waiting time is equal to 76.83 seconds (51.22 * 1.50 = 76.83).

Table 12.1 shows the cluster where each run of both applications belongs to.

Now, we have got all information needed to fill the utility function and then understand for which runs of both target applications cloud was more convenient according to our evaluation model which takes into account not only the performance

TABLE 12.1: Cluster where each execution of Cross Motif Search and BloodFlow belongs to.

CPU Number	**4**	**8**	**16**	**32**	**64**	**128**	**256**
Cross Motif Search			8	8	2	12	3
BloodFlow	14	4	5	11	15	16	

		CROSS MOTIF SEARCH				
		16	32	64	128	256
User preference	0,0	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,1	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,2	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,3	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,5	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,6	MARCONI	MARCONI	MARCONI	MARCONI	CLOUD
	0,7	MARCONI	MARCONI	MARCONI	MARCONI	CLOUD
	0,8	MARCONI	MARCONI	MARCONI	CLOUD	CLOUD
	0,9	MARCONI	CLOUD	MARCONI	CLOUD	CLOUD
	1,0	CLOUD	CLOUD	MARCONI	CLOUD	CLOUD

FIGURE 12.1: Preferred architecture for running Cross Motif Search

		BLOODFLOW					
		4	8	16	32	64	128
User preference	0,0	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,1	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,2	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0,3	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI	MARCONI
	0,5	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0,6	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0,7	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0,8	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI
	0,9	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI
	1,0	CLOUD	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI

FIGURE 12.2: Preferred architecture for running BloodFlow

and the cost but also the turnaround time and the user preference.

Figures 12.1 and 12.2 show which architecture might be the preferred for each run depending on the user preference. For 22% of all Cross Motif Search runs, Cloud Infrastructure might be the best architecture for running the application. The percentage rises up to 48% for the BloodFlow application. These results are heavily affected by the chosen relative waiting time. Indeed, for the Cross Motif Search application, the relative waiting time for each cluster where the runs belong to is always lower than all the relative waiting time used for the BloodFlow application. Just increasing the relative waiting time of the cluster number 2 from 0.07 to 0.40, the cloud preference for Cross Motif Search rises up from 22% to 28%.

It is worth noting that the increase we introduced changing the relative waiting time from 0.07 to 0.40, is not negligible. In fact, supposing to have an elapsed time of 3,681.70 sec (which is the real elapsed time Cross Motif Search took being run on Marconi using 64 cores), changing the relative waiting time from 0.07 to 0.40, the waiting time goes from 258 seconds to 1,473. According to this observation, we can state that our model is robust, since a high perturbation of the relative waiting time brings a small variation in the convenience to use the cloud infrastructure rather than the HPC system. The results presented above also show that although cloud computing might be more expensive and less powerful than the HPC system, when turnaround time becomes important, cloud computing can be a convenient alternative for running scientific applications.

Chapter 13

Conclusion

Cloud infrastructure has emerged as an interesting and convenient alternative of on-premise system for running a broad range of software applications, such as web sites, customer relationship management systems, enterprise resource planning systems, business performance management systems and so on. Because of the wide number of advantages brought along by such technology, many enterprises are thinking to move their own software and infrastructure toward the cloud, reducing costs and increasing the system availability. But when performance are important, the landscape might be quite different. Indeed, applications that are designed to be executed on an HPC system, might behave worse when executed on the cloud infrastructure. This has been proven in this work, where we moved two applications to the cloud , named Cross Motif Search and BloodFlow. Both applications can be considered as opposite to each other as they are based two different communication models, so they can represent a huge range of scientific applications. Even though Cross Motif Search showed a good scalability on the cloud (similar to that observed running the application on the HPC system), both applications performed worse when executed on the cloud, most over for BloodFlow where, using a large number of concurrent MPI processes, the time spent to compute the entire dataset grew fastly becoming soon unmanageable, making cloud infrastructure not useable for that application. Profiling activities we did on both applications revealed that the main factor affecting the application performance is the network infrastructure and even optimizing the code, cloud environment lie behind the HPC infrastructure.

It seems to be clear that, as moving an application to the cloud is not effortless, before starting such activities it might be worth knowing in advance how an application might behave being executed on the cloud, and in general on a different infrastructure. As one of the most important factor which might affect the application performance is the interconnection network, we built an analytical model describing the communication model of both applications. Gathering application-related and network-related parameters of the HPC system and the cloud infrastructure, we have been able to build the analytical model for our target applications. Even though such model cannot be used to predict the time spent by the application inside MPI functions, it might be useful to get hint and insight about how the application might perform being run on a infrastructure.

In conclusion, cloud computing does not seem to be a convenient place for running scientific application at least from the performance perspective. In order to understand if at least under the economical point of view cloud might be interesting, we made a similar analysis comparing the cost for running a generic scientific application in a cloud environment provided by three different service provider (Google, Amazon and Microsoft). Results showed that the cost a researcher must afford for running its application on the cloud is higher than the cost afforded to run the same application for the same amount of time on an HPC system. So not even from the

economical perspective, cloud computing seems to be convenient at all. Anyway, taking into account other factors making the cloud more appealing, the landscape in the user convenience changes. Choosing the right infrastructure, then, can be essentially seen as a multi-attribute decision-making problem. We introduced an evaluation model, based on a weighted geometric aggregation function, that takes into account a set of parameters, among which job geometry, cost, execution and turnaround time. The notion of user preference modulates the model, and allows to determine which platform, cloud or HPC, might be the *best* one.

As the evaluation model takes into account also the job waiting time, in order to assess such parameters, we made a characterization of the workload of Marconi, a real HPC system. But instead of getting only one waiting time for all jobs submitted on Marconi, we clusterized the workload according the job geometry using k-means algorithm. Then, for each cluster, the corresponding job waiting time has been assessed and used into our evaluation function in order to evaluate the utility for running our target jobs on the HPC system and on the cloud infrastructure. The model has then been used to evaluate the best architecture for several runs of two applications, based on two different communication models. Applying our evaluation model to all executions of our target application revealed that there is a not negligible number of configuration where cloud computing seems to be the best architecture for running Cross Motif Search and BloodFlow. Results also showed that the model as high perturbation in the waiting time brings small variation in the results.

13.1 Future works

The evaluation model described in this work has to be intended as preliminary. Our aim indeed is the extension of the model in order to include not just cloud and HPC systems but also on-premise architectures and GPU-based infrastructures. Furthermore, the model should be extended to keep into account also other parameters, such as the energy cost and other factors making cloud computing appealing. We further aim to apply the evaluation model to other scientific applications, based on a different communication model.

Appendix A

Introduction to MPI

Applications running on distributed memory architectures rely on message passing communication for sending and receiving messages across all processes. Although several message-passing libraries have been developed so far, many applications make use of MPI (Message-Passing Interface) [106], which is a message-passing library interface specification developed by a consortium of academic, research, and industry partners. Being an interface specification, it is not a real implementation but a set of rules and specifications each implementation need to adhere. Thankfully to its flexibility and efficiency, the interface is now considered as the standard for building parallel applications working on distributed memory systems. Several parallel applications indeed rely on such technology to distribute messages across processes running on the same infrastructure and cooperating together to achieve an objective. MPI supports both point-to-point as well as collective communications. Actually, there are several implementations of the library such as OpenMPI [107], IntelMPI [108], MPICH [109] and many others. Most MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran and any language able to interface with such libraries, including C#, Java or Python.

The last definition of MPI interface includes two different sets of communication functions named respectively point-to-point and collective. In a point-to-point operation, just two processes are involved in the communication. Such processes are said *sender* and *receiver*. Examples of point-to-point operations are *MPI_Send*, *MPI_Recv* or *MPI_Wait*. In a collective operation, instead, many processes can be involved in the communication. Examples of collective operations are *MPI_BCast*, *MPI_Gather*, *MPI_Scatter* or *MPI_Reduce*.

In many message-passing libraries, the method by which the system handles messages is chosen by the library. By default, library uses the standard mode as method to manage messages and transmissions but although it gives acceptable reliability and performance for all possible communication scenarios, it may hide possible programming problems or may not give the best performance in specialized circumstances. To have more control over how the system handles the messages, programmers can select a proper communication mode to achieve better performance. There are four different communication modes [110], named respectively *standard*, *synchronous*, *ready* and *buffered*, which can be combined with *blocking* and *non-blocking* calls. Communication modes as well as the blocking policy should be decided by the programmer. Table A.1 summarizes sending and receiving calls which will be described in this chapter.

A.1 Blocking and non-blocking behaviour

Blocking functions suspend the application execution until the message buffer is safe to use, avoiding that data stored in the buffer may be modified by the application

TABLE A.1: MPI Sending and Receiving calls.

Communication Mode	Blocking functions	Non-blocking functions
Synchronous	MPI_SSend MPI_Recv	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND
Standard	MPI_SEND MPI_RECV MPI_SENDRECV MPI_SENDRECV_REPLACE	MPI_ISEND MPI_IRECV

before them are actually sent. Buffer overwriting might happen because usually the buffer used to contain data to send/receive is re-used. Blocking functions guarantee that data stored in the buffer will be not corrupted, stopping the application execution in order to avoid further buffer accesses until the data stored into the buffer are actually sent. The most typical blocking functions are MPI_Send and MPI_Recv.

Non-blocking functions instead separate communication from computation. Using such functions, application does not stop when non-blocking function is called, such that the application might get back to its computation. However, programmer should control access to the buffer to avoid corrupting data previously stored inside.

A.1.1 Blocking Synchronous Send

Figure A.1 shows two processes, *sender* (*S*) and *receiver* (*R*) involved in a blocking synchronous communication. They make use of *MPI_SSend* and *MPI_Recv*. At the beginning, both *S* and *R* are busy to do some computation (heavy straight and dashed lines). Then, *S* completes its computation-intensive job and invokes *MPI_SSend* function, which is a blocking and synchronous function. The execution of the sender *S* is blocked, then, the underlying MPI system sends a *ready to send* message to the receiver *R* which is still running in its computation. After a while, the receiver *R* completes its own job, then invokes the *MPI_Recv*, which sends back a *ready to receive* message to the sender. Due to the network latency, all messages are not sent and received immediately but it takes some time. When the sender (which is blocked yet) receives the *ready to receive* message, the data are then transferred. After completing the data transmission, the sender gets out from the blocking stage coming back to do its computational job. Due to the latency, the data are completely received after a while and until the transmission process is not completed, the receiver waits until the buffer is filled. In this communication there are two different overhead: *system overhead* is incurred for copying the message data from the sender's message buffer onto the network, and for copying the message data from the network into the receiver's message buffer; *synchronization overhead* is instead the time spent waiting for an event to occur on another task. In the figure A.1, the sender must wait for the receiver to be executed and for the handshake to arrive before the message can be transferred. The receiver also incurs some synchronization overhead in waiting for the handshake to complete. Synchronization overhead can be significant, not surprisingly, in synchronous mode. As we shall see, the other modes try different strategies for reducing this overhead. In this example the most

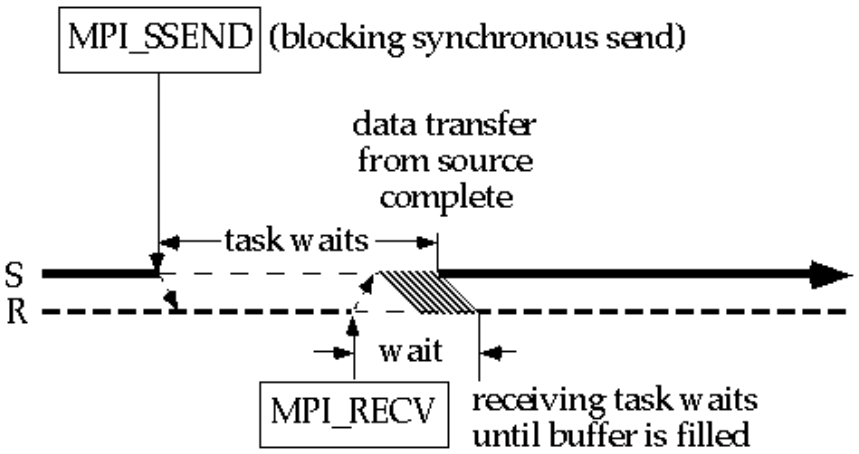

FIGURE A.1: An example of Blocking Synchronous Send

amount of synchronization overhead is experienced by the sender but if the receiver call precedes the sending, most of the synchronization overhead will be incurred by the receiver.

A.1.2 Blocking Ready Send

In the *blocking ready send* mode, represented in Fig. A.2, the receiver has to start the communication, sending to the sender a *ready to receive* message. After sending the notification, the receiver waits until the buffer is filled. When the sender completes its own computation, invokes the *MPI_RSend* function starting the data transfer. When the data are completely sent, the sender can get back to its computation. When the buffer is completely filled, the receiver can use the received data going back to its computation. In this model, if the *ready to receive* message hasn't arrived yet, the ready mode send will incur an error. Ready mode aims to minimize system overhead and synchronization overhead incurred by the sending task. In the blocking case, the only wait on the sending node is until all data have been transferred out of the sending task's message buffer. The receiver can still incur substantial synchronization overhead, depending on how much earlier it is executed than the corresponding send.

A.1.3 Blocking Buffered Send

The *blocking buffered send* mode uses a buffer where the message is stored before it is sent to the receiver. As shown in Fig. A.3 when the *MPI_BSend* function is invoked, the sender does not stop waiting for the receiver but the message is stored in a user-supplied buffer so that the sending task can proceed with computation that modify the original message buffer, knowing that these modifications will not be reflected in the data actually sent. The data will be copied from the user-supplied buffer over the network once the *ready to receive* notification has arrived. Buffered mode incurs extra system overhead, because of the additional copy from the message buffer to the user-supplied buffer. Synchronization overhead is eliminated on the sending task – the timing of the receiver is now irrelevant to the sender. Synchronization overhead can still be incurred by the receiving task. Whenever the receiver is executed before the sender, it must wait for the message to arrive before it can return.

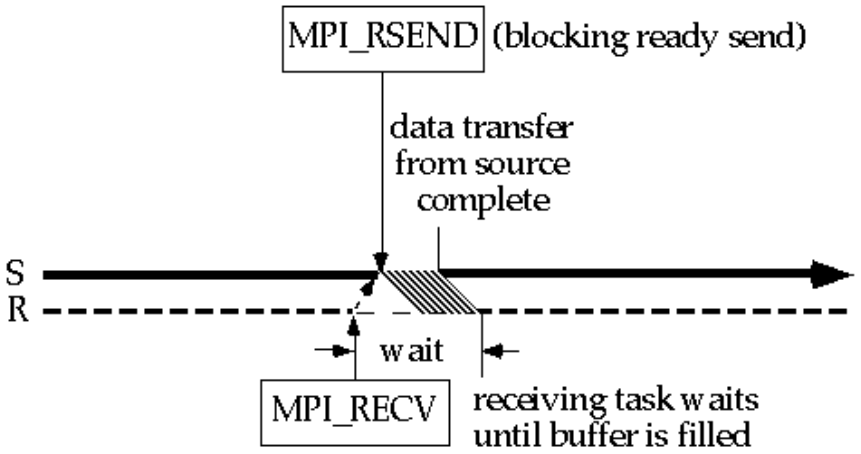

FIGURE A.2: An example of Blocking Ready Send

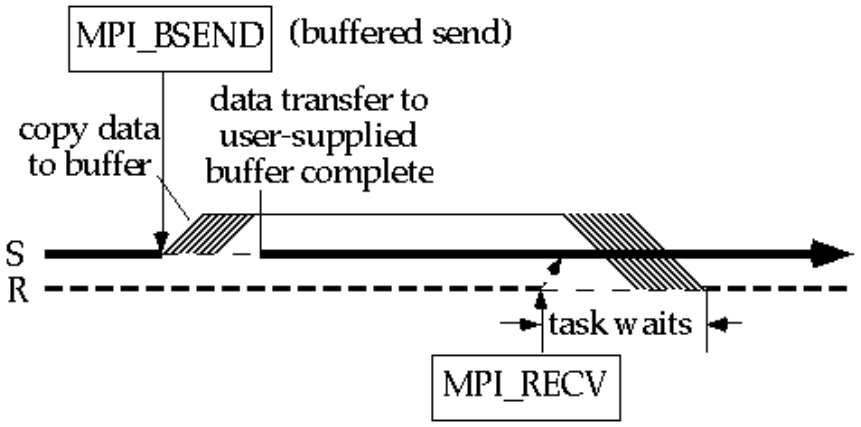

FIGURE A.3: An example of Blocking Buffered Send

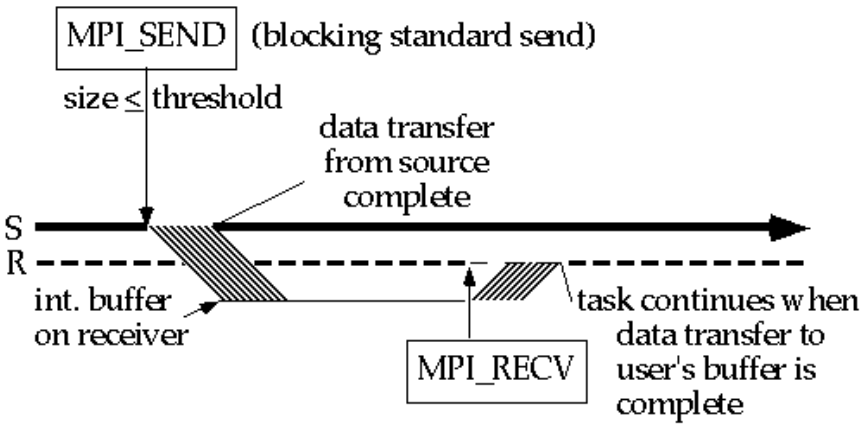

FIGURE A.4: An example of Blocking Standard Send with message smaller than the threshold

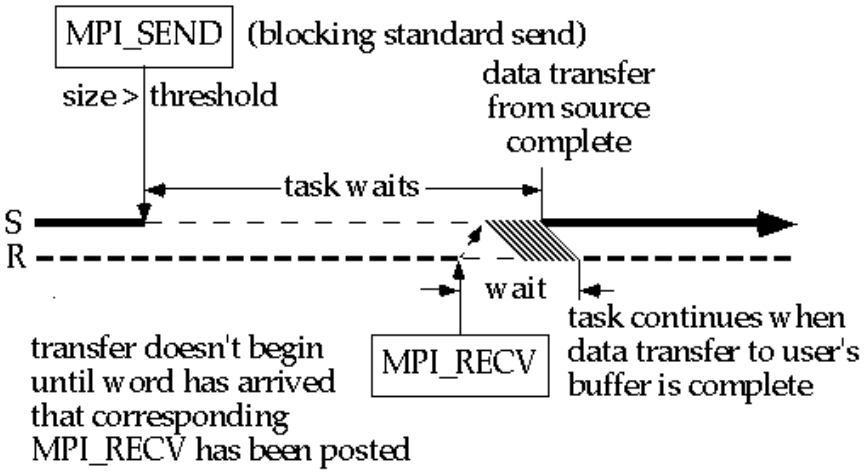

FIGURE A.5: An example of Blocking Standard Send with message larger than the threshold

A.1.4 Blocking Standard Send

The *blocking standard send* mode is the default behaviour adopted when traditional *MPI_Send* and *MPI_Recv* functions are used. Usually MPI implementations use two different algorithms, named *eager* and *rendezvous*, and deciding which algorithm to use is not defined by the MPI standard, but it is left up to implementers and usually depends on the message size and the number of tasks in the application. Figures A.4 and A.5 depict such mode. If the number of tasks and the message size are smaller than a threshold value, then the eager protocol is used, else the rendezvous protocol is used. Figure A.4 shows a communication between sender and receiver when the message size is smaller than the threshold. In this case, the blocking standard send *MPI_Send* copies the message over the network into a system buffer on the receiving node. The standard send then returns, and the sending task can continue computation. The system buffer is attached when the program is started – the user does not need to manage it in any way. There is one system buffer per task that will hold multiple messages. The message will be copied from the system buffer to the receiving task when the receive call is executed.

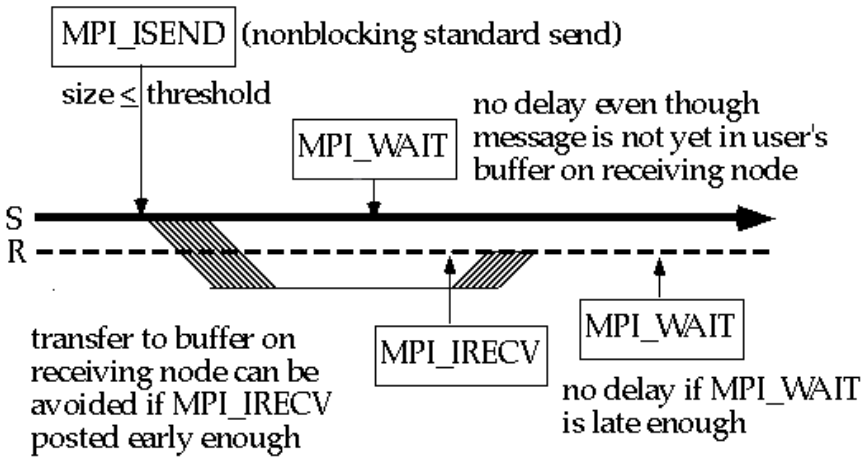

FIGURE A.6: An example of Non-Blocking Standard Send with message smaller than the threshold

As with buffered mode, the usage of a buffer decreases the likelihood of synchronization overhead on the sending task at the price of increased system overhead resulting from the extra copy to the buffer. As always, synchronization overhead can be incurred by the receiving task if a receive is posted first.

Unlike buffered mode, the sending task will not incur an error if the buffer space is exceeded. Instead, the sending task will block until the receiving task calls a receive that pulls data out of the system buffer. Thus, synchronization overhead can still be incurred by the sending task.

When the message size is greater than the threshold (Fig. A.5), the behaviour of the blocking standard send *MPI_Send* is essentially the same as for synchronous mode.

A.1.5 Non-blocking Standard Send

Figures A.6 and A.7 show a non-blocking standard send *MPI_Isend* and a non-blocking receive *MPI_Irecv*. As before, the standard mode send proceed differently depending on the message size. Figure A.6 demonstrates the behaviour for message size less than or equal to the threshold, whilst A.7 demonstrates the behaviour for message size greater than threshold.

The sending task posts the non-blocking standard send when the message buffer content is ready to be transmitted. It returns immediately without waiting for the copy to the remote system buffer to complete. *MPI_Wait* is called just before the sending task needs to overwrite the message buffer.

The receiving task calls a non-blocking receive as soon as a message buffer is available to hold the message. The non-blocking receive returns without waiting for the message to arrive. The receiving task calls *MPI_Wait* when it needs to use the incoming message data (i.e. needs to be certain that it has arrived).

The system overhead does not differ substantially from the blocking send and receive calls unless data transfer and computation can occur simultaneously. Since the CPU must perform both the data transfer and the computation, computation will be interrupted on both the sending and receiving nodes to pass the message. When the interruption occurs should not be of any particular consequence for the

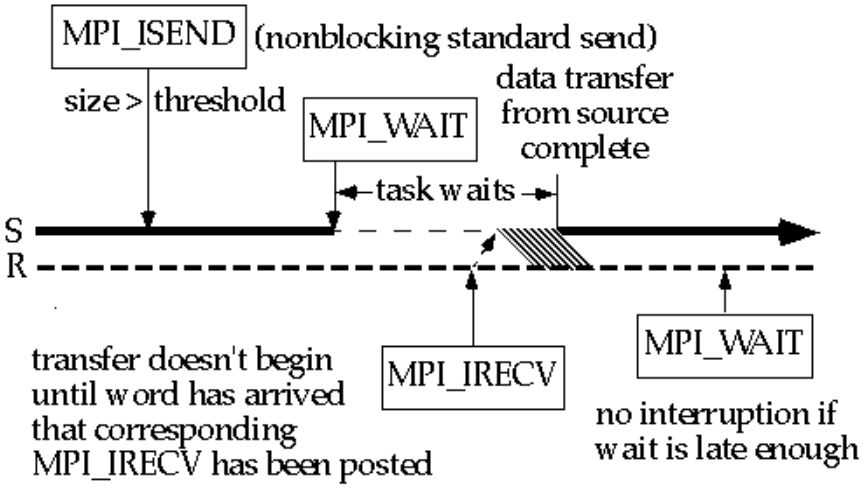

FIGURE A.7: An example of Non-Blocking Standard Send with message larger than the threshold

program that is running. Even for architectures that overlap computation and communication, the fact that this case applies only to small messages means that no great difference in performance would be expected.

The advantage of using non-blocking send occurs when the system buffer is full. In this case, a blocking send would have to wait until the receiving task pulled some message data out of the buffer. If a non-blocking call is used, computation can be done during this interval.

The advantage of a non-blocking receive over a blocking one can be considerable if the receive is posted before the send. The task can continue computing until the Wait is posted, rather than sitting idle. This reduces the amount of synchronization overhead.

The case of a non-blocking standard send *MPI_Isend* for a message larger than the threshold is more interesting (Fig. A.7). For a blocking send, the synchronization overhead would be the period between the blocking call and the copy over the network. For a non-blocking call, the synchronization overhead is reduced by the amount of time between the non-blocking call and the *MPI_Wait*, in which useful computation is proceeding.

Again, the non-blocking receive *MPI_Irecv* will reduce synchronization overhead on the receiving task for the case in which the receive is posted first. There is also a benefit of using a non-blocking receive when the send is posted first. Consider how the figure would change if a blocking receive were posted. Typically, blocking receives are posted immediately before the message data must be used (to allow the maximum amount of time for the communication to complete). So, the blocking receive would be posted in place of the *MPI_Wait*. This would delay the synchronization with the send call until this later point in the program, and thus increase synchronization overhead on the sending task.

A.2 What you should consider when you work with MPI

Running the same application on different parallel architectures might yield different performance results which depend on several factors. Here is a small list of such factors limiting the application performance:

- core number per CPU
- cache size
- CPU frequency
- amount of RAM
- RAM latency and bandwidth
- network latency and bandwidth
- code (pure MPI or hybrid)

As the number of factors is high, it becomes hard for an MPI implementation to be always efficient. Programmers then should tune the implementation parameters to fit the MPI implementation to the application. Using benchmark tools, such as Intel MPI Benchmark or MVAPICH2 MicroBenchmark, or profiling and tracing tools, such as Intel VTune, can help to understand which parameters are the best for an application. Even knowing the application communication model can help to get insight about how the application behaviour can be optimized just modifying the MPI parameters. Another factor which might affect the application performance is how processes are assigned to the CPU. Different implementations have different ways to do this but such behaviour can be controlled by several environment variables such as *I_MPI_PIN_PROCESSOR_LIST* and *I_MPI_PIN_DOMAIN* for IntelMPI implementation and*MV2_CPU_BINDING_POLICY*, *MV2_CPU_BINDING_LEVEL* and *MV2_CPU_MAPPING* for MVAPICH2.

As described above, all MPI implementations use two different protocols for sending messages: *eager* and *rendezvous*. Choosing which protocol to use depends on the message size and the number of processes involved in the communication. Eager protocol is used for sending small messages. The sender sends the message without taking care if receiver is ready to receive it. The message, indeed, is stored in a buffer and will be restored by the receiver when it is ready. Rendezvous protocol instead is used for large messages which might not be stored in the buffer. This protocol forces sender and receiver to be synchronized before the message is sent. Both protocols have pros and cons. Eager protocol indeed can reduce the synchronization time but requires to use a buffer which might be completely filled if the amount of messages to be sent is high. Rendezvous protocol instead introduces a higher delay due to the handshake between sender and receiver. Although all implementations switch between both protocols in an automatic way, programmers can control the usage of such protocols setting up some environment variables such as *I_MPI_EAGER_THRESHOLD* for the Intel implementation, *MV2_IBA_EAGER_THRESHOLD* for the MVAPICH2 implementation and the options *–mca btl_openib_eager_limit* and *–mca btl_openib_rndv_eager_limit* for the Open-MPI implementation.

Parallel architectures are usually built with hundreds or even thousands nodes each of them equipped with tens of cores. In distributed applications, processes running on each core cooperate together sending and receiving messages. When the communication happens among two processes running on two cores belonging to the same node, the communication is said *intranode*. If processes run on two different nodes, instead, the communication is said *internode*. Intranode communications are less expensive than the internode ones, because in an internode communication the message is sent across the network which introduces delay due to its latency and

bandwidth. The intranode communications instead are usually solved sending and receiving messages through shared memory. This approach is used in all MPI implementations but it works fine for small messages. When messages get larger, shared memory might not be the right solution because of the same problems well-known as *tie down CPU* and *cache pollution*. For this reason, another method to deliver messages has been introduced. Such approach is named *Kernel Assisted*, which is an ideal method for both small and large messages. This approach requires the usage of some kernel module, such as *LiMIC* and *KNEM*, and some kernel features, such as *Cross Memory Attach*. CMA has been introduced in the Linux kernel 3.2 and is actually supported by IntelMPI, MVAPICH2 and OpenMPI. It is usually used by default when large messages are sent, but programmers can control such approach using the environment variable *I_MPI_SHM_LMT* (for IntelMPI implementation) to specify if shared memory or kernel assisted has to be used.

Appendix B

Measuring network performance

Applications designed and developed to be executed on a target architecture might behave differently, sometimes worse, when they are executed on a different infrastructure.

Predicting application performance is an interesting challenge and many authors developed methods and methodology for modelling and predicting the MPI programs performance. In [111] for instance, authors define PEMPIs, a new methodology to model and predict MPI application performance. It is based on an analytical model which is derived from a graph representing relationships between tasks (computation) and processes (communications), and from the time spent by the application to compute each task and communicate to each others. A different approach for the performance application prediction is described in [112, 113] where authors developed a tool, named dPerf, based on the Rose compiler, which is able to decompose the source code into an intermediate representation, name IR. Then, using PAPI library and SimGrid as simulation environment, authors showed how to predict the performance of an application named NAS Integer Sort. A lower level methodology is instead defined in [114]. In this work, the total elapsed time is computed summing up the communication time and the computation time. The authors suggest a way to build the analytical model which is validated with several applications being run first on the MareNostrum HPC system and then on a simulated environment using Dimemas.

Sadly, predicting the right application performance is not easy, mostly for parallel applications, because their performance might be affected by several factors, some related to the hardware, such as memory, processors and network infrastructure, others instead strictly related to the application, the communication model and the operating system. Perhaps the two most important factors are the interconnection network and the communication model, which might limit heavily application scalability and performance. Many efforts have been done to reduce the side effect of the communication, trying to boost the performance of the underlying interconnection network by introducing new technology, such as Infiniband, that increase bandwidth and reduce latency. Many other researches instead focused on boosting performance from the MPI implementation perspective, and then developing new MPI libraries mainly optimized to be used in specific architectures.

Authors in [37] describe three different ways to optimize MPI functions:

1. **Optimization below the MPI layer**: this optimization relies on changing parameters below the MPI layer which affect the MPI communications. Some tests, for example, have demonstrated that changing the TCP window size impact the MPI communication performance;

2. **Optimization within the MPI layer**: in this optimization, the MPI library is partially rewritten or extended in order to use new technology. For example,

some MPI implementations might be optimized for being used on the Infiniband infrastructure as they make use of RDMA call. Others instead might have directly access to some kernel module, to control, for example, process migration among cores (CPU affinity) or even the usage of the eager and rendezvous protocols which might affect the point-to-point and collective communications.

3. **Optimization on top of MPI layer**: in this optimization, collective functions are rewritten or extended in order to exploit the physical network topology and then reducing the total amount of messages exchanged among processes. To be comparable, different implementations of the same collective function need to be measured. There are two different approaches to make a performance analysis of a collective function. The first is by using an analytical model, which is the topic of this chapter; the second instead, is by using an empirical (experimental) approach:

   (a) **Analytical model approach**: in this method, each implementation is translated into a mathematical expression which is able to describe a collective function. Different implementations of the same collective function have different analytical models. By comparing both models, researchers can understand which implementation performs better on a given infrastructure. There are three different approaches to define an analytical model able to describe the dynamic behaviour of the application: *scalar parameters*, *functions* and *statistic models*. The first approach uses a set of scalar parameters to model the application behaviour. Such parameters describe the application behaviour under fixed conditions; the second one instead makes use of function to have a better flexibility; the last approach instead makes use of statistic methods (such as the Markov Model);

   (b) **Empirical approach**: In this method, the performance of the different implementations are measured running both algorithms on the target infrastructure. Performance are measured using profiling and tracing tools, then the results are compared. Sometimes, instead, the physical network infrastructure is emulated using simulator, such as Dimemas. Simulators are able to emulate the real network infrastructure setting up parameters such the number of nodes in the network, the number of CPU per each node, the number of links for each node, the number of buses for each link, the transmission directory for each link (half-duplex or full-duplex), the resource contention level, the bandwidth for intranode and internode communications, the latency for intranode and internode communication and the processor speed.

B.1 Some analytical models

Performance of a collective function can be evaluated using an analytical model describing the collective function in terms of point-to-point functions and the underlying infrastructure network. In literature there are several models which can be used to describe a point-to-point communication [115–121]. Here a brief description of some.

- **Honkney model**: this model assumes that the time to send a message of size m between two nodes is $a + bm$ where a is the latency for each message and b is the transfer time per byte or reciprocal of network bandwidth;

- **LogP**: this model describes a network in term of latency L, overhead o, gap per message g, and number of nodes involved in the communication P. The time to send a message between two nodes according to this model is $L + 2o$. LogP assumes that only constant-size, small messages are sent among nodes;
- **LogGP**: this model is an extension of LogP model that additionally allows for large messages by introducing the gap per byte parameter G. LogGP model predicts the time to send a message of size m between two nodes as $L + 2o + (m - 1)G$;
- **PLogP**: is a further extension of the LogP model which we decided to use in all the experiments described in this work. Next paragraph covers all about such model.

B.2 PLogP model

PLogP [122] model is able to capture the relevant aspects of message passing in distributed systems, describing a point-to-point communication just by using five parameters: P is the number of the processors; L is the end-to-end latency from process to process, that combines all contributing factors such as copying data to and from network interfaces and transfer over the physical network; $o_s(m)$, $o_r(m)$ and $g(m)$ are respectively send overhead, receive overhead and gap. The function $g(m)$ is the minimum time interval between consecutive message transmissions or receptions. It is the reciprocal value of the end-to-end bandwidth from process to process for messages of a given size m. Overall, the time for sending a message of size m between two nodes in the pLogP model [56] is

$$T = L + g(m) \tag{B.1}$$

A graphical representation of all pLogP parameters is shown in Fig. B.1. To have a numerical representation describing the time spent by a sender to send a message to a receiver, all pLogP parameters need to be gathered and combined together according to the formula B.2. In [102] authors describe a method for measuring all pLogP parameters. They also developed a tool, named *logp_mpi*, to measure such parameters. The tool can be downloaded from [123]. Such tool has been used in our experiments in order to get all required pLogP parameters.

B.3 PLogP Drawback

The pLogP parameters are strictly related to the underlying physical network infrastructure. Because of this reason, neither application-related overheads nor temporary network congestion and resource contentions are detected. These factors might increase the time spent by a sender waiting until all required physical resources are free. As pLogP model does not include congestion and contention component on its model, it might underestimate completion time for collective operations. Therefore, during performance prediction, analysts should take care about the performance loss due to these factors.

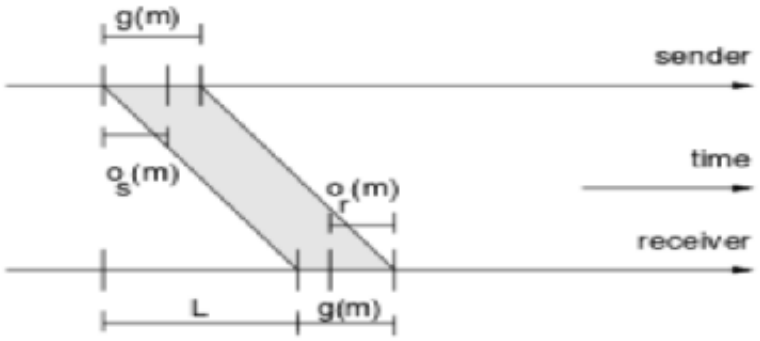

Figure 4: The pLogP parameters

FIGURE B.1: pLogP parameters

B.4 Extending pLogP model for collective operations

The pLogP model described in the previous paragraph can be applied just for point-to-point functions. For collective functions instead that model can be still applied, but it has to be extended, according to the implementation of the collective functions. Extending the model for a specific implementation might help developers to understand the performance of the implementation and compare them with the performance of a different one. Let's consider the *Broadcast* function implemented using the linear algorithm. Let's suppose to have n processes and the master process wants to broadcast a m-byte message (Fig. B.2). It starts sending the message to the first receiver. After a while, depending on the message size and the network bandwidth, the message is completely sent. At this time, although the receiver has not completely received the message yet (because of the network latency), the sender starts sending the message to the second receiver. The collective operation is then completed when the message has been completely received by the other *n-1* receivers. Figure B.2 shows such behaviour which can be described analytically using the pLogP model as follows:

$$T_{BROADCAST} = L + (n-1) * g(m) \tag{B.2}$$

Many works [40, 41, 115, 124–126] tried to define different implementations of collective functions particularly optimized for specific architecture and under specific conditions. A list of the algorithms used by the collective operations in Intel MPI implementation can be found in [127].

B.5 PLogP models for collective functions

As described in the previous paragraph, all collective functions can be analytically described using pLogP. This is usually done for two main reasons: 1) to evaluate the complexity of a collective function to select the most performing one; 2) to predict the communication time spent to complete the collective function. As described above, different implementations of the same collective function yield different analytical

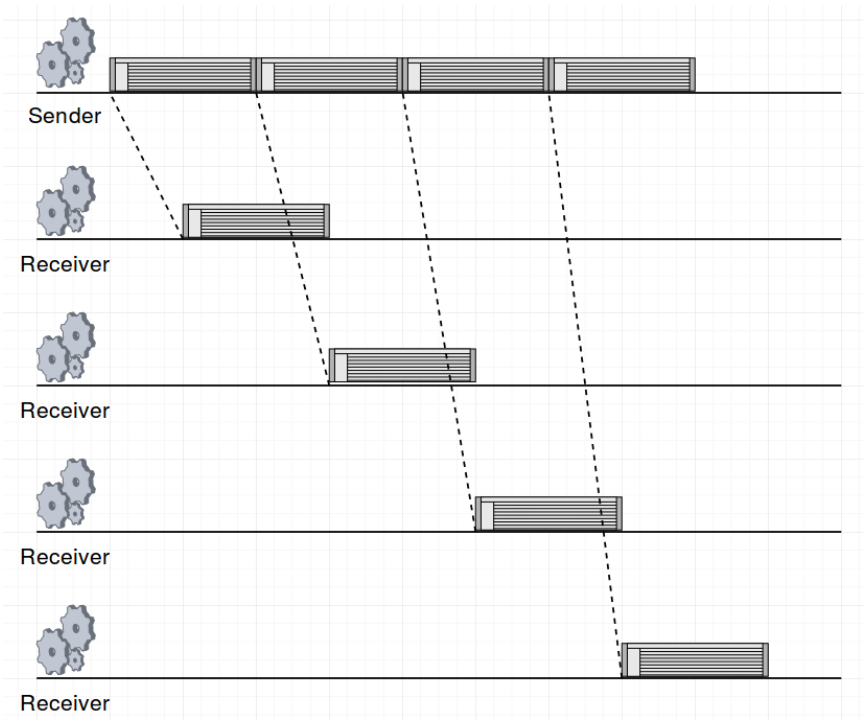

FIGURE B.2: An example of the Broadcast function

models. Table B.1 shows the pLogP model corresponding to different implementations of many collective functions, supposing to use a number of processor P power of 2 and the message is split in just one segment having a size equals to m.

B.6 Working with heterogeneous architectures

Measuring the performance of a collective function becomes harder when the architecture is heterogeneous, that is for those network infrastructures where the communication between cores is not always the same but depends on some factor. For example, the BlueGene supercomputer makes use of a Torus network, that is a heterogeneous network from the latency perspective. The communication between two nodes, indeed, depends on the number of hops the message needs to done before being received from the receiver.

To make a correct analytical model for heterogeneous architectures, an exhaustive comprehension of the underlying network infrastructure needs to be done. It would be useful to know 1) how latency and bandwidth changes among cores and 2) the network symmetry (the time spent by a sender to send a message to a receiver is as equal as the time spent for getting back the response message). After having understood the underlying interconnection network, the whole architecture can be subdivided in virtual clusters having homogenous nodes. Then the pLogP parameters can be gathered just between two nodes belonging the same cluster and between two nodes belonging to different clusters. Then if N is the number of nodes belonging to the parallel architecture and C is the number of clusters, instead of gathering *N*(N-1)* pLogP parameters, it will be needed just *C*(C-1) + C* measurements which can be halved if links are symmetric [125].

TABLE B.1: PLogP models for some collective functions.

MPI Function	**Algorithm**	**pLogP Model**
Barrier [115]	Flat-Tree	$T_{min} = P*g(0) + 2*L$ $T_{max} = P*(g+o_r) + 2*(L-o_r)$
	Double Ring	$T = 2*P*(L+g(0))$
	Recursive Doubling	$T = log_2(P)*(L+g(0))$
Barrier [115]	Bruck	$T = log_2(P)*(L+g(0))$
Broadcast [115, 124, 125]	Linear, Flat-Tree	$T = L + (P-1)*g(m)$
	Flat-Tree Rendezvous	$T=(P-1)*g(m)+2*g(1)+3*L$
	Pipeline, Chain	$T = (P-1)*(L+g(m))$
	Chain Rendezvous	$T = (P-1)*(g(m)+2*g(1)+3*L)$
	Binomial	$T = log_2(P)*(L + g(m))$
	Binary	$T = (upper(log_2(P+1))-1)*(L + 2*g(m)$
	Binary Tree	$T = log_2(P)*(2*g(m) + L)$
	Splitted-binary	$T = (upper(log_2(P+1))-1)*(L + 2*g(m))$
	Binomial Tree Rendezvous	$T = log_2(P)*g(m) + log_2(P)*(2*g(1)+3*L)$
Reduce [115]	Flat Tree	$T = L + (P-1)*max(g(m), o_r(m))$
	Pipeline	$T = (P-1)*(L+max(g(m), o_r(m))$
	Binomial	$T = upper(log_2(P))*(L+max(g(m), o_r(m)+o_s(m))$
	Binary	$T = (upper(log_2(P+1))-1)*(L+2*max(g(m), o_r(m)))$
AllToAll [115]	Linear	$T = P*L+(P-1)*(P+1-P/2)*g(m)$
	Pairwise exchange	$T = (P-1)*(L+g(m))$
Scatter [124]	Flat Tree	$T = (P-1)*g(m) + L$

Bibliography

[1] G. Carlyle, S. L. Harrell and P. M. Smith, Cost-Effective HPC: The Community or the Cloud?, 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Indianapolis, IN, 2010, pp. 169-176.

[2] Rashid Hassani, Md Aiatullah, Peter Luksch, Improving HPC Application Performance in Public Cloud, In IERI Procedia, Volume 10, 2014, Pages 169-176, ISSN 2212-6678.

[3] M. Mancini and G. Aloisio, How advanced cloud technologies can impact and change HPC environments for simulation, 2015 International Conference on High Performance Computing & Simulation (HPCS), Amsterdam, 2015, pp. 667-668.

[4] Yang, Tao, Xiaosong Ma, and Frank Mueller. Predicting parallel applications performance across platforms using partial execution. ACM/IEEE Supercomputing Conference. 2005.

[5] Chakthranont, N., Khunphet, P., Takano, R., & Ikegami, T. (2014, December). Exploring the performance impact of virtualization on an HPC cloud. In Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on (pp. 426-432). IEEE.

[6] Expsito, R. R., Taboada, G. L., Ramos, S., Tourino, J., & Doallo, R. (2013). Performance analysis of HPC applications in the cloud. Future Generation Computer Systems, 29(1), 218-229.

[7] Dror G. Feitelson, Dan Tsafrir, David Krakov, Experience with using the Parallel Workloads Archive, Journal of Parallel and Distributed Computing, Volume 74, Issue 10, 2014, Pages 2967-2982, ISSN 0743-7315,

[8] Gonzalo P. Rodrigo, P.-O. Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, Lavanya Ramakrishnan, Towards understanding HPC users and systems: A NERSC case study, Journal of Parallel and Distributed Computing, Volume 111, 2018, Pages 206-221, SSN 0743-7315,

[9] S. Di, D. Kondo and W. Cirne, "Characterization and Comparison of Cloud versus Grid Workloads," 2012 IEEE International Conference on Cluster Computing, Beijing, 2012, pp. 230-238.

[10] Pascale Minet, Eric Renault, Ines Khoufi, Selma Boumerdassi. Data analysis of a Google data center. CCGRID 2018 : 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2018, Washington Dc, United States. IEEE Computer Society, Proceedings CCGRID 2018 : 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp.342 - 343, 2018

[11] Hussain, A.; Aleem, M. GoCJ: Google Cloud Jobs Dataset for Distributed and Cloud Computing Infrastructures.Data 2018, 3, 38.

[12] Liu, Zitao & Cho, Sangyeun. (2012). Characterizing Machines and Workloads on a Google Cluster. Proceedings of the International Conference on Parallel Processing Workshops. 397-403. 10.1109/ICPPW.2012.57.

[13] Emeras, J., Ruiz, C., Vincent, J. M., & Richard, O. (2013, May). Analysis of the jobs resource utilization on a production system. In Workshop on Job Scheduling Strategies for Parallel Processing (pp. 1-21). Springer, Berlin, Heidelberg.

[14] Smith, W., Taylor, V., & Foster, I. (1999, April). Using run-time predictions to estimate queue wait times and improve scheduler performance. In Workshop on Job scheduling strategies for Parallel Processing (pp. 202-219). Springer, Berlin, Heidelberg

[15] Kianpisheh, Somayeh & Jalili, Saeed & Charkari, Nasrolah. (2012). Predicting Job Wait Time in Grid Environment by Applying Machine Learning Methods on Historical Information.

[16] Kumar R., Vadhiyar S. (2015) Prediction of Queue Waiting Times for Metascheduling on Parallel Batch Systems. In: Cirne W., Desai N. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2014. Lecture Notes in Computer Science, vol 8828. Springer, Cham

[17] Andresen, D., Hsu, W., Yang, H., & Okanlawon, A. (2018). Machine Learning for Predictive Analytics of Compute Cluster Jobs. arXiv preprint arXiv:1806.01116.

[18] A. Matsunaga and J. A. B. Fortes, "On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications," 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, VIC, 2010, pp. 495-504.

[19] Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta. 2006, August 24. Retrieved from https://aws.amazon.com/it/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2—beta/

[20] The NIST Definition of Cloud Computing. 2011, September. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-145/final

[21] Dillon, T., Wu, C., & Chang, E. (2010, April). Cloud computing: issues and challenges. In Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on (pp. 27-33). Ieee.

[22] Google Trends. 2018, November. Retrieved from https://trends.google.com/trends/explore?date=2006-10-01%202018-11-27&q=%2Fm%2F02y_9m3

[23] Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019. 2019, April 2. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g

[24] Gartner Says Worldwide IaaS Public Cloud Services Market Grew 29.5 Percent in 2017. 2018, August 1. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2018-08-01-gartner-says-worldwide-iaas-public-cloud-services-market-grew-30-percent-in-2017

[25] AWS Simple Montly Calculator. 2019, August 30. Retrieved from https://calculator.s3.amazonaws.com/index.html

[26] Azure Price Calculator. 2019, August 30. Retrieved from https://azure.microsoft.com/it-it/pricing/calculator/

[27] Google Cloud Platform Pricing Calculator. 2019, August 30. Retrieved from https://cloud.google.com/products/calculator/

[28] IBM Cloud pricing. 2019, August 30. Retrieved from https://www.ibm.com/cloud/pricing

[29] Cloud Computing Comparison Engine. 2019, August 30. Retrieved from https://www.cloudorado.com/cloud_server_comparison.jsp

[30] Study finds moving some computer services to cloud would save significant energy. 2013, June. Retrieved from https://phys.org/news/2013-06-cloud-significant-energy.html

[31] Expósito, R. R., Taboada, G. L., Ramos, S., Touriño, J., & Doallo, R. (2013). Performance analysis of HPC applications in the cloud. Future Generation Computer Systems, 29(1), 218-229.

[32] T. Passerini, J. Slawinski, U. Villa and V. Sunderam, "Experiences with Cost and Utility Trade-offs on IaaS Clouds, Grids, and On-Premise Resources," 2014 IEEE International Conference on Cloud Engineering, Boston, MA, 2014, pp. 391-396.

[33] A. G. Carlyle, S. L. Harrell and P. M. Smith, "Cost-Effective HPC: The Community or the Cloud?," 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Indianapolis, IN, 2010, pp. 169-176.

[34] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., ... & Wright, N. J. (2010, November). Performance analysis of high performance computing applications on the amazon web services cloud. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on (pp. 159-168). IEEE.

[35] Hybrid Parallelism: A MiniFE* Case Study. 2016, October 13. Retrieved from https://software.intel.com/en-us/articles/hybrid-parallelism-a-minife-case-study

[36] Rashid Hassani, Md Aiatullah, Peter Luksch, Improving HPC Application Performance in Public Cloud, In IERI Procedia, Volume 10, 2014, Pages 169-176, ISSN 2212-6678.

[37] Dichev, K., Lastovetsky, A.: Optimization of collective communication for heterogeneous HPC platforms. In: High-Performance Computing on Complex Environments, Wiley, pp. 95–114. Wiley (2014)

[38] Intel® MPI Library Collective Optimization on the Intel® Xeon Phi™ Coprocessor Using Environment Variable Collective Operation Control. 2015, December 18. Retrieved from https://software.intel.com/en-us/articles/intel-mpi-library-collective-optimization-on-intel-xeon-phi

[39] Rabenseifner R., Träff J.L. (2004) More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. In: Kranzlmüller D., Kacsuk P., Dongarra J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2004. Lecture Notes in Computer Science, vol 3241. Springer, Berlin, Heidelberg

[40] Rajeev Thakur and William Gropp, "Improving the Performance of Collective Operations in MPICH," in Proc. of the 10th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2003), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, LNCS 2840, Springer, September 2003, pp. 257-267.

[41] S. S. Vadhiyar, G. E. Fagg and J. Dongarra, "Automatically Tuned Collective Communications," Supercomputing, ACM/IEEE 2000 Conference, 2000, pp. 3-3.

[42] Y. Gong, B. He and J. Zhong, "Network Performance Aware MPI Collective Communication Operations in the Cloud," in IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 11, pp. 3079-3089, Nov. 2015.

[43] Ramakrishnan, L., Canon, R. S., Muriki, K., Sakrejda, I., & Wright, N. J. (2012). Evaluating interconnect and virtualization performance for high performance computing. ACM SIGMETRICS Performance Evaluation Review, 40(2), 55-60.

[44] Zhai, Y., Liu, M., Zhai, J., Ma, X., & Chen, W. (2011, November). Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. In State of the Practice Reports (p. 11). ACM.

[45] Chakthranont, N., Khunphet, P., Takano, R., & Ikegami, T. (2014, December). Exploring the performance impact of virtualization on an HPC cloud. In Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on (pp. 426-432). IEEE.

[46] High-performance computing in the cloud? 2016, October 24. Retrieved from https://www.csc.fi/-/high-performance-computing-in-the-cloud

[47] Real World AWS Scalability. 2016, November 7. Retrieved from https://aws.amazon.com/it/blogs/compute/real-world-aws-scalability/

[48] Solving I/O Bottlenecks to Enable Superior Cloud Efficiency. 2019, August 30. Retrieved from http://www.mellanox.com/related-docs/whitepapers/WP_Solving_IO_Bottlenecks.pdf

[49] Viktor Mauch, Marcel Kunze, Marius Hillenbrand, High performance cloud computing, In Future Generation Computer Systems, Volume 29, Issue 6, 2013, Pages 1408-1416, ISSN 0167-739X.

[50] E. S. Jung, R. Kettimuthu, "Challenges and opportunities for data-intensive computing in the cloud", IEEE Computer Society, December 2014, pp. 82-85.

[51] Napper, Jeffrey, and Paolo Bientinesi. "Can cloud computing reach the top500?." Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop. ACM, 2009.

[52] Amazon EC2 C3 Instance cluster. 2019, August 30. Retrieved from https://www.top500.org/system/178321

[53] Nanath, K., & Pillai, R. (2013). A model for cost-benefit analysis of cloud computing. Journal of International Technology and Information Management, 22(3), 6.

[54] Marconi, the new Tier-0 system. 2018, May 16. Retrieved from http://hpc.cineca.it/hardware/marconi

[55] Slurm Overview. 2019, August 30. Retrieved from https://slurm.schedmd.com/overview.html

[56] Steffenel, Luiz Angelo, and Grégory Mounié. "A framework for adaptive collective communications for heterogeneous hierarchical computing systems." Journal of Computer and System Sciences 74.6 (2008): 1082-1093.

[57] Machined types. 2018, May 16. Retrieved from https://cloud.google.com/compute/docs/machine-types

[58] Overview of Virtual Private Cloud. 2018, May 16. Retrieved from https://cloud.google.com/vpc/docs/vpc

[59] Andromeda 2.1 reduces GCP's intrazone latency by 4%. 2018, May 16. Retrieved from https://cloudplatform.googleblog.com/2017/11/Andromeda-2-1-reduces-GCPs-intra-zone-latency-by-40-percent.html

[60] Enter the Andromeda zone: Google Cloud Platform's latest networking stack. 2018, May 16. Retrieved from https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html

[61] Egress throughput caps. 2018, May 16. Retrieved from https://cloud.google.com/compute/docs/networks-and-firewalls#egress_throughput_caps

[62] Protein Data Bank. 2018, May 16. Retrieved from https://www.rcsb.org

[63] 11. Badr G, Al-Turaiki I, Mathkour H. Classification and assessment tools for structural motif discovery algorithms. BMC Bioinformatics. 2013;14 Suppl 9(Suppl 9):S4.

[64] 12. Shapiro, J. and Brutlag, D. (2004), FoldMiner: Structural motif discovery using an improved superposition algorithm. Protein Science, 13: 278-294.

[65] 13. Timothy L. Bailey, Mikael Boden, Fabian A. Buske, Martin Frith, Charles E. Grant, Luca Clementi, Jingyuan Ren, Wilfred W. Li, William S. Noble; MEME SUITE: tools for motif discovery and searching, Nucleic Acids Research, Volume 37, Issue suppl 2, 1 July 2009, Pages W202W208,

[66] 14. E. Krissinel and K. Henrick,Secondary-structure matching (SSM), a new tool for fast protein structure alignment in three dimensions; Acta Crystallographica Section D 2004;60(12):2256-2268

[67] 15. Liisa Holm, Chris Sander, Protein Structure Comparison by Alignment of Distance Matrices, Journal of Molecular Biology, Volume 233, Issue 1, 1993, Pages 123-138, ISSN 0022-2836,

[68] Shi S, Zhong Y, Majumdar I, Krishna SS, Grishin NV. Searching for three- dimensional secondary structural patterns in proteins with ProSMoS. Bioinformatics 2007; 23(11):1331–1338

[69] Shi S, Chitturi B, Grishin NV. ProSMoS server: a pattern-based search using interaction matrix representation of protein structures. Nucleic Acids Research 2009; 37(Web Server issue):W526–W531.

[70] Hutchinson G, Thornton JM. PROMOTIF—a program to identify and analyze structural motifs in proteins. Protein Science 1996; 5:212–220

[71] Dror O, Benyamini H, Nussinov R, Wolfson H. MASS: multiple structural alignment by secondary structures. Bioinformatics 2003; 19(1):i95–i104.

[72] Ferretti M, Musci M. Geometrical Motifs Search in Proteins: A Parallel Approach. Parallel Computing February 2015; 42:60–74.

[73] Ballard DH. Generalizing the Hough transform to detect arbitrary shapes. Pattern Recognition 1981; 13(2): 111-122

[74] Ferretti M, Musci M, Santangelo L. A Hybrid OpenMP and OpenMPI Approach to Geometrical Motif Search in Proteins. Proceedings of the IEEE International Conference on Cluster Computing (IEEE Cluster 2014), IEEE Computer Society, 2014; 298–304.

[75] Ferretti M, Musci M, and Santangelo L. "MPI-CMS: a hybrid parallel approach to geometrical motif search in proteins." Concurrency and Computation: Practice and Experience 27.18 (2015): 5500-5516.

[76] Murzin A. G., Brenner S. E., Hubbard T,. Chothia C. (1995). SCOP: a structural classification of proteins database for the investigation of sequences and structures. J. Mol. Biol. 247, 536-540

[77] Quarteroni, A., & Valli, A. (2008). Numerical approximation of partial differential equations (Vol. 23). Springer Science & Business Media.

[78] Formaggia, L., Quarteroni, A., & Veneziani, A. (Eds.). (2010). Cardiovascular Mathematics: Modeling and simulation of the circulatory system (Vol. 1). Springer Science & Business Media.

[79] Bertagna, Luca & Deparis, Simone & Formaggia, Luca & Forti, Davide & Veneziani, Alessandro. (2017). The LifeV library: engineering mathematics beyond the proof of concept.

[80] Bertagna, L., Deparis, S., Forti, D., Formaggia, L., & Veneziani, A. (2016), "The LifeV library: engineering mathematics beyond the proof of concept", Tech Report Dept. Math & CS, Emory University, TR2016-008, www.mathcs.emory.edu

[81] M. A. Heroux et al., "An overview of the trilinos project", ACM Trans. Math. Softw., vol. 31, no. 3, pp. 397-423, 2005

[82] The LifeV library: engineering mathematics beyond the proof of concept. 2017. Retrieved from https://www.mate.polimi.it/biblioteca/add/qmox/53-2017.pdf

[83] Avoid Heap Contention among threads. 2017, July 31. Retrieved from https://software.intel.com/en-us/articles/avoiding-heap-contention- among-threads

[84] Ran Liu, Haibo Chen, SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability, Proceedings of the Asia-Pacific Workshop on Systems, p.1-6, July 23-24, 2012, Seoul, Republic of Korea.

[85] Boost C++ Libraries. 2017, July 31. Retrieved from http://www.boost.org/doc/libs/1_45_0/boost/algorithm/string/trim.hpp

[86] Std::Locale CPPRefernece. 2017, July 31. Retrieved from http://en.cppreference.com/w/cpp/locale/locale

[87] Ferretti, M., & Santangelo, L. (2018, March). Hybrid OpenMP-MPI parallelism: porting experiments from small to large clusters. In 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) (pp. 297-301). IEEE.

[88] Ferretti M, Musci M, and Santangelo L. MPI–CMS: a hybrid parallel approach to geometrical motif search in proteins. Concurrency and Computation: Practice and Experience 27.18 (2015): 5500 5516.

[89] Ferretti, M., & Santangelo, L. (2018, September). Protein secondary structure analysis in the cloud. In Proceedings of the 6th International Workshop on Parallelism in Bioinformatics (pp. 63-70). ACM.

[90] Ferretti, M., Santangelo, L., & Musci, M. (2019). Optimized cloud-based scheduling for protein secondary structure analysis. The Journal of Supercomputing, 75(7), 3499-3520.

[91] K. A. Huck and J. Labarta, "Detailed Load Balance Analysis of Large Scale Parallel Applications," 2010 39th International Conference on Parallel Processing, San Diego, CA, 2010, pp. 535-544.

[92] Marconi, the new Tier-0 system. 2017, July 21. Retrieved from http://hpc.cineca.it/hardware/marconi

[93] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general- purpose distributed computing systems," in IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 141-154, Feb 1988.

[94] Plastino, Alexandre, Celso C. Ribeiro, and N. R. Rodriguez. "Load balancing algorithms for SPMD applications." Submitted for publication (2001).

[95] K. Skenteridou and H. D. Karatza, "Job scheduling in a grid cluster," 2015 International Conference on Computer, Information and Telecommunication Systems (CITS), Gijon, 2015, pp. 1-5

[96] S. Kushwaha and S. Kumar, "Analysis of list scheduling algorithms for parallel system," 2014 International Conference on High Performance Computing and Applications (ICHPCA), Bhubaneswar, 2014, pp. 1-6.

[97] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, Nancy Wilkins-Diehr, "XSEDE: Accelerating Scientific Discovery", Computing in Science & Engineering, vol.16, no. 5, pp. 62-74, Sept.-Oct. 2014

[98] Stampede User Guide. 2017, July 30. Retrieved from https://portal.xsede.org/tacc-stampede

[99] Comet System Overview. 2017, July 30. Retrieved from https://portal.xsede.org/sdsc-comet

[100] Billing policy on Marconi. 2017, July 30. Retrieved from http://www.hpc.cineca.it

[101] Auricchio, F., Fedele, M., Ferretti, M., Lefieux, A., Romarowski, R., Santangelo, L., & d VENEZIANI, A. (2018). Benchmarking a hemodynamics application on Intel based HPC systems. Paral Comput Everywhere, 32, 57.

[102] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. 2000. Fast Measurement of LogP Parameters for Message Passing Platforms. In Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing (IPDPS '00), José D. P. Rolim (Ed.). Springer-Verlag, London, UK, UK, 1176-1183.

[103] M. Mao, H. Humphrey, A performance study on the VM startup time in the cloud, IEEE 5th International Conference on Cloud Computing, June, IEEE, 2012, pp. 423-430 (2012)

[104] Razavi K., Razorea L.M., Kielmann T. (2014) Reducing VM Startup Time and Storage Costs by VM Image Content Consolidation. In: an Mey D. et al. (eds) Euro-Par 2013: Parallel Processing Workshops. Euro-Par 2013. Lecture Notes in Computer Science, vol 8374. Springer, Berlin, Heidelberg

[105] Marathe, Aniruddha & Harris, Rachel & K. Lowenthal, David & R. de Supinski, Bronis & Rountree, Barry & Schulz, Martin & Yuan, Xin. (2013). A comparative study of high-performance computing on the cloud. HPDC 2013 - Proceedings of the 22nd ACM International Symposium on High-Performance Parallel and Distributed Computing.

[106] MPI: A Message-Passing Interface Standard. 2019, August 30. Retrieved from https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[107] Open MPI: Open Source High Performance Computing. 2019, August 30. Retrieved from https://www.open-mpi.org/

[108] Boost Distributed Application Performance. 2019, August 30. Retrieved from https://software.intel.com/en-us/mpi-library

[109] MPICH. 2019, August 30. Retrieved from https://www.mpich.org/

[110] MPI Point to Point Communication. 2019, August 30. Retrieved from https://www.cs.mtsu.edu/ rbutler/courses/pp6330/www.navo.hpc.mil/pet/Video/Courses/MPI/Mod_2/Slides/more.html

[111] Midorikawa, Edson Toshimi, Helio Marci De Oliveira, and Jean Marcos Laine. "PEMPIs: a new methodology for modeling and prediction of MPI programs performance." International Journal of Parallel Programming 33.5 (2005): 499-527.

[112] Cornea, Bogdan Florin, and Julien Bourgeois. "Performance prediction of distributed applications using block benchmarking methods." 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing. IEEE, 2011.

[113] Casas, Marc, Rosa M. Badia, and Jesus Labarta. "Prediction of behavior of MPI applications." 2008 IEEE International Conference on Cluster Computing. IEEE, 2008.

[114] Heinrich, Franz, et al. "Predicting the Performance and the Power Consumption of MPI Applications With SimGrid." (2017).

[115] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel and J. J. Dongarra, "Performance analysis of MPI collective operations," 19th IEEE International Parallel and Distributed Processing Symposium, 2005, pp. 8 pp.-.

[116] Akihiro Nomura, Hiroya Matsuba and Yutaka Ishikawa, Network performance model for TCP/IP based cluster computing, 2007 IEEE International Conference on Cluster Computing, Austin, TX, 2007, pp. 194-203.

[117] L. Li, X. Zhang, J. Feng and X. Dong, mPlogP: A Parallel Computation Model for Heterogeneous Multi-core Computer, 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, VIC, 2010, pp. 679-684.

[118] Hoefler T., Mehlan T., Lumsdaine A., Rehm W. (2007) Netgauge: A Network Performance Measurement Framework. In: Perrott R., Chapman B.M., Subhlok J., de Mello R.F., Yang L.T. (eds) High Performance Computing and Communications. HPCC 2007. Lecture Notes in Computer Science, vol 4782. Springer, Berlin, Heidelberg

[119] Hockney, R.: The communication challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Comput. 20(3), 389398 (1994)

[120] Alexandrov, A., Ionescu, M.F., Schauser, K.E., Scheiman, C.: LogGP: Incorporating long messages into the LogP model. In: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, pp. 95105. ACM Press, New York (1995)

[121] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 112. ACM Press, New York (1993)

[122] Kielmann, Thilo, Henri E. Bal, and Kees Verstoep. "Fast measurement of LogP parameters for message passing platforms." International Parallel and Distributed Processing Symposium. Springer, Berlin, Heidelberg, 2000.

[123] PLogP Source Code. 2019, August 30. Retrieved from http://www.cs.vu.nl/pub/kielmann/

[124] Barchet-Estefanel L.A., Mounié G. (2004) Fast Tuning of Intra-cluster Collective Communications. In: Kranzlmüller D., Kacsuk P., Dongarra J. (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2004. Lecture Notes in Computer Science, vol 3241. Springer, Berlin, Heidelberg

[125] Luiz Angelo Steffenel, Grégory Mounié, A framework for adaptive collective communications for heterogeneous hierarchical computing systems, In Journal of Computer and System Sciences, Volume 74, Issue 6, 2008, Pages 1082-1093, ISSN 0022-0000,

[126] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke and W. Rehm, "A practical approach to the rating of barrier algorithms using the LogP model and Open MPI," 2005 International Conference on Parallel Processing Workshops (ICPPW'05), Oslo, Norway, 2005, pp. 562-569.

[127] I_MPI_ADJUST Family Environment Variables. 2019, March 7. Retrieved from https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family

[128] Ferretti M, Santangelo L. Cloud vs On-Premise HPC: a model for comprehensive cost assessment. In press, 2019.