Scheduling of the Applications in the Cloud

Master's Thesis

Natural Science Faculty of the University of Basel

Department of Mathematics and Computer Science

High Performance Computing

Examiner: Prof. Dr. Florina Ciorba

Supervisor: Dr. Ahmed Eleliemy

Supervisor: Dr. Ali Mohammed

Drilon Vukaj

drilon.vukaj@stud.unibas.ch

18-058-339

16.09.2021

Abstract

Cloud computing is a paradigm that sparked the interest of the High Performance Computing (HPC) community due to the cloud service providers offering of HPC infrastructure, which provides an alternative to on-premise clusters for executing HPC workloads. Cloud computing systems were traditionally suited only for loosely-coupled parallel/distributed applications. However, cloud service providers enhanced their capabilities and added low-latency interconnects to provide a better support also for tightly coupled HPC workloads. In this thesis, we investigate whether a Cloud-based cluster can provide a high performance as on-premise cluster for HPC scientific applications. HPC scientific applications are complex and they are mainly based on loops which can often be irregular. Loop scheduling is a key element for obtaining good performance. Therefore, we investigate the performance of various loop scheduling techniques at one-level (thread) and two-level (process + thread) scheduling on cloud and on-premise. Our work shows that several experiments conducted in the cloud are HPC-competitive and very comparable to the on-premise HPC, but cloud suffers from performance variability due to node allocation and sharing of the resources with other cloud users.

Acknowledgments

I would like to express my gratitude to Prof Dr. Florina Ciorba for giving me the opportunity to write the thesis in her research group. She was always supportive and steered me in the right direction. I would like to thank my supervisors, Dr. Ahmed Eleliemy and Dr. Ali Mohammed. They have given me constructive feedback and crucial insights throughout all my thesis. The knowledge I got from them is not bounded inside the thesis scope, but we discussed other topics related to their research and how to spot different problems.

Furthermore, I want to thank my parents Fadil Vukaj and Fatmire Vukaj, and my brother Ilir and his beautiful family (his wife Donjeta and daughter Leandra) for constantly encouraging me throughout all my studies. They always gave me unconditional love and support. This work would not have been possible without them.

Finally, I want to thank my girlfriend Brikena Celaj for her continuously support and helpful discussions and suggestions.

Contents

Chapter 1

Introduction

Scientific applications simulate the actions of real-world objects based on mathematical models and such applications involve immense computations and amounts of data. Some examples are weather prediction and computational fluid dynamics applications. To satisfy the demands of scientific applications, the execution power and memory of High-Performance Computing (HPC) systems is required [31].

HPC is based on parallel architecture and parallel computation by providing a high number of computing units and distributing the work among them. Therefore, HPC is becoming one of the main pillars of scientific research.

Commonly, scientific applications involve large loops which present a good potential for parallelism. However, the execution time of these loop iterations can vary due to conditional statements inside the loop, system variation, and different input data. Loop parallelization is done in such a way that loop iterations are assigned to the processing elements (PEs). One of the main challenges in parallelizing scientific applications is the load imbalance, which is manifested when the PEs have uneven execution finishing times. Such loops are called irregular loops. The scheduling of the loop iterations to the PEs determines the performance of such applications. Scheduling techniques are classified as static and dynamic. Dynamic load balancing is achieved via dynamic loop scheduling (DLS) techniques.

One of the main aspects that need to be considered when parallelizing a scientific application on such a setup is the load balancing aspect. If the scheduling of the

tasks to the PEs is not done correctly, the load imbalance may impact and degrade the performance of the application. The load imbalance can exist across the nodes at the process and within a node at the thread level. The application performance is driven by the slowest thread because the threads that finish their work still have to wait for the execution of the slowest one [42]. In this context, the scheduling aspect is very important to achieve good load balancing. Currently, scheduling process is the conventional way to address load balancing. Many research efforts have been made to address load imbalance, and there exist several scheduling techniques in the literature.

An HPC system is a collection of nodes connected via low latency interconnection networks to form a single cluster, where each node is a self-contained computer with its own OS, CPU, RAM and hard drive. They usually have a shared memory within a single node and a distributed memory across several nodes. High-performance applications are usually executed in parallel, and parallelism can come in different forms. HPC system parallelism can be exploited by different existing programming models. Two common programming models of parallel programming in HPC are: shared-memory and distributed-memory programming. In the shared memory programming model, all parts share a common address space and they can directly access one another's data. It corresponds to the multithreaded programming, where multiple threads are of control within a single process. An example of shared memory programming is Open Multiprocessing (OpenMP) [13]. On the other hand, multiprocess programming corresponds to distributed memory. Here, inter-process communication takes place where multiple processes exchange data through message passing. A prominent example here is Message Passing Interface (MPI) [30]. Since the nature of the HPC systems is hybrid, the most common approach in scientific applications is to parallelize them in the hybrid fashion using MPI+OpenMP at the process and thread-level [42].

When it comes to accessing the HPC capabilities to execute scientific applications, the users have two possible scenarios: one is to acquire and operate an on-premise HPC cluster; the other option is utilizing cloud computing resources. The second

approach is known as "HPC on cloud", which refers to the use of cloud resources to run HPC applications [43]. Cloud computing is an emerging computing technology that provides the renting of computing resources via the internet based on a pay-as-you-go approach. The resources are operated based on virtual machines (VMs) and ran on a multi-tenant mode on the physical hardware [24]. However, executing HPC workloads in the cloud imposes difficulties due to the different properties that they have when comparing them with traditional enterprise and web applications. They have different execution mechanisms, enterprise and web applications run in a 24x7 fashion while HPC applications are usually run as jobs in batches. Moreover, HPC applications require more computing power [43].

The cloud computing paradigm has sparked a lot of interest from the HPC community because of the claims on the cost-effectiveness and the flexibility of this approach. The advantages of the cloud are the a)elasticity, where the users can quickly provision and adjust resources on-demand at any moment where users pay in pay-as-you-go approach, b) virtualization, that enables flexibility, customization, and resource control [33].

One of the challenges in cloud computing is the scheduling of the VMs during the resource provisioning [38]. In the resource provisioning process, the user makes a request for the resources and the VMs are created and allocated to the users. Improper scheduling induces load imbalance which affects the performance of the scientific applications.

It is important to highlight the difference between the two types of load imbalances that we introduced here because they are used in two different contexts. First, we introduced load imbalance challenge in the context of scheduling of the tasks to PEs, while in cloud computing context, the load imbalance is caused by the scheduling of the VMs to cloud users during the resource provisioning. In this work, we are focused on the load imbalance challenge in the task scheduling context.

Many studies [24][29] [2] evaluate the performance of cloud systems for HPC applications by either using synthetic benchmarks or real scientific applications. These studies have consistently identified that cloud computing is not suitable for tightly

coupled HPC applications and that the lack of a low latency network is the main bottleneck of cloud systems. Moreover, it has been concluded that cloud suffers from the performance variability. They have also consistently shown that cloud computing is competitive for HPC applications that are not communication bound. With the rapid development of the cloud, it is not known if this outcome is still valid However, there is a recent research work [32] that contradicts previous works due to the improvements made by the cloud providers. It shows that cloud computing can also provide very good performance for memory and communication-intensive applications. Certain research [42] works also measure the performance of the scientific applications which are designed to run on HPC systems with different scheduling techniques. However, there are no evaluation studies that compare the performance of such applications with different DLS techniques on on-premise HPC, with their performance on the cloud resources.

Two approaches on-premise HPC and cloud computing are converging. Understanding the gap between HPC and cloud systems is a very important aspect that needs to be considered to further help the convergence. This work not only evaluates and compares the performance of the latest cloud computing resource improvements with on-premise HPC, but also evaluates how the applications which are designed for HPC systems with the tuning of different DLS techniques behave in both an on-premise HPC system and cloud computing resources. The DLS techniques designs are not specific to a programming model. They are implemented and can be applied in different levels. In this work they are applied at thread and process level to two scientific applications, Mandelbrot[39] and SPHYNX[20]. What makes this work unique is the filling of the gap between two approaches to access HPC capabilities, in such a way that we compare the performance trends of scientific applications using different DLS techniques during the execution running on both approaches.

Chapter 2

Background

2.1 Programming Frameworks

2.1.1 OpenMP

OpenMP Application Programming Interface (API) is a specification for parallel programming and supports shared memory multiprocessing programming in C, C++, and Fortran [13]. In OpenMP applications, the parallelization is done implicitly, and the runtime behavior is based on the compiler directives, library routines, and environment variables [13]. Hence, it requires the compiler to support this specification and a runtime library which provides an interface to the compiler. Many well-known compilers such as Clang and GNU Compiler Collection (GCC) [12] support the OpenMP specification.

OpenMP operates on the shared memory programming model where the execution is made within a single node. The implicit parallelization of the program is made via directives and *pragmas* which are specified in the part of the source code that will be executed in parallel. The compiler is responsible to parallelize the specified part based on the directives. OpenMP supports different constructs to parallelize a program. Some of them are parallel regions, work-sharing, variable scoping, critical regions, and synchronization. These constructs can be combined to achieve the goal of parallelization. One of the main features of OpenMP is the parallelizing of the

loops which require specifying only a line just before the for loop:

$$\#pragma\,omp\,for[clause[[,]clause]...]new-line$$
$$for-loops$$

However, the programmer is responsible to analyse the dependencies between the loop iterations in such a way that iterations are independent of each other. This assures the correct final result.

2.1.2 MPI

MPI is a message-passing library interface specification for distributed memory systems where the processes work in parallel and have their individual memory space [30]. It is not a library but a specification of what a message passing library should be. It defines the syntax and the semantics of library routines. MPI can be used to develop message-passing programs using C, C++, and Fortran [30].
The parallelization in the MPI is done explicitly. The programmer is responsible to organize the processes, data distribution, and interaction between the processes. There are two communication types used in MPI: point-to-point communication and collective communication. Both of them have different modes and corresponding library routines that allow programmers to implement them.

2.2 Loop Scheduling techniques

The most time-consuming parts during the computations in large programs are loops. However, loops represent a huge source of parallelism. Their parallelization can bring benefits to applications in terms of execution time. Scheduling task is a process in which the parallelization of the loops is materialized by decomposing them in iteration blocks and assign those blocks to the available PEs. As the loop iterations can have different execution times due to conditional statements inside them, different inputs, and system variations, this can lead to load imbalance which means that PEs don't

have even execution finishing time. The PEs that finish earlier than others, stay in idle mode. This represents a waste of the computation power potential. To achieve balanced load execution, scheduling techniques balance the load into the PEs, so in this case, the finishing time of each PEs iterations is similar and there is no wasted potential. However, scheduling also involves overhead. The overhead can degrade the performance. There is a trade-off between the scheduling overhead and the load balance and based on that, different scheduling techniques fit better on reducing the overall execution time. There are two main approaches of loop scheduling techniques: static and dynamic. In the following two subsections an overview of two approaches and different scheduling techniques that exist for them is given. The selection of the scheduling technique per application, per loop per system basis is a decision to be made, and some techniques require the chunk parameters. There is a novel approach AUTO4MP that addresses this process by an automated selection of the scheduling technique. In this approach, several automated selection techniques are implemented, and a brief overview of these techniques is given in the last subsection of this section.

2.2.1 Static techniques

In the static techniques, the iterations are divided into chunks of a fixed size, which are then assigned to PEs before the execution of the loop starts. As the scheduling decision is made before execution, there is minimal scheduling overhead. This is a very good approach when the loops are regular and iterations have the same execution time. However, these techniques can produce load imbalance (uneven finishing times of PEs) in cases when the loops are irregular or due to system variability. Static chunking (SC) is an example of static loop scheduling techniques, where PE has a chunk of tasks that is equal to the number of tasks N divided by the number of PEs (N/P). The fixed chunks sizes are given to PEs before the execution of the loop. In the trade-off between scheduling overhead and load balance, SC represents the extr<eme where it has minimum load balance with the minimum scheduling overhead.

2.2.2 Dynamic self-scheduling techniques

Dynamic techniques differ from static because the scheduling decisions are made not before the execution of the loop but dynamically during the application execution. The PEs that have finished their assigned iterations take over other iterations during the runtime. This can be done in two ways: centralized fashion, where a master PE assigns tasks to worker PEs; decentralized fashion, where each PEs assigns tasks via self-scheduling using a central pool. DLS techniques are also divided into adaptive and nonadaptive techniques. Adaptive DLS techniques adapt the scheduling decisions based on the information they obtain during the runtime. Some of adaptive techniques are adaptive weighted factoring (AWF) [18], its variants AWF-B, AWF-C, AWF-D, AWF-E [21] and adaptive factoring (AF) [16]. Non-adaptive DLS techniques make decisions of scheduling during the runtime based on pre-computed information prior to execution. This information does not change anymore during the runtime. Some of nonadaptive techniques are self-scheduling (SS) [46] , fixed-sized chunking (FSC) [37], guided self-scheduling (GSS) [44], trapezoid self-scheduling (TSS) [47], modified fixed-sized chunking (mFSC) [17], factoring (FAC) [35], weighted factoring (WF) [34] and random (RAN) [23] etc. Adaptive DLS techniques incur a higher scheduling overhead comparing to the non-adaptive ones. However, their design is focused on outperforming non-adaptive in highly irregular execution environments. Next in this subsection an overview of DLS techniques is give.

- **Self-scheduling (SS)** assigns a single iteration per PE request. The chunk size is equal to one.

  $$C_s = 1$$

  Regarding the trade-off between scheduling overhead and load balance, SS represents one of the extremes. It has the maximum load balance with the maximum scheduling overhead. Implementing SS in a decentralized model for getting loop iterations, gives the possibility to reduce the scheduling overhead.

- **Fixed size chunking (FSC)** - tries to reduce the scheduling overhead of the

SS technique by assigning loop iterations into the fixed size of chunks instead of a single one. The chunk size depends on the standard deviation of the iterations, sigma, and the scheduling overhead, h.

$$C_s = \left(\frac{\sqrt{2}Nh}{\sigma P \sqrt{\log P}} \right)^{2/3}$$

With the above formula, we try to find the optimal size of scheduling overhead and at the same time have a good load balance.

- **Guided Self-scheduling (GSS)** - assigns a decreasing chunk size to PE where this chunk size is the number of remaining loop iterations R_i, divided by the number of PEs P.

$$C_{s_i} = \left\lceil \frac{R_i}{P} \right\rceil$$

GSS tries to reduce the time of scheduling overhead by using less chunks and at the same time have a good load balance. It addresses the uneven starting time of PEs.

- **Trapezoid self-scheduling - (TSS)** assigns a decreased chunk size to PE similar to GSS, but in TSS the chunk size decreases linearly. Due to this linear calculation, the scheduling overhead is reduced because TSS is more efficient and simpler to implement. Trapezoid Self-Scheduling takes the first chunk f and last chunk l as inputs from the user and calculates the size as given below.

$$A = \left\lceil \frac{2N}{f+l} \right\rceil,$$

$$\delta = \frac{f-l}{A-1},$$

$$C_s(1) = f,$$

$$C_s(t) = C_s(t-1) - \delta$$

The number of chunks is represented with A and current scheduling operations

are represented with t.

- **Factoring (FAC)**- this technique is based on GSS FSC techniques. Different to this two methods, FAC assigns iterations into batches of P equal size chunks. It would be as GSS method if each batch has one chunk, or as FSC method if it has only one batch. To calculate the size of chunks, it uses a probabilistic analysis.

$$C_{s_j} = \left\lceil \frac{R_j}{x_j P} \right\rceil,$$

$$R_0 = N, R_{j+1} = R_j - PC_{s_j},$$

$$b_j = \frac{P\sigma}{2\sqrt{R_j}\mu},$$

$$x_0 = 1 + b_0^2 + b_0\sqrt{b_0^2 + 2},$$

$$x_j = 2 + b_j^2 + b_j\sqrt{b_j^2 + 4}, j > 0,$$

  where j represents the index of batch and R represents the remaining iterations. FAC is robust to variances of iteration execution time. Depending on σ, it is similar to SS if σ is hight or it is similar to static for low σ.

- **Weighted-Factoring (WF)** is similar to FAC, but to calculate the chunk size it considers the PE speed. It dynamically decreases the chunk size of iterations to PEs in proportion to their processing speed. Therefore, PEs have a weight w where after the calculation of the batch and chunk size, these chunks are multiplied by w.

$$C_{s_{ij}} = w_i \times C_{s\_factoring_j} and \sum_{i=1}^{P} w_i = P$$

  This equation represents the size of batch j for PE i.

- **Adaptive Weight Factoring (AWF)** - this technique works similarly to WF. However, AWF adjusts the weights after each time step to balance the load. The profiling and prior knowledge of system load are not required. This technique

is created for time-stepping applications. To find the weight, AWF uses the performance of all PEs in the previous step.

Adaptive Weighted Factoring Variants – AWF has different variants like AWF-b, AWF-c, AWF-D, AWF-E. The main goal of these variants is to deal with AWF limitations. While in AWF the adaption happens after each step, in the AWF variants, the adaption can happen also during the loop execution. These variants adjust the PE weights with a different frequency depending on the variant. Chunk size is calculated similarly to AWF with a modified version of the weighted average ratio.

$$\pi_i = \frac{\sum_{j=1}^{S_i} j \times t_{ij}}{\sum_{j=1}^{S_i} j \times n_{ij}}$$

- **Adaptive Factoring (AF)** - It is a generalized version of FAC and WF. Different from FAC where mean and standard variation is known and are the same in all PEs, in AF, both values are calculated during the runtime. Compared to WF, AF dynamically calculate the new chunk's size based on the recent performance. To calculate the size of the chunk, AF uses a probabilistic model. It has more overhead than AWF due to frequent time calculation on iteration level.

2.2.3 AUTO4MP

Selecting the scheduling techniques is a time-consuming process for the users given that there exist several ones, and the selection must be done carefully by considering each technique with different characteristics and different chunk parameters. The complexity is increased when the decision has to be made for each loop, application, and system. The OpenMP standard provides the auto parameter for the schedule clause or as a scheduling option, where the scheduling decision is delegated to the compiler/runtime implementation. As the implementations of auto in OpenMP standard runtime libraries usually map to a certain technique, AUTO4MP is a novel

approach that implements four novel algorithms to automatically select a scheduling technique among the existing ones by extending the implementation of auto technique in the LLVM OpenMP runtime library. The algorithms implemented in AUTO4MP are Auto Random selection, Auto Binary selection, Auto Exhaustive selection, and Auto Expert selection. The remaining part of this subsection gives an overview of these algorithms is given.

- **Auto Random selection** - Random selection method chooses the scheduling algorithm randomly based on a defined probability P_j without trying to find the highest performing algorithm beforehand. This probability, P_j, is defined by the load imbalance at a certain point of execution, and it is calculated with the formula $P_j = LIB/10$, where LIB is the percent load imbalance. At each time step, the Random Selection method generates a number in [0,1] interval. For all the cases where the P_j is smaller than the generated number, the algorithm is not changed, otherwise, the algorithm is changed and chosen from Auto4OMP portfolio. The disadvantage of this method is that it can have a low performance.

- **Auto Exhaustive selection** - Exhaustive Selection method chooses the scheduling algorithm with the highest performance for a loop. To find out this algorithm, this method tries all algorithms in the Auto4OMP portfolio by selecting one algorithm at each time-step and calculate its execution loop time. If there is a high imbalance loop execution with the chosen algorithm, the Exhaustive Selection method restarts the process of searching for a better algorithm. The disadvantage of this method is that its trials to find the scheduling algorithm are proportional to the size of the Auto4OMP portfolio.

- **Auto Expert selection** - Expert Selection method, similar to the Random Selection method, chooses the scheduling algorithm without doing any search beforehand. However, this method uses fuzzy logic and expert rules, to decide which scheduling algorithm to select. It also collects information of the loop execution time during the execution and benefits from them. The fuzzy logic

helps to reduce the complexity of all problems that have to do with uncertainties. Its approach is to express expert knowledge as rules which help it to make a decision based on the uncertain inputs. There are three steps during the process of selection when using fuzzy logic: i) fuzzification, ii) evaluation of expert rules iii) defuzzification.

2.3 Scheduling libraries

2.3.1 OpenMP level

At the OpenMP, the runtime library is responsible for the scheduling task. OpenMP allows us to specify the scheduling algorithm to be used by the compiler. This can be done either on source code in pragma, or it can be chosen during the runtime by specifying *runtime* in the pragma. For the latter, the name of the algorithm should be specified in the environment variable OMP_SCHEDULE. OpenMP supports three standard loop scheduling techniques [13]: static, dynamic, and guided. The chunk size can be specified in the pragma as well, so the scheduling technique will know how many iterations will be given to the PEs. But of course, this depends on the chosen technique. If for instance the static technique is chosen, then blocks of iterations of the specified size will be assigned to each PEs before the loop execution. In the dynamic technique case, the specified number of iterations are assigned to the idling PEs during the runtime. In addition to standard OpenMP scheduling techniques (static, dynamic and guided), the eLaPeSD [22] library supports four additional DLS techniques: FSC, TSS, FAC and RAN. The LB4OMP [36] is an extended LLVM OpenMP runtime library that is used in this work. Apart from the standard OpenMP scheduling techniques, it also supports other dynamic and adaptive loop scheduling techniques from the literature: TSS, FSC, FAC, mFAC (improved implementation of Factoring), FAC2 (a practical variant of factoring), WF2 (a practical variant of weighted factoring), TAP (tapering), mFSC (modified FSC), TFSS (Trapezoid factoring self-scheduling), FISS (Fixed increase self-scheduling (FISS), RAND. It

also supports the auto selection methods Auto Random, Auto Exhaustive, and Auto Expert.

2.3.2 MPI level

There exist tools that implement DLS techniques at the process level using MPI. One of them is DLB_tool [41]. It uses a master-worker execution model where there is a master process and the other processes are called workers. The MPI worker processes request work from the master process that executes the self-scheduling techniques. DLB_tool initially implemented nine loop scheduling techniques: static, mfSC, GSS, FAC, AWF-B, AWF-C, AWF-D, AWF-E, and AF. Afterward, it was extended into LB4MPI [41] to support four additional DLS techniques: SS, FSC, TSS, and WF.

2.4 HPC systems

Nowadays research studies are mainly conducted using computer modeling, simulation, and analysis. Usually, solving complex problems require computing capabilities beyond a single server. This is made possible by the HPC systems. HPC systems are usually supercomputers comprised of a large number of CPUs, GPUs, and a very fast interconnection network between the computing elements. Their interconnection is based on well-defined network topology. The most common architecture of HPC systems is cluster architecture. HPC clusters usually have a large number of nodes configured identically, and it looks like a single system. More than 86% of Top500 systems are based on the cluster architecture [25].

2.5 Cloud computing technology

Cloud computing technology is based on sharing the computer system resources and allows users to rent them on-demand. The name "cloud computing" refers to the fact that the resources and data centers are made available to users over the internet [24]. The goal is to offer users a simplified way to access the technologies. Cloud

computing systems are usually built from commodity (consumer-level) hardware [24]. Cloud services can be divided into three main layers [24]:

- **Infrastructure as a Service (IaaS)** - Users can rent servers, storage, networking, run different operating systems on those servers. It is a suitable solution for cases when the users want to control all the computing elements by themselves. Hence, it requires good technical skills to manage the services.

- **Platform as a Service (PaaS)** - A platform is provided to the users, which allows them to develop, run, and manage their applications. It keeps the underlying infrastructure away from the user.

- **Software as a Service (SaaS)** -Delivers the applications as a service where the users can only access the software via a web browser. The users are not concerned about the underlying infrastructure.

One of the key benefits of cloud computing technology is that it allows to easily set up the IT infrastructure by reducing time and effort and also avoiding all the complexities of owning and maintaining it. Other than that, the cloud typically uses a pay-as-you-go fashion where the users pay only what they use. Sometimes applications have higher peaks in usage at a particular hour, week, or month. The scaling up and down of the resources makes financial sense, as having dedicated hardware and software, in this case, might cause idling for much of the time. For this, cloud computing provides burst computing capability by enabling quick resource adjusting to meet the demands on certain hours [24].

The fundamental technology behind cloud computing is the virtualization software which separates physical devices into many virtual devices, and allows managing them to perform different computing tasks [24]. It increases the efficiency and utilization of the existing computer hardware.

There exists public and private cloud providers which differ in the way they operate. The public cloud refers to the model where IT services are offered and shared with different organizations across the network and are provided via the internet. In this

case, it is the cloud service provider's task to manage and maintain the compute resources shared among their tenants. On the other hand, the private cloud refers to the solution where the resources are dedicated to a single organization. The compute resources can be located on-premise or managed by a third party in another place. Giant data centers are managed by public cloud providers and are located in different places around the world. Hence, they are usually divided into different geographic regions like the EU, US West, etc. The regions are physically isolated from each other in terms of location, power, etc. Each region is then further subdivided into so-called availability zones. One zone does not correspond to a single data center but is rather backed by one or more of them.

2.5.1 HPC on cloud

HPC systems are usually operated by non-profit organizations such as universities or national laboratories. They are typically funded by government agencies and allocated to research communities. HPC systems before were limited to the capacity provided by the on-premise infrastructures of these organizations. Today, cloud computing gives HPC users an alternative to this if the on-premise infrastructure is not available. In contrast to on-premise HPC systems, cloud systems are built from profit organizations to meet the market demand. Motivated by the growing commercial interest in large-scale machine learning training, they are continuously upgraded [24]. The virtualization technology allows cloud systems to run applications into the same hardware, and with this, it aims to achieve the economies of scale [24].

A difference worth to be mentioned is also the way how they can be accessed. For the on-premise HPC systems, the users usually have to make a request to HPC providers to allocate computing resources for the proposed research work. Sometimes the HPC facilities do not scale in the same line with computational demands, and the access to them might be rejected. Moreover, when access is granted, if the resources are overloaded, the jobs are queued until there are free resources to proceed. On the other hand, the cloud has no queues due to a high number of available resources. Easy and quick set up of the cluster in the cloud also reduces time-to-solution. Hence,

considering cloud computing as an alternative to on-premise HPC systems in certain cases is worth it.

The viability of executing HPC applications on the cloud has started with the increasing popularity of cloud computing in the late 2000s [19] [14].

The biggest players in the public cloud world currently are Amazon[1], Google Cloud [3], Microsoft [2], and IBM [5], Oracle [8].

Chapter 3

Related Work

Load imbalance has been studied in many works in different contexts using very similar terminology. Two contexts that are close to our topic and terminology are the load imbalance induced upon the scheduling of the applications into tasks manifested in the process and thread level, and the scheduling of the resources in the cloud during the resource provisioning.

The effective scheduling of the resources in the cloud improves quality of services (QoS) parameters as resource utilization, reliability, execution cost, etc [38]. The goal of scheduling in this context is to specify the best resources demanded by the end-user for the execution of a task. Inefficient scheduling of the resources may lead to performance degradation or wastage of cloud resources due to having overutilized or underutilized resources respectively [38]. The works [38] [15] present a survey regarding the scheduling techniques in cloud computing. However, the focus of this thesis is not on the scheduling of the resources in the cloud during resource provisioning but rather on the task scheduling of the applications. Scientific applications mainly consist of large loops of different sizes and more specifically, this thesis is focused on DLS techniques used to schedule the loop iterations on the PEs. Since this thesis aims to evaluate the performance of different task scheduling techniques executed in the cloud and compare them with on-premise HPC systems, it is also important to see the results of the other works in terms of performance evaluation for both approaches. HPC applications have different properties comparing to the traditional enterprise and

web applications and hence executing them in the cloud imposes difficulties. They require high computing power capabilities not only in terms of CPU but also in terms of memory and network speed needed to support the execution [43]. The on-demand elasticity of computing resources provides a good advantage for cloud computing. Despite this fact, previous studies [43] [29] [33] [24] [] have evaluated the performance of applications on cloud resources, and they have consistently shown that running the scientific applications on the cloud has some performance-limiting factors such as the virtualization overhead and lack of low-latency networks. Most of the performance evaluations are done using Amazon Web Services (AWS). Being the first player in the market in making simplifications to use cloud resources for individuals and small organizations, and the fact the AWS provided free credits to the researchers to utilize the cloud infrastructure, made AWS be in the limelight of cloud service providers.

Early studies [28], [27],[24] evaluated AWS instances for HPC applications, and the main conclusion was that cloud at that time were not suitable to run tightly coupled HPC applications due to the virtualization overhead and poor network performance. Afterward, AWS did some improvements in cloud instances to overcome those constraints. The work [29] evaluated these new instances (CC1 and CC2, which currently are outdated) using 512 cores. They showed the processors have higher computational power in new instances and that the scalability of the communication-intensive codes is limited due to high start-up latencies and limited bandwidths which are imposed by virtualized access to network interface cards (NIC). The study [33] tries to answer the "who" (who should use the cloud for HPC) and "what" (for what they should choose cloud for HPC)questions by evaluating the performance of HPC applications in different on-premise HPC systems and public/private clouds. The outcome was that cloud systems are suitable for small and medium-scale organizations which can benefit from the pay-as-you-go model and that the cloud systems provide comparable performance with on-premise HPC when executing applications with less-intensive communication patterns. The [43] also confirms that the lack of low-latency networks represents the main bottleneck for the cloud.

Considering the fact that most of the studies had similar outcomes regarding the performance of the HPC applications on the cloud, cloud service providers made improvements as a response to these research outcomes. The main paper close to our work is [32] because it is one of the most recent ones that evaluate the HPC application performance in the most recent improved instances in the cloud. This work [32] shows that AWS has made significant advances in offering higher bandwidth and lower latency networks. They show that today, cloud computing is not only suitable for compute-intensive applications but also for memory and communication-intensive ones.

To run HPC applications efficiently in the cloud, we must understand the gap between on-premise HPC and cloud systems. Minimizing this gap on the performance aspect would help scientists to make a better decision between the two approaches. Here, the related work revealed the lack of performance evaluations of HPC applications using different scheduling techniques in on-premise HPC and cloud. This thesis aims to address this lack by evaluating HPC applications and benchmarks in cloud systems of different cloud service providers comparing with on-premise HPC in the context of task scheduling.

Chapter 4

Methods

This section presents the methods used to analyze the performance of the scientific applications using different scheduling techniques on the cloud and compares them to the on-premise HPC systems. There are variables such as the processor, memory, network, and application that affect the performance on different systems. Therefore, we also take a closer look at each component separately by measuring their performance in isolation. On this basis, we divide our experiments into two categories: i) benchmarks and ii) scientific applications. In the first category, we use different tools to measure the performance gap between the systems on each component. In the second category, we select two scientific applications as representatives, Mandelbrot [39] and SPHYNX [20] applications. The experiments are conducted on sciCORE, miniHPC, and Google Cloud. It is worth mentioning that we perform the same set of experiments for each category in all three systems. They are not similar at the hardware level, but they are similar at the software level. Hence, the purpose is not to compare the performance of different platforms, because they are different. Instead, we perform investigations to understand the performance trends on these three platforms.

4.1 Goal

The goal of this work was to analyze the performance of the HPC workloads on on-premise HPC and Cloud. To do so, we had to choose the systems, benchmarks, and scientific applications on which the performance using different scheduling techniques is analyzed. Comparing the performance of cloud and on-premise HPC systems has been subject to studies in previous works, however, analyzing the performance of the scientific applications on cloud resources and on-premise HPC systems in the context of different task scheduling techniques is the contribution of this work. A set of loop scheduling techniques are available in the literature and are implemented in the libraries ready to be used. In this work, apart from performing the benchmarks on different systems, we are focused to compare the performance of different scheduling techniques in different applications. The scheduling techniques are applied on the thread-level and combined process with thread-level (two-level scheduling). Regarding the scientific applications and comparing the performance of the applications using different scheduling techniques, we divide our experiments into two phases: experiments where the scheduling techniques are only applied in the thread-level (OpenMP); experiments where the scheduling techniques are applied in two-level, thread and process level (OpenMP + MPI). Hence, a good amount of work was focused on selecting scientific applications, carefully planning the design of experiments by selecting the systems and scheduling techniques to be applied. Moreover, preparing the environments for the executions, including the library installation and compilation, was an important piece of this work. An overview of the components used for our work can be seen in the figure 4-1.

4.2 Benchmarks

Comparing the performance of computer systems using benchmarks allows us to investigate and gain knowledge on different aspects of the systems. In this category of experiments, we investigate the difference in contributions of the processors in the

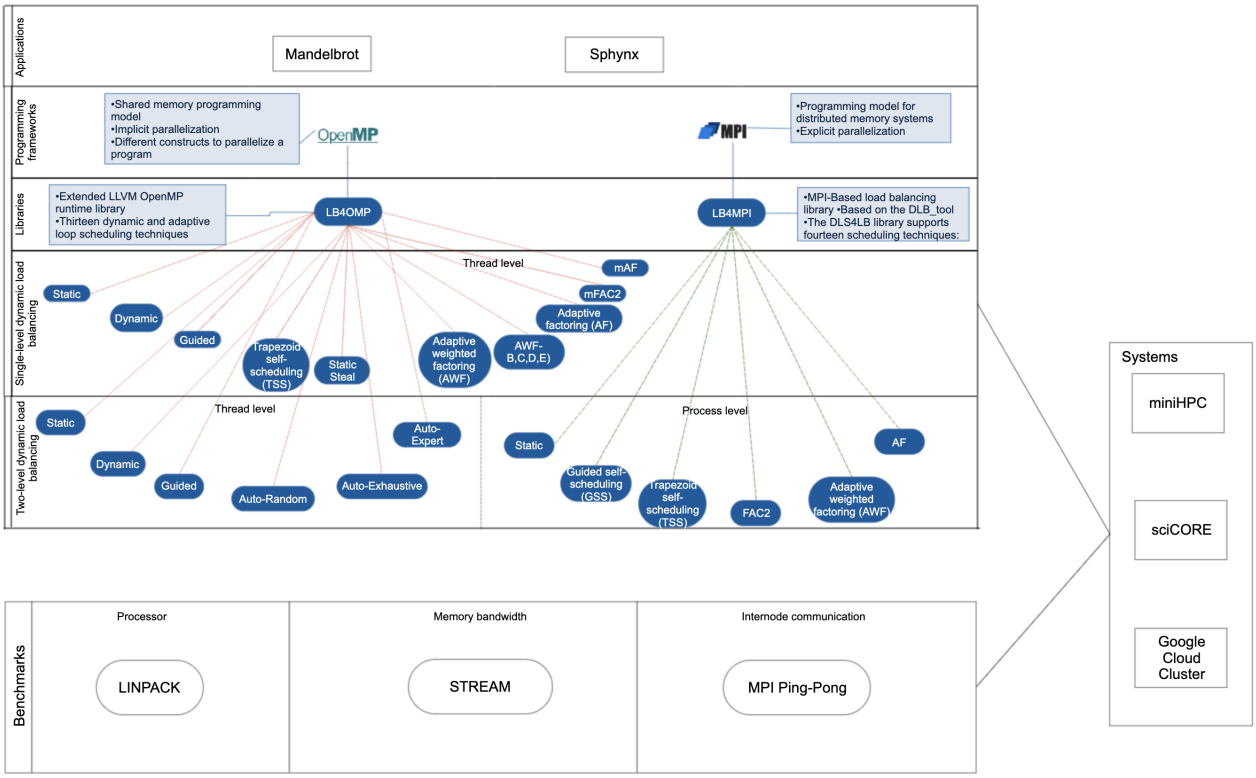

Figure 4-1: Methodology

selected HPC systems, then, we investigate the memory and the network performance.

4.2.1 Processor

The selected systems are built from multi-core processors, where each multi-core processor is considered as a node. Here, we selected the High Performance Linpack (HPL) [26] benchmark as a tool to perform stress testing on the processor. HPL benchmark solves a random linear system of equations and reports the floating-point operations the system can perform per second (FLOPS). It is often considered as a de-facto standard by the HPC community for processor benchmarking. It allows users to scale the problem size and optimize the software by specifying some parameters to achieve near-peak floating-point performance. However, the HPL usually requires several iterations of benchmark tuning of parameters to find the parameters to reach optimal performance. HPL is controlled by two critical parameters, P and Q, that describe the process distribution among the cores, where their multiplication gives

the total number of MPI ranks. Other parameters needed by HPL are N, which is the problem size to be solved, and NB, the block size for the data distribution. TOP500 also uses this tool to achieve the best performance for a given machine [11]. However, the reported number does not reflect the overall performance of the system, and the actual performance of an application might be lower than the performance reported by the HPL benchmark. Intel compiler was used on all three platforms with the Intel Math Kernel Library. For the Google Cloud, Intel OneAPI [6] compiler suite.

4.2.2 Memory Bandwidth

Another component that may affect the performance is the memory bandwidth of the system. To throw light on the memory bandwidth of our systems, we use the STREAM benchmark[40]. STREAM benchmark measures the memory bandwidth (in MB/s) by performing four-vector operations: copy, scale, sum, and triad. It has a general rule where each array must be at least four times the size of the sum of all the last-level caches used in the run.

4.2.3 Internode communication

Internode communication is an important aspect of the system that needs to be considered. To investigate it, we use the MPI Ping Pong[7] program which uses the MPI_Send and MPI_Recv to continually pass data to each other, and the bandwidth is calculated by measuring the time the data is transferred and taking into account the size of the data being transferred. Moreover, we use SkAMPI [45] benchmarking system, which allows us to perform collective communication operations as MPI_Alltoall to explore the latency aspect with a certain number of processes involved.

4.3 Scientific applications

Apart from the benchmarks to compare the HPC and cloud systems, we also selected two scientific applications which are used to measure their performance using different scheduling techniques. The selected applications are Mandelbrot written in C and Sphynx written in Fortran. Both applications are compiled using the Intel compiler.

4.3.1 Mandelbrot

When executed, Mandelbrot creates and displays fractal geometric images. Applying the equation $fc(z) = z4+c$ to a certain number of pixels in an iterative process which yields the image. Mandelbrot is considered a computationally-intensive application where the main work is done to compute the Mandelbrot set to generate the 2D image pixels. The calculation of every pixel is considered as a task, hence the application is parallelized in such a way that each of these tasks is performed in parallel. The execution time of these tasks has a high variation from each other. However, we use a modified version of Mandelbrot for both experiment types, thread-level, and two-level (process+level) scheduling. The modified version of Mandelbrot is that we executed it in time-steps and add two additional loops to explore loops with different characteristics. The original loop is with constant load imbalance, while for the other two, one is with increasing load imbalance and the other is with decreasing load imbalance. We refer to the three loops with *L0*, *L1* and *L2*

4.3.2 SPHYNX

SPHYNX is an accurate density-based smoother particle hydrodynamics (SPH) method for astrophysical applications [20]. It is developed and maintained at the University of Basel. Generally speaking, it is comprised of two heavy loops (we refer to them as *L0* and *L1*), which are hydrodynamics simulations. It is a time-stepping application, parallelized using MPI and OpenMP [20]. The performance of SPHYNX is studied for one simulation test-case, *Evrard collapse* which simulates the collapse of an unstable cloud of gas and the formation of the subsequent shock-wave [20]. Our experiments

are conducted on a domain size of one million particles to explore its performance.

4.4 Systems

Selecting the systems to compare the performance of scientific applications for on-premise and cloud approaches, was part of the investigation as well. For the cloud approach, we explored what do the different cloud service providers offer in terms of HPC such as AWS, Google Cloud, Azure, and Oracle. Moreover, we compiled a table that compares the prices of the instances offered by different cloud service providers, and the prices to gain access to the HPC centers. For the on-premise solutions, we explore the Pizz Daint instance from CSCS (Swiss National Computing Center) and sciCORE instances. The cost comparison can be seen in Appendix B. Our experiments are conducted on three different compute platforms: miniHPC, sciCORE and Google Cloud. Details of the selected systems are listed in the table below.

Table 4.1: Caption!

Platform	**Core**	**Frequency (GHz)**	**Processor**	**Memory (GB)**	**Network**	**L1**	**L2**	**L3**
miniHPC	10	2.4	Intel Xeon	64	Intel Omni-Path 100 Gbit/s	32 Kb	256 Kb	25 Mb
sciCORE	56	2.7	Intel Xeon Platinium 8280 CPU	386	Infiniband: 40GbE Ethernet: 1GbE	32 Kb	1 Mb	40 Mb
Google Cloud c2-standard-30	30 vCPUs	up to 3.8	Intel Scalable Processors (Cascade Lake)	120	32 Gbps	32 Kb	256 Kb	50 Mb

The reasons to select those systems are the easy availability and the diversity of the architectures. The miniHPC is available at the HPC group of the University

of Basel, which made it easy for us to gain access and conduct the experiments. sciCORE is also a center of scientific computing which provides infrastructure and services for high-performance computing. It is under the University of Basel umbrella and granting access was a straightforward process. We choose Google Cloud as the cloud service provider for the experiments of this work also due to the payment model that they apply, which was very convenient for the budget planning of this master thesis. The other service provider didn't provide the possibility to allocate some credits beforehand into our account and spend the credits on their services as we want. The only cost model offered by other cloud service providers was the monthly bill on the pay-as-you-go model and the upfront payment of specific instances which turned out to be a non-feasible solution for our case.

4.4.1 miniHPC

miniHPC is a small HPC cluster that has two types of nodes: Intel Xeon (22 nodes) and Intel Xeon Phi Knights Landing (KNL) (4 nodes). It has a peak performance of 28.9 double-precision TFLOP/s. We choose the Xeon partition due to the higher number of available nodes for our experiment. It is comprised of 22 computing nodes, 1 login node, and 1 storage node. The computing nodes are interconnected using a two-level fat-tree topology, with 100 Gbit/s speed. Slurm [9] is used as a cluster management tool and job scheduling system. The CPU speed in GHz is 2.4, it offers 20 threads and it has 64 Gb of RAM.

4.4.2 sciCORE

sciCORE is the scientific computing core facility of the University of Basel where the processor type is Intel Xeon Platinium 8280. It has 56 cores per node, two sockets with each 28 cores. The memory is 386 GB and the frequency is 2.7 GHz. The network bandwidth supported is 40Gbe of Infiniband. Also here Slurm is used as workload manager.

4.4.3 Google Cloud

Google Cloud offers a variety of computing instances offered for different purposes ranging from general-purpose, compute-optimized, and network-optimized. Appendix A shows the details of the type of instances offered by the Google Cloud. In particular, as the third system, we choose the Google Cloud commodity cluster built on top of the *c2-standard-30* compute-optimized instances (C2 family). The simplification of using cloud environments is a very important aspect identified in the past works. In the context of HPC cluster creation, Google Cloud offers a service called *Slurm cluster* which enabled us to quickly allocate a set of nodes within a specified placement group and availability zone. Google Cloud cluster runs on top of shared VMs, and it provides the Slurm workload manager out of the box.

One of the main advantages of this service is that the user doesn't need to have system expertise to build an HPC cluster, as everything can be configured by the Google Cloud console. During the cluster creation process, several decisions have to be made as choosing the instance type, selecting the availability zone, specifying the minimum number of nodes (the compute nodes which are all the time active), the maximum number of the nodes to be included in the cluster (maximum number of compute nodes that can be reached based on the nodes required in jobs). Apart from the compute nodes, the Slurm cluster also starts a login node and a controller node. The Slurm cluster in Google Cloud can be built in two ways: using Terraform [10] tool and by GUI in the Google Cloud console.

Google Cloud offers optimized images for HPC workloads that are pre-tuned for optimal performance [4] and are focused on tightly coupled MPI applications. The image is based on CentosOS 7 Virtual Machine. The pre-tuned HPC-optimized image comes with the hyper-threading disabled. The Slurm cluster is built on top of the HPC-optimized images, regardless of which instance type we choose. However, we mentioned that we selected *c2-standard-30* compute-optimized instance which has 30 virtual CPUs (vCPU). In Google Cloud, a vCPU is equivalent to a hyper-thread, and in the case of disabling the hyper-threading, we have 15 physical cores available in

one node.

4.5 Software stack

Upon the system selection, we came to a point to decide about the software stack in those three systems. We tried to make the software level similar in all systems. Some parts did not depend on us, for example, the operating system of the chosen systems. In the cloud, one can choose the operating system for the instances we want. However, the HPC-optimized images that we used for the experiments were built on top of the Centos 7 operating system. The same operating system was used in miniHPC and sciCORE as well.

The scheduling libraries that we selected were LB4OMP [36] and LB4MPI [41]. For their compilation, we used the Intel compiler which is compatible with the libraries. Installing and compiling them in the Google Cloud cluster was not a straightforward process. Everything had to be figured out on how to set up and prepare the environments for the executions. For the Google Cloud, we used a free version of the Intel compiler which is Intel OneAPI [6]. Here, we provide Appendix B with instructions how to prepare the Google Cloud cluster for the experiments, because several things had to be installed and also some workarounds had to be found. On the on-premise HPC, miniHPC and sciCORE, the compilers were already installed and ready to be used.

4.6 Design of experiments

The benchmarks and the applications were tested on three different systems, miniHCPC, sciCORE, and Google Cloud. It was important to choose a system size that gives us a fair comparison between cloud and on-premise HPC in the context of the performance of the applications with different scheduling techniques. Choosing the system size had an impact on choosing the Google Cloud instance as well. The experiments for one-level scheduling are executed within one node, using 20 threads for

miniHPC and sciCORE. Upon this decision, we selected the *c2-standard-30* compute-optimized instance for Google Cloud, which has 30 vCPUs, and in the disabled hyper-threading mode, it has 15 physical cores. On this basis, we used 15 threads on a node for Google Cloud. The reason for that is because we choose an instance that is comparable to what miniHPC and sciCORE could offer within a node in terms of PEs capacity. For the two-level scheduling, the system size was 10 nodes, 1 rank per node, and 20 threads per node in miniHPC and sciCORE. For Google Cloud, we used the same number of nodes and rank per node, but with 15 threads. All the experiments are performed with the hyper-threading disabled.
Both applications, Mandelbrot and SPHYNX are time-stepping applications. We run both of them with two hundred time-steps for one-level and two-level scheduling.

4.6.1 Thread level scheduling

For the thread-level scheduling, fourteen techniques are considered using the LB4OMP library. Techniques that require profiling information were not considered in this work as they are influenced by the provided profiling parameters such as standard deviation of task execution times sigma and the scheduling overhead h. The techniques that are considered in this section of experiments are STATIC and STATIC_STEAL for the scenarios where application tasks are statically and equally divided among the threads, SS GSS, TSS, STATIC_STEAL from non-adaptive DLS techniques and AWF, AWF_B, AWF_C, AWF_D, AWF_E, mAF for adaptive DLS techniques. In addition, we also evaluate three auto selection techniques implemented in LB4OMP library, the Auto Random, Auto Exhaustive, and Auto Expert.

4.6.2 Two-level scheduling (process+thread level)

In two-level scheduling, we consider five techniques at the process level and six techniques at the thread level yielding 5x6=30 different combinations per application. Also in this set of experiments, the techniques that require profiling information are not considered at any level. SS is considered only in the thread-level since using it in

Factors	**Values**			**Properties**
Applications	Mandelbrot			$N = 0.6x10^6$ tasks Time-step: 200
	Sphynx			$N = 1x10^6$ tasks Time-step: 200
	Linpack			Processor Benchmark
	STREAM			Memory Bandwidth Benchmark
	MPI PingPong			Network Bandwidth and Latency
	SKampI			Network Latency
	Scheduling level	**Library**	**Techniques**	
Single level dynamic load balancing	**Thread level**	OpenMP Standard	STATIC, SS, GSS	
		OpenMP non-standard	TSS	
		LB4OMP	static_steal, mFAC2, AWF, AWF-B, AWF-C, AWF-D, AWF-E, mAF, Auto Random, Auto Exhaustive, Auto Expert	
Two level dynamic load balancing	**Thread level**	OpenMP Standard	STATIC, SS, GSS	
		LB4OMP	Auto Random, Auto Exhaustive, Auto Expert	
	Process level	LB4MPI	STATIC, GSS, TSS, FAC, AWF	
Computing system	**Instance type**		**Price per hour**	
	miniHPC			
	scicore		0.007 CHF per core	
	Google Cloud		2.02 $ per node	

the process level causes a waste on thread-level parallelism by using only one thread when a single task is assigned to the requesting process. We omitted AF from the two-level scheduling experiments since we confirm the results shown in [42] that AF employed in process level generates large overhead and performs poorly for the experiments conducted on miniHPC.

The techniques considered in the process level are STATIC, GSS, TSS, FAC and AWF, and in the thread-level STATIC, SS, GSS, and three auto selection techniques Auto Random, Auto Exhaustive, and Auto Expert.

Chapter 5

Results

In this section, the results of our experiments are presented. Considering the fact that our systems are different in hardware level, we first investigate the performance of their different components using benchmarks. This is followed by the experiments conducted using scientific applications with different scheduling techniques.

5.1 Hardware and System view

5.1.1 Processor

The HPL benchmark required many iterations to scale and reach optimal performance. The figure 5-1 shows theoretical peak performance on the left and the HPL peak performance on the right for each platform for a single node setup using the maximum number of cores. Google Cloud reached around 65% of its theoretical peak, while for sciCORE the gap between theoretical and reached performance is much smaller. Also, the performance for sciCORE is much higher given the more processing power it has. For miniHPC, the reached performance is 677 GFlops/s, which is 85% of its theoretical peak performance. For the Google Cloud, we run a set of trials to find the best parameters, however, we are not sure if something else was running on the same machine and if that prevented us to achieve higher peak performance. We recall the fact that the reported number does not reflect the overall performance

of the system, since it may be the case that we didn't find the best parameters in our trials.

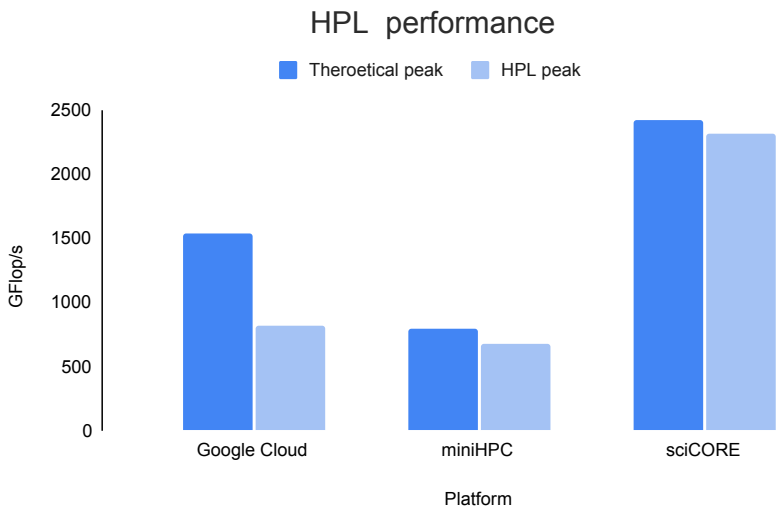

Figure 5-1: HPL performance

5.1.2 Memory bandwidth

STREAM benchmark offers us the possibility to measure the memory bandwidth with data sets larger than the available cache on the system [40]. We run the benchmark on a single node with the maximum available number of threads with hyper-threading disabled. The parameters used for the benchmark are 8 bytes per array element, array size of 120000000, offset 0. The memory per array was 915.5 MiB (= 0.9 GiB) and the total memory required = 2746.6 MiB (= 2.7 GiB). The best time is taken over the 10 runs to calculate the bandwidth. The results are shown in MB/s. The table 5.1 shows the results from the STREAM benchmark. sciCORE shows a higher memory bandwidth than the other two systems, while we can see that there is no significant difference between the miniHPC and Google Cloud when all the available cores are used and the data does not fit in their caches.

Table 5.1: STREAM benchmark results with array size 120000000, offset 0, memory per array 915.5 MiB

Platform	Threads	Copy	Scale	Add	Triad
Google Cloud	15	70427.8	70237.4	79457.4	79358.3
miniHPC	20	70219.6	70124.2	79025.5	79147.1
sciCORE	56	139584.8	139536.4	159990.4	160823.3

5.1.3 Inter-Node communication

We explore the inter-node communication in a multi-node setup by studying the network performance in two aspects, bandwidth, and latency. For latency, we employ two MPI Ping-Pong [7] and SKaMPI [45] benchmarks, while for bandwidth only MPI Ping-Pong. In figure 5-2 and 5-3, the experiments for MPI Ping-Pong bandwidth and latency are conducted using two nodes, one process per node. The results for MPI Ping-Pong bandwidth 5-2 shows us that Google Cloud reaches a peak bandwidth of about 17 Gbit/s for its c2-standard-30 instances, with a slight drop after package size exceeds 10^5 bytes. On the other hand, our results show that peak bandwidth for sciCORE is 22Gbit/s, while for miniHPC is 17Gbit/s and falls after the package size exceeds 10^5 bytes.

The figure 5-3, illustrates the latency aspect. Google Cloud c2-standard-30 instance obtains higher start-up latency (16 µs) than other systems, and when the package size exceeds 10^5, the latency is linearly increased. miniHPC and sciCORE start with lower latency with smaller package sizes, 3,76 µs and 1.50 µs respectively. After the package size exceeds 10^5 bytes, we can notice a slight increase of the latency in sciCORE and a significant increasing for miniHPC.

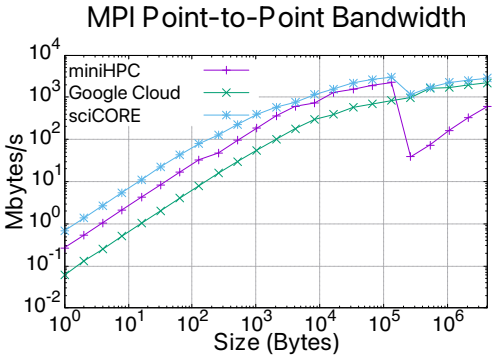

Figure 5-2: MPI Point-to-Point bandwidth with 2 nodes and 1 process per node

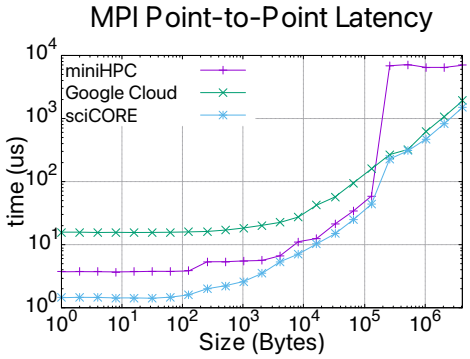

Figure 5-3: MPI Point-to-Point latency with 2 nodes and 1 process per node

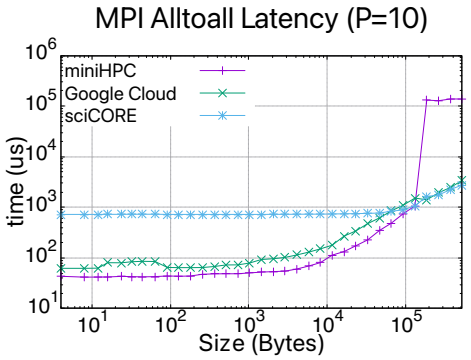

Figure 5-4: MPI Alltoall latency on 10 nodes, 1 process per node

Figure 5-4 shows the network latency when performing collective MPI_Alltoall operation with 10 nodes and 1 process per node. For smaller package sizes miniHPC and Google Cloud show no significant difference in latency, while for sciCORE latency is somewhat higher and continues at a constant rate. The latency starts to slightly increase after the package size reaches 10^4 bytes for sciCORE and Google Cloud, and their performance starts to overlap. While up to that point, miniHPC shows similar performance to Google Cloud, but the gap for miniHPC significantly increases after the package size exceeds 10^5 bytes.

5.2 Application view

In this section we present the results obtained from the experiments executed on thread-level scheduling using LB4OMP library, and two-level (process+thread) scheduling using LB4OMP and LB4MPI libraries for two scientific applications, Mandelbrot and SPHYNX. The same set of experiments is conducted on all three systems. We report the results based on the execution level for both applications.
Before starting the experiments, we observe the performance variability of the applications by running a set of experiments each hour in a range of ten hours on different days. Even for the static cases, in two-level scheduling, the results from Google Cloud show the coefficient of variation 17%, while for miniHPC 3% and sciCORE 2%. This indicates that the performance variability on Google Cloud is high due to the fact the cloud resources are allocated upon request when the jobs are submitted and deallocated after the jobs are finished, and the resources are shared with other cloud users. This may affect our results. However, our applications are time-stepping and we consider the average execution time among time-steps as well. For the thread-level scheduling executions, we repeat the experiments five times across all systems, while for two-level scheduling we take one sample on Google Cloud due to the budget limitations, and five repetitions for miniHPC and sciCORE.

5.2.1 Thread-level scheduling results

The goal of the performance analysis in this subsection was to examine which DLS technique can provide better performance for each application's loop using the LB4OMP library for the thread-level scheduling. The figures 5-5, 5-6, 5-7, shows the average parallel execution time for each technique for individual application executed on a certain system using one node and the maximum number of threads on Google Cloud (15) and miniHPC (20), while on sciCORE we limit the thread number to 20. The x axis in the figures 5-5, 5-6, 5-7, represents the loop scheduling technique, and the y axis the application execution time. As LB4OMP allows us to gather information for loops individually, we express them in different colors to explore which technique

performed better on which loop. The Best in the figure represents the combination of the best performing loop scheduling techniques for the corresponding loop, and it outperforms every single technique.

We observe from the figures 5-5, 5-6, 5-7, for the Mandelbrot executions that dynamic and non-adaptive scheduling technique SS and GSS provided a fairly good performance across all three systems, where GSS culminates with being part of the Best combination for the L0 and L2 loops on Google Cloud. TSS gives slightly worse performance than SS and GSS on all platforms. From the dynamic and adaptive scheduling techniques, AWF-C and AWF-E techniques consistently provide a good performance despite the AWF-C experiments on sciCORE. AWF, AWF-B, AWF-D and mAF give a lower performance on all three systems compared to other dynamic adaptive techniques. On Google Cloud, speaking of the parallel execution time aspect, auto techniques gives better performance than dynamic and adaptive techniques, as they are more closer to the Best combinations. This holds for miniHPC and sciCORE as well. Generally speaking, they provide very similar performance to the dynamic and non-adaptive techniques on Google Cloud and miniHPC. Moreover, on sciCORE, the performance gap between auto selection methods and other techniques is higher, where the best result is reached from the Auto Expert technique outperforming all other techniques for every loop and participating for each loop in the Best combination. This is consistent with miniHPC Best combination for L0 and L2 loops. On miniHPC, the best performance for the L1 comes from STATIC_STEAL, and this is consistent with the selection of the Best technique for L1 in Google Cloud as well. In general, all the DLS techniques provide a performance improvement on each system comparing with the STATIC case. Here, we can highlight that the performance trends of the DLS techniques are similar on the cloud to on-premise HPC systems when applied to the Mandelbrot scientific application.

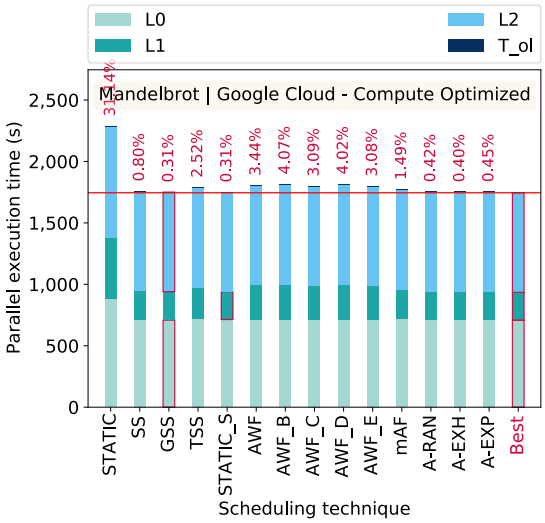

Figure 5-5: Mandelbrot performance in thread-level scheduling - Google Cloud

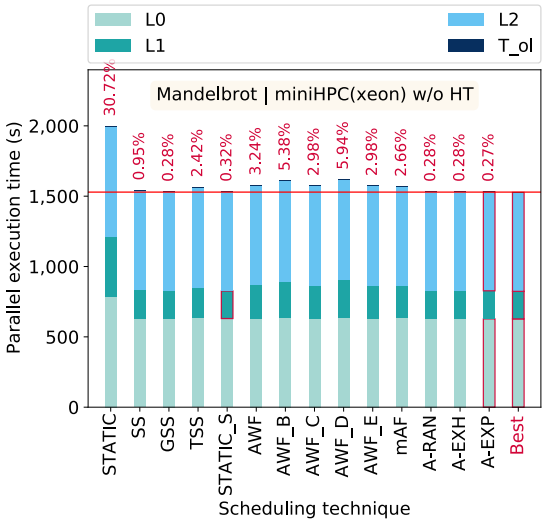

Figure 5-6: Mandelbrot performance in thread-level scheduling - miniHPC

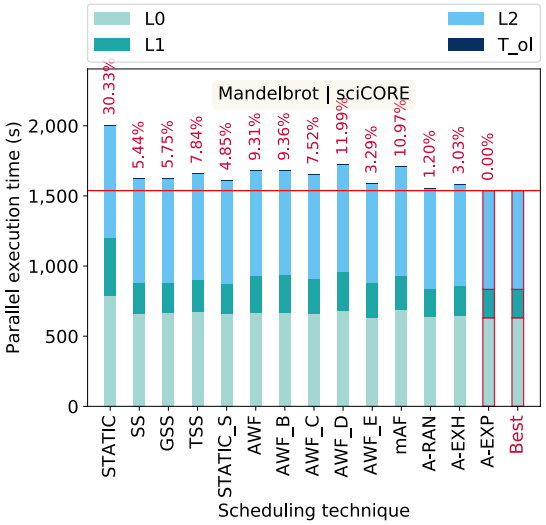

Figure 5-7: Mandelbrot performance in thread-level scheduling - sciCORE

From the figures 5-8, 5-9, 5-10, from SPHYNX application for dynamic and non-adaptive scheduling techniques, we observe that techniques mAF, GSS, and TSS consistently provided fairly high performance in almost all experiments across the systems. In Google Cloud, the best non-adaptive scheduling technique was GSS, with a difference of 7.56% worse than the Best combination in this set of experiments. TSS performed slightly worse on miniHPC and Google Cloud, but it provided the best performance among the non-adaptive scheduling techniques on sciCORE outperforming GSS with 3.6%. SS performance is penalized in all systems for SPHYNX

due to the high scheduling overhead. The adaptive DLS techniques add scheduling overhead due to the making of the scheduling decision during the runtime. However, they are capable to adapt to system variations and heterogeneity. We observe that adaptive scheduling techniques as AWF and all of its versions, together with mAF, consistently provide high performance among all systems. Adaptive DLS techniques' performance culminates with the mAF which is included in the Best combination for both SPHYNX loops on two systems, Google Cloud and miniHPC. The Best combination of techniques per loop for SPHYNX loops in sciCORE also comes from adaptive DLS techniques, and it is AWF_E technique.

Auto selection methods performance was consistent among on-premise systems, miniHPC and sciCORE providing a worse performance compared to other DLS techniques on those systems. However, they give us a different picture for the experiments conducted on Google Cloud when looking at the performance trend and compare auto selection methods with other DLS techniques. In Google Cloud, auto selection methods experiments provided a more comparable performance to the other techniques on the same system, with the Expert being best among them, and 7.27% worse than the Best combination.

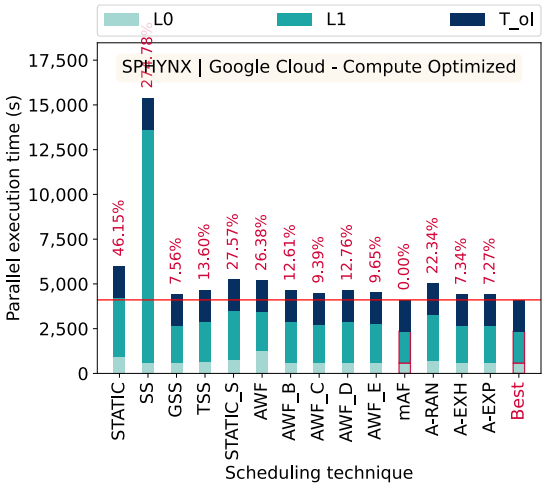

Figure 5-8: SPHYNX performance in thread-level scheduling - Google Cloud

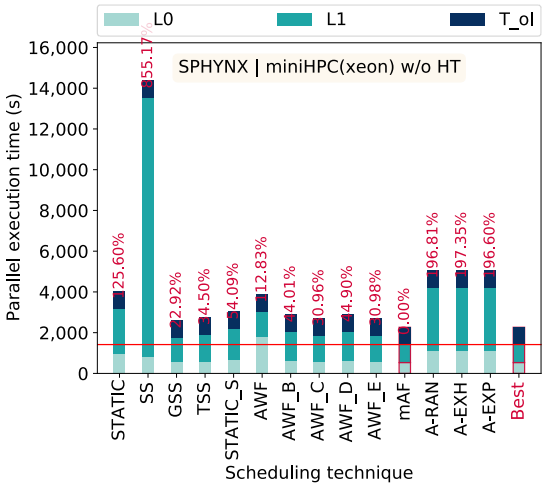

Figure 5-9: SPHYNX performance in thread-level scheduling - miniHPC

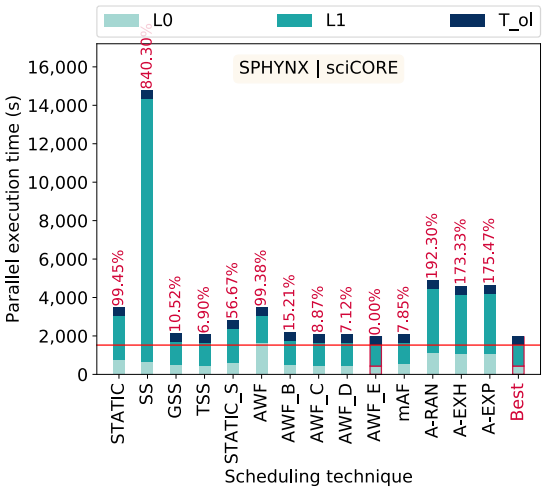

Figure 5-10: SPHYNX performance in thread-level scheduling - sciCORE

5.2.2 Two-level scheduling results

In examining the performance of DLS techniques on the process+thread level we execute parallel applications using hybrid parallel programming fashion with MPI+OpenMP. We choose a system size of 10 nodes, 1 process per node, and the maximum number of threads with hyperthreading disabled. Given the varying hardware characteristics among all systems, for Google Cloud, we use the maximum number of 15 cores available for the chosen c2-standard-30 instance, while for both on-premise HPC clusters we use 20 cores per node. The Google Cloud allowed us to place all 10 nodes

with HPC optimized images in the same placement group, which is critical to achieving low-latency network performance. The goal of the performance analysis in this subsection was to examine which combination of DLS techniques can provide better performance for each application, and see if the cloud can provide HPC competitive performance for scientific applications.
The figures, show the parallel execution time of SPHYNX for all combinations and highlights the best combination in terms of parallel execution time. The combinations in the figure are clustered process-wise where the techniques selected at the process level appear in the x-axis, and each bar represents the technique employed at the thread level. Also, these figures show in percentage how worse are other combinations performing when comparing to the best combination.

From the figures 5-11, 5-12, 5-13 we observe that using two-level scheduling for SPHYNX in Google Cloud didn't show any improvement compared to the case when not using any DLS technique in terms of application execution time, since the best performance is achieved when using the STATIC technique in both levels. However, the combinations of DLS techniques, for example, GSS at the process level with GSS, or with auto selection methods on the thread level, give a worse performance compared to the best combination for about less than 5.5% on Google Cloud and less than 4.26% on miniHPC. Also, employing TSS on the process level and maintaining STATIC on the thread level shows a fairly good performance on Google Cloud and miniHPC, with the performance percentage worse than the best performing combination 2.45% on Google Cloud and 3.87% on miniHPC. Combinations of the TSS technique in the process level with other techniques on the thread level (except SS in the thread level) show a comparable performance trend between Google Cloud and miniHPC. The best combination for SPHYNX in miniHPC in terms of application execution time turned to be the STATIC+Auto Expert. This combination has consistently shown high performance across the systems since on sciCORE resulted as the best combination as well, and on Google Cloud performed 3.3% worse than the best combination. However, other combinations with STATIC at the process level

and selected techniques on a thread level, except SS, give a very similar performance on miniHPC, with a difference of less than 1% with the best performing combination. Combining FAC in the process level with the chosen techniques in the thread-level gave a similar performance on Google Cloud and miniHPC, where the performance difference between these combinations is less than 2.5% except FAC+Auto Exhaustive on Google Cloud which gave slightly worse performance compared to other combinations. This is not entirely consistent with sciCORE results since the difference between different combinations with FAC at the process level oscillates on a higher range than other systems.

Employing AWF on the process level shows a similar performance trend on all three systems, keeping the difference between combinations under 3%, except the AWF+STATIC on Google Cloud that gave a slightly worse performance than other AWF combinations.

We observe the combinations where the auto selection methods are involved in the thread level, and it is consistently shown across the systems that the best performance provided by these techniques is when used in combination with STATIC technique at the process level culminating with Auto Expert as part of the Best combination in two systems, miniHPC, and sciCORE. Auto Expert combination with STATIC shows the best performance among auto selection methods used on the thread level on Google Cloud as well. We also observe that the combination of three auto selection methods with FAC and AWF at the process level, show similar performance across all three systems keeping the oscillations of the experiments under 2.5%, despite the combination of FAC+Auto Exhaustive on Google Cloud that shows relatively worse performance than other FAC and auto selection methods combinations. The combinations of GSS with auto selection methods results show that GSS+Auto Random slightly outperforms other GSS and auto selection methods combinations on Google Cloud and sciCORE. However, this is not the case for miniHPC where the difference among GSS and auto selection methods is around 0.2%.

We also take a look at the experiments from the time-stepping aspect. In the figures 5-18, 5-14, 5-22 we extract the performance results based on the average execution

time of the time-step executions. In the figures 5-19, 5-15, 5-23 the plot is based on the median values of time-step executions, while for the figures 5-20, 5-16, 5-24 the minimum of all time-step executions is taken. The first column of these plots represents the time-step execution time for STATIC+STATIC combination, the second is for the best DLS technique at the thread level, the third is the best at the process level, and the fourth is the best two-level combination. However, for the Google Cloud and sciCORE, when selecting best at each level and the best combination per time-step, we exclude the STATIC+STATIC from the calculation since the best results were given from this combination and our interest is to investigate the DLS techniques behavior. We observe that when considering the average, median and minimum value per time-step, the Auto Expert technique has consistently shown the best performance across all three systems, except in the case when the minimum value of time-steps is taken on Google Cloud which shows the Auto Random as the best technique on the thread level. This is in line with the results considered from the parallel execution time of all time-steps as well. Looking at the median value per time-step consistently shows the TSS technique as the best performing DLS on the process level. However, when considering the average and minimum value per time-step on Google Cloud, it is the FAC technique that gives better results. In one case, when looking at the minimum value per time-step on sciCORE shows the GSS as the best technique on the process level. The best combination from the time-stepping perspective shows a strong competition between STATIC+Auto Expert and TSS+STATIC, since from the average value per time-step perspective the STATIC+Auto Expert performed the best on Google Cloud and sciCORE, and TSS+STATIC on miniHPC, while from the median per time-step perspective, TSS+STATIC performs better on Google Cloud and miniHPC, and STATUC+Auto Expert on sciCORE. However, considering the results system-wise, TSS+STATIC was consistently the best combination among the average, median and minimum perspective on miniHPC, and STATIC+Auto Expert on sciCORE. Google Cloud gives us a different picture in this aspect since in all three perspectives (average, median, minimum) a different combination is shown as the best. From the average value, median, and minimum time-steps values, the best

combinations are STATIC+Auto Expert, TSS + STATIC, and FAC+STATIC respectively.

As SPHYNX is a communication-intensive application, we can see from the results that the cloud can occasionally provide HPC competitive performance for tightly coupled HPC workloads with different scheduling techniques, since due to the budget limit, we run one experiment for each technique combination on the cloud for two-level scheduling. However, it was also shown that the cloud has performance variability that goes up to 17%, and we may need more repetitions to confirm our results.

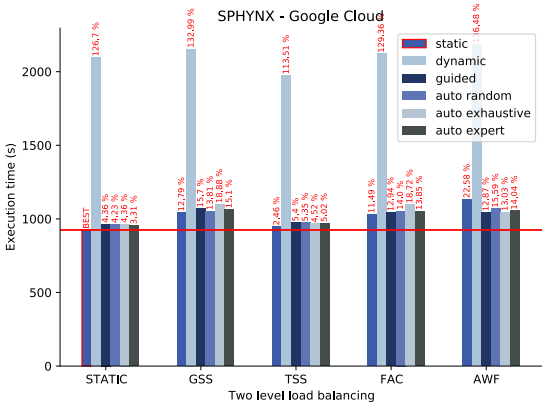

Figure 5-11: SPHYNX performance in two-level scheduling - Google Cloud

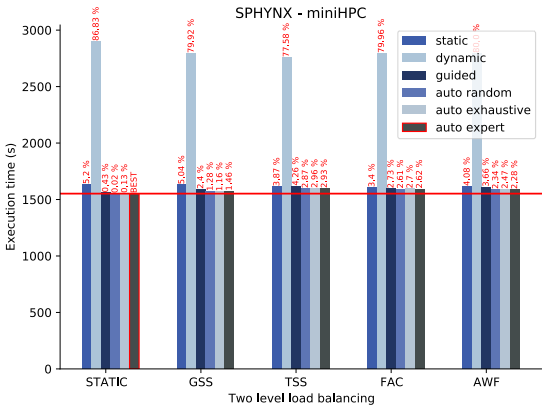

Figure 5-12: SPHYNX performance in two-level scheduling - miniHPC

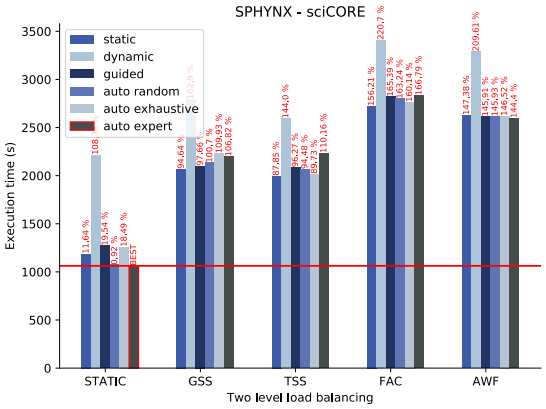

Figure 5-13: SPHYNX performance in two-level scheduling - sciCORE

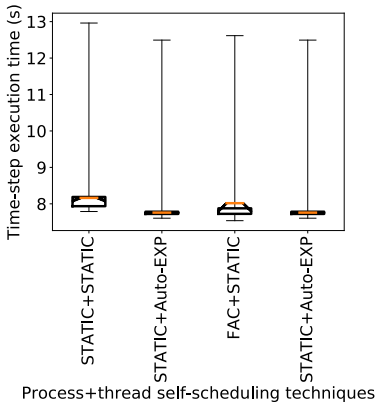

Figure 5-14: SPHYNX - miniHPC- min average per level

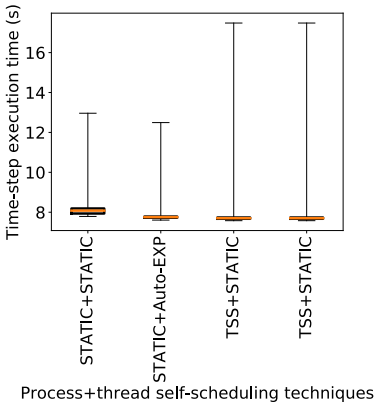

Figure 5-15: SPHYNX - miniHPC- min median per level

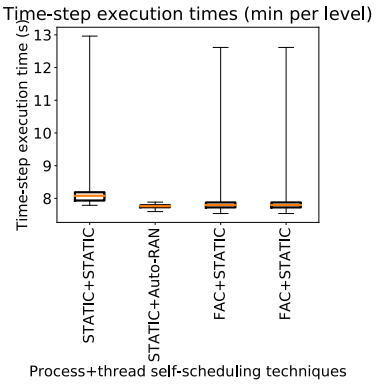

Figure 5-16: SPHYNX - miniHPC - min per level

Figure 5-17: miniHPC

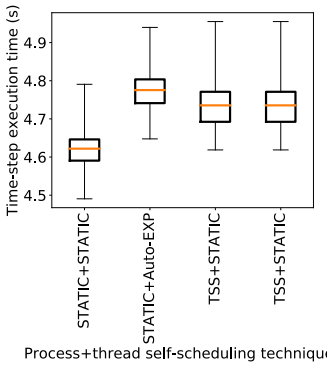

Figure 5-18: SPHYNX - Google Cloud - min average per level

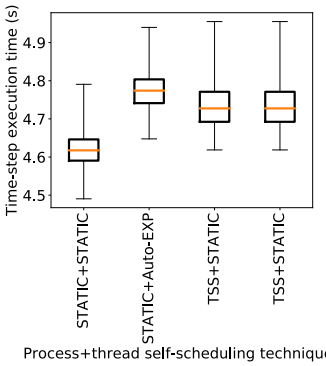

Figure 5-19: SPHYNX - Google Cloud - min median per level

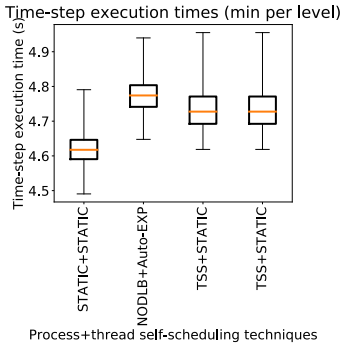

Figure 5-20: SPHYNX - Google Cloud - min per level

Figure 5-21: test

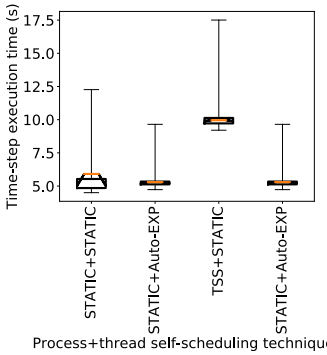

Figure 5-22: SPHYNX - sciCORE - min average per level

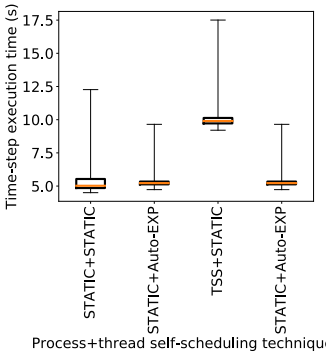

Figure 5-23: SPHYNX - sciCORE- min median per level

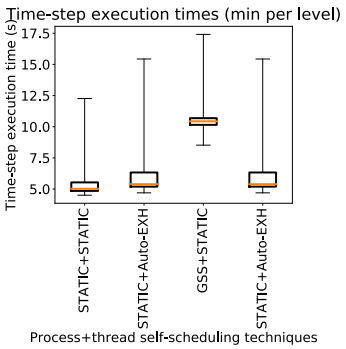

Figure 5-24: SPHYNX - sciCORE - min per level

Figure 5-25: test

For the Mandelbrot application, we merged an OpenMP time-stepping version

of it with the MPI version. We observed a significant difference in the execution times of our experiments with different scheduling techniques combinations between cloud cluster and miniHPC. Even for a small input size of the problem, the difference between execution times in cloud and miniHPC was for a scale factor of four to seven in favor of miniHPC for the same combination This observation is not consistent with other experiments results and with the assumption that cloud can provide HPC competitive performance for computationally-intensive applications. Nevertheless, this can be subject to a study in future work which investigates the reasons for such a behavior of Mandelbrot MPI and time-stepping application in the cloud.

Chapter 6

Conclusion & Future Work

In this work, the performance of on-premise HPC and cloud systems has been analyzed in the context of thread-level scheduling and two-level scheduling (process+thread). For this, we choose Google Cloud as representative for cloud platforms, and two on-premise HPC systems. We have built a cluster on Google Cloud and installed the necessary toolset which allowed us to perform our experiments on one-level and two-level scheduling. Several benchmarks for testing different hardware components have been performed. For the one-level and two-level scheduling experiments, two scientific applications, Mandelbrot and SPHYNX have been used in this work. For the scheduling process, we used different scheduling techniques implemented in LB4OMP [36] for the thread level and LB4MPI [41] for the process level. We have shown that performance trends of DLS techniques are similar on the cloud with on-premise HPC when comparing them to each other on the OpenMP level. Considering one-level scheduling, we identified that mAF, GSS, and AWF_E techniques have relatively good performance across all three systems for both applications. Also, auto selection methods show high performance when employed with the Mandelbrot application, especially Auto Expert which has shown highest performance across systems for Mandelbrot application. On the other hand, we identified AWF and AWF_B as techniques that provided lower performance in all platforms for both applications.

For the two-level scheduling, when looking from the overall application execution time aspect, the performance didn't improve on Google Cloud for DLS techniques,

with the STATIC in both levels being the best combination. However, for the two-level scheduling in the cloud, we execute one experiment for each combination, and more repetitions would be required. We identified GSS and TSS as the techniques that provided relatively good performance when employed at the process level despite some cases in sciCORE. GSS has also shown high performance when employed at the thread level. Employing SS for thread-level in two-level scheduling has shown worse performance due to the scheduling overhead. On the other hand, the auto selection methods have shown relatively good performance with the Auto Expert being the best among them in almost all cases.

While the observations from the time-stepping aspect have shown that the Auto Expert selection method provided high performance when employed in the thread level. For the process level, TSS has shown to be the best in most of the cases, while for the combination of process and thread level, TSS+STATIC has shown to be the best in most of the cases

We have also shown that also on MPI+OpenMP applications, even for the tightly coupled applications, the cloud can occasionally offer HPC-competitive performance and that the performance trends of DLS techniques can be similar on cloud and on-premise HPC. However, the cloud has shown to have a high variability for our experiments even on a small scale.

For future work, it would be interesting to perform more extensive experiments by increasing the number of scheduling techniques included in experiments. This can scale also platform-wise by performing the experiments on other cloud service providers as well. Moreover, the reasons for Mandelbrot MPI and time-stepping version behavior could be investigated in future work.

Appendices

Appendix A

	Optimization	Family	instance Type	CPU cores	vCPU	Default Threads per Core	Memory (Gib)	Network Bandwidth (Gbps)	Processor type	Frequency GHz	Price One-demand($ per hour)
AWS	Compute Optimized	C4	c4.2xlarge		8		15		Intel Xeon E5-2666 v3 Processor	2.9	$0.454
			c4.4xlarge		16		30		Intel Xeon E5-2666 v3 Processor	2.9	$0.909
			c4.8xlarge		36		60		Intel Xeon E5-2666 v3 Processor	2.9	$1.817
		C5n	c5n.2xlarge		8		21	25	Intel Xeon Platinum Processor	3	$0.492
			c5n.4xlarge		16		42	25	Intel Xeon Platinum Processor	3	$0.984
			c5n.9xlarge		36		96	50	Intel Xeon Platinum Processor	3	$2.214
		C5	c5.2xlarge		8		16	10	2nd gen Intel Xeon Scalable Processors (Cascade Lake)	3.4	$0.388
			c5.4xlarge		16		32	10	2nd gen Intel Xeon Scalable Processors (Cascade Lake)	3.4	$0.776
			c5.9xlarge		32		72	10	2nd gen Intel Xeon Scalable Processors (Cascade Lake)	3.4	$1.746
		C6g	c6g.2xlarge		8		16	10	Custom built AWS Graviton2 Processor	3.4	$0.3104
			c6g.4xlarge		16		32	10	Custom built AWS Graviton2 Processor	3.4	$0.6208
			c6g		32		64	10	Custom built AWS Graviton2 Processor	3.4	$1.2416
	Memory Optimized	R4	r4.2xlarge		8		122	10	Intel Xeon E5-2686 v4 Processor	2.3	$0.6402
			r4.4xlarge		16		244	10	Intel Xeon E5-2686 v4 Processor	2.3	$1.2804
			r4.8xlarge		32		488	10	Intel Xeon E5-2686 v4 Processor	2.3	$2.5608
		R5n	r5n.2xlarge		8		64	Up to 25	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3	$0.492
			r5n.4xlarge		16		128	Up to 25	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3	$0.984
			r5n.8xlarge		32		256	25	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3	$2.214
		R5b	r5b.2xlarge		8		64	Up to 10	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3.4	$0.388
			r5b.4xlarge		16		128	Up to 10	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3.4	$0.776
			r5b.8xlarge		32		256	10	Custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake)	3.4	$1.746
		R6g	r6g.xlarge		8		32	Up to 10	Custom built AWS Graviton2 Processor		$0.4864
			r6g.2xlarge		16		64	Up to 10	Custom built AWS Graviton2 Processor		$0.9728
			r6g.4xlarge		32		128	Up to 10	Custom built AWS Graviton2 Processor		$1.9456
Google cloud	Compute optimized	C2	c2-standard-8		8		32	16	Intel Scalable Processors (Cascade Lake)	3.1 up to 3.8	$0.54
			c2-standard-16		16		64	32	Intel Scalable Processors (Cascade Lake)	up to 3.8	$1.08
			c2-standard-30		30		120	32	Intel Scalable Processors (Cascade Lake)	up to 3.8	$2.02
	Memory optimized	M1	m1-ultramem-40		40		961	32	Intel Xeon Scalable Processor (Skylake)	2.0 up to 2.7	$6.3
Azure Microsoft	Compute optimized	H series	Standard_H8		8		56	40	Intel Xeon E5 2667 v3	3.2 to 3.3	$1.61
			Standard_H16		16		112	80	Intel Xeon E5 2667 v3	3.2 to 3.3	$1.901
	Memory optimized	Dv2-series 11-15	Standard_D13_v2		8		56	6			$1.056
			Standard_D14_v2		16		112	12		2.0 up to 2.7	$2.111
			Standard_D15_v2		20		140	25			no price
CSCS		Piz Daint	XC50 Compute Nodes (5704 Nodes)	12			64		Intel® Xeon® E5-2690 v3	2.6	Packages: 100 Node Hours (NH) - CHF 200 200 NH - CHF 550 1000 NH - CHF 2300 Additional price per NH: CHF 0.53 (Minimum request 10k node hours
			XC40 Compute Nodes (1813 Nodes)	2 x 18			64/128		Two Intel® Xeon® E5-2695 v4	2.1	
			Login Nodes	10			256		Intel® Xeon® CPU E5-2650 v3	2.3	
sciCORE				2x28			386		Intel Xeon Platinium 8280 CPU @ 2.7 GHz	2.7	CHF0.392
ORACLE		Oracle	VM.Optimized3.Flex	1/18/2021			1–256	Up to 50 Gbps	3rd Gen Intel Xeon Ice Lake	3.0 up to 3.6	$0.054/core/hr

Appendix B

Google Cloud SLURM cluster:

Prerequisites:

- Google Cloud account
- Billing account connected to your Google Cloud account

Steps:

1. Start the Google Cloud shell
2. Use *gcloud auth list* to see if cloud shell is configured properly. The expected output is:
   Credentialed accounts:
   - <myaccount>@<mydomain>.com (active)
3. Get the project if with *gcloud config list project* command
4. Set the project id with *gcloud config set project <PROJECT_ID>*. Expected output: Updated property [core/project].
5. Clone the Git repository that contains the Slurm for Google Cloud Platform Terraform files:
   git clone https://github.com/SchedMD/slurm-gcp.git
6. *cd slurm-gcp*
7. *cd tf/example/basic*

8. *cp basic.tfvars.example basic.tfvars*

9. Open *basic.tfvars* file

10. Modify Terraform file parameters to set up all the cluster configuration parameters as *cluster_name*, *project*, *zone*, *machine_type* etc.

11. Perform *terraform init*

12. Perform *terraform apply -var-file=basic.tfvars*
    This step may take a while to set up the cluster

13. After the cluster is set up, you can go to the *VM instances* in Google Cloud console. You should see two nodes started, controller and login node.

14. Login to the login node via SSH button in the table on *VM instances* page

    Up to this point we managed to set up the cluster and log in.

Intel OneAPI:

Google Cloud SLURM cluster comes with OpenMP compiler module installed. However, you can install the Intel OneAPI compiler. If you don't use an installation framework such as EasyBuild, you should install the Intel compiler in the home directory so the other compute nodes can have access to it.

LB4OMP:

Installing LB4OMP has its challenges as well. First you need to perform the following installations:

(a) sudo yum update

(b) sudo yum install gcc gcc-c++ make

(c) sudo yum install -y openssl-devel

(d) wget https://cmake.org/files/v3.13/cmake-3.13.3.tar.gz

(e) tar -zxvf cmake-3.13.3.tar.gz

(f) cd cmake-3.13.3

(g) ./bootstrap

(h) make

(i) sudo make install

Cmake needs to be installed in the home directory as well.
Google Cloud SLURM cluster runs on top of HPC optimized images which run on Centos 7 OS. This OS installs by default the gcc 4.8.5 which does not include shared_mutex file required by the LB4OMP compilation. The LB4OMP library needs newer gcc versions which contains the shared_mutex file. You can install newer gcc with these commands:

(a) yum install centos-release-scl

(b) yum install devtoolset-8

(c) scl enable devtoolset-8 – bash

In order the library to work with Google Cloud SLURM cluster, you need to comment the IF part for ICC version in the file
sudo vi /usr/local/cuda/include/crt/host_config.h
Every time you login, it is required to export the *CPLUS_INCLUDE_PATH* environment variable (or set it in the */.bash_profile* file)
export CPLUS_INCLUDE_PATH=/opt/rh/devtoolset-8/root/usr/include/c++/8
environment variable every time time you login to the cluster (or you can put it in the */.bash_profile* file). Another important export is the *I_MPI_PMI_LIBRARY* environment variable to point to the Slurm Process Management Interface (PMI) library:
export I_MPI_PMI_LIBRARY=/usr/local/lib/libpmi.so

Bibliography

[1] Amazon Web Services. https://aws.amazon.com/.

[2] Azure Microsoft. https://azure.microsoft.com/en-us/.

[3] Google Cloud. https://cloud.google.com/.

[4] Google Cloud HPC Optimized VM Image. https://cloud.google.com/blog/topics/hpc/introducing-hpc-vm-images.

[5] IBM. https://www.ibm.com/cloud.

[6] Intel OneAPI. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[7] Mpi ping pong to demonstrate cuda-aware mpi. https://github.com/olcf-tutorials/MPI_ping_pong.

[8] Oracle. https://www.oracle.com/cloud/.

[9] SLURM Workload Manager. https://slurm.schedmd.com/quickstart.html.

[10] Terraform. https://www.terraform.io/.

[11] Top 500 frequently asked questions. https://www.top500.org/resources/frequently-asked-questions/.

[12] *Using the GNU compiler collection*. Free Software Foundation, 4(02), 2003.

[13] *OpenMP Application Programming Interface*. 5 edition, 2018.

[14] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[15] AR. Arunarani, D. Manjula, and Vijayan Sugumaran. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems*, 91:407–415, 2019.

[16] I. Banicescu and Z. Liu. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes. In *Proceedings of the High Performance Computing Symposium*, pages 122–129, 2000.

[17] Ioana Banicescu, Florina M. Ciorba, and Srishti Srivastava. *Scalable Computing: Theory and Practice*, chapter Performance Optimization of Scientific Applications using an Autonomic Computing Approach, pages 437–466. Number Chapter 22. John WileySons, Inc., 2013.

[18] Ioana Banicescu, Vijay Velusamy, and Johnny Devaprasad. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing*, 6:215–226, 07 2003.

[19] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[20] R. M. Cabezón, D. García-Senz, and J. Figueira. SPHYNX: an accurate density-based SPH method for astrophysical applications. , 606:A78, October 2017.

[21] Ricolindo Cariño and Ioana Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. *The Journal of Supercomputing*, 44:41–63, 04 2008.

[22] F. Ciorba, Christian Iwainsky, and Patrick Buder. Openmp loop scheduling revisited: Making a case for more schedules. *ArXiv*, abs/1809.03188, 2018.

[23] Florina M. Ciorba, Christian Iwainsky, and Patrick Buder. Openmp loop scheduling revisited: Making a case for more schedules, 2018.

[24] Susan Coghlan and Katherine Yelick. The magellan final report on cloud computing. 12 2011.

[25] HPC-AI Advisory Council. Introduction to high-performance computing. https://www.hpcadvisorycouncil.com/pdf/Intro_to_HPC.pdf.

[26] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15:803–820, 08 2003.

[27] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In *CloudComp*, 2009.

[28] C. Evangelinos and C. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere. 2008.

[29] Roberto R. Expósito, G. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance analysis of hpc applications in the cloud. *Future Gener. Comput. Syst.*, 29:218–229, 2013.

[30] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[31] Gropp, W. Lusk, E., and Skjellum A. *Using MPI: portable parallel programming with the message-passing interface*. MIT, 1999.

[32] Giulia Guidi, Marquita Ellis, Aydin Buluç, Katherine Yelick, and David Culler. 10 years later: Cloud computing is closing the performance gap, 02 2021.

[33] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, and C. H. Suen. Evaluating and improving the performance and scheduling of hpc applications in cloud. *IEEE Transactions on Cloud Computing*, 4(3):307–321, 2016.

[34] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, page 318–328, New York, NY, USA, 1996. Association for Computing Machinery.

[35] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, August 1992.

[36] Jonas Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications, 06 2021.

[37] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.

[38] Mohit Kumar, S.C. Sharma, Anubhav Goel, and S.P. Singh. A comprehensive survey for scheduling techniques in cloud computing. *Journal of Network and Computer Applications*, 143:1–33, 2019.

[39] Benoit B. Mandelbrot. *Fractal aspects of the iteration of $z \to \lambda z(1\text{-}z)$ for complex λ and z*, pages 37–51. Springer New York, New York, NY, 2004.

[40] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[41] A. Mohammed and F. M. Ciorba. Research Report - University of Basel, Switzerland. https://drive.switch.ch/index.php/s/aanqAdp3X2Fxsoe, 2018.

[42] Ali Mohammed, Aurélien Cavelan, Florina Ciorba, Ruben Cabezon, and Ioana Banicescu. *Two-level Dynamic Load Balancing for High Performance Scientific Applications*, pages 69–80. 01 2020.

[43] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato Luiz de Freitas Cunha, and Rajkumar Buyya. HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Comput. Surv.*, 51(1):8:1–8:29, 2018.

[44] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[45] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Muller. Skampi: A detailed, accurate mpi benchmark. *Lecture Notes in Computer Science*, 1497, 10 1998.

[46] Peiyi Tang and Pen Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the International Conference on Parallel Processing*, Proceedings of the International Conference on Parallel Processing, pages 528–535. IEEE, December 1986.

[47] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.