AN EFFICIENT PARTICLE PHYSICS DATA ANALYSIS FRAMEWORK FOR HOMOGENEOUS AND HETEROGENEOUS PLATFORMS

André Pereira
ampereira@di.uminho.pt

A pre-thesis submitted for the degree of
Doctor of Philosophy in Computer Science

July 2014

Contents

Abstract

Most event data analysis tasks in the ATLAS Experiment require intensive data accessing and processing, where the computing performance of the applications constrain both the analysis throughput and the accuracy of the studied physics properties. This work focus on the compute bound issues at the last stage of ATLAS detector data analysis.

The goal for this work is to provide an efficient unified particle physics framework to aid the development of data analysis applications, within the LIP research group. It will replace the LipMiniAnalysis skeleton library, developed by LIP, and offer a more robust tool that efficiently uses the available computational resources of multi-CPU platforms coupled with hardware accelerators, while abstracting the intrinsic complexities of theses systems to the user. It redesigns the current programming model of data analysis, by providing efficient data structures, parallelization mechanisms, and managing all data analysis common functionalities, freeing the user to only code the specifics of each analysis.

Preliminary work identified the LipMiniAnalysis skeleton library inefficiencies, and a design of the new particle physics framework was proposed. It overcomes the data structure inefficiencies of LipMiniAnalysis and allows for a kernel-like programming model of data analysis, while automatically handling the data I/O, event data structure creation, and efficient load balancing among multiple CPU cores and hardware accelerators, such as GPUs and the Intel Xeon Phi. Most computing features of the framework were already detailed in the design specification.

Several prototypes of some of the framework core features were implemented and tested in the current LipMiniAnalysis skeleton library. This document presents a new data structure capable of holding multiple events in memory, an automatic parallelization of the even processing in a multi-CPU computing node, and a dynamic interface generator that provides a *plug-and-play* integration of the framework with legacy data analysis code.

Chapter 1

Introduction

Today's computing platforms are becoming increasingly complex with multiple interconnected computing nodes, each with multiple multicore CPU chips, and sometimes coupled with hardware accelerators. While the application performance is an important issue to tackle, the efficient usage of the resources of these systems is a crucial subject that needs to be addressed. Guaranteeing that the available computational resources are being adequately used by an application may require deep knowledge of the underlying architecture details of both CPUs and hardware accelerators, as well as extensive tuning of each individual application. It is crucial to comprehend the key issues that currently have more impact on computing performance and efficiency, namely the relationship between the cost of numerical computation, memory access, and data communication among available computing units. The architecture design of many-core hardware accelerators is significantly different among devices, with no standard yet defined. The programmer must know the architectural details of each hardware accelerator and their interconnection topology to the CPU to produce efficient code.

From the hardware point of view, efficiency may have a different meaning: it can be considered as the ratio between power usage and computational throughput. This is a subject of extensive research in a field also known as "Green Computing", where the goal is to reduce power consumption of the hardware while minimising the performance degradation. This is important for both mobile computing and to reduce the cost of maintaining huge computing clusters and data centres.

Computing clusters are the most popular High Performance Computing (HPC) platforms, constituted of many different computing nodes, interconnected by specialised communication channels in a distributed memory environment. The computing nodes may be characterised as homogeneous or heterogeneous platforms, where the former has one or more CPUs in a shared memory environment, and the latter has hardware accelerators coupled to the CPUs by a PCI-Express interface, in a distributed memory environment. This implies that the data is always visible to the CPUs, but must be explicitly transferred to the accelerator devices.

A proper data management is relevant to ensure the efficiency of an application.

Code parallelism is a must to take advantage of the multiple cores in both the CPUs and the hardware accelerators, adapted to the different memory and programming paradigms. Data races, resource contention and, when considering heterogeneous platforms, explicit memory transfers are complex challenges for the programmer. Also, each accelerator manufacturer uses their own frameworks and compilers to program their devices. Non-computer scientists opt to invest most of the research time on their field of science and have little time to improve their programming skills. These factors reinforced the collaboration of multidisciplinary teams of scientists from various fields with computer scientists to develop high performing, efficient, and robust applications.

1.1 Context

The European Organization for Nuclear Research [1] (CERN, acronym for *Conseil Européen pour la Recherche Nucléaire*) is a consortium of 21 European countries and more than 30 "observer" countries, with the purpose of operating the largest particle physics laboratory in the world. Founded in 1954, CERN is located in the border between France and Switzerland, and employs thousands of scientists and engineers representing 608 universities and research groups of 113 different nationalities.

CERN research focus on the basic constituents of matter to understand the fundamental structure of the universe, which started by studying the atomic nucleus but quickly progressed into high energy physics (HEP), namely on the interactions between particles. The instrumentation used in nuclear and particle physics research is essentially formed by particle accelerators and detectors, alongside with the facilities necessary for delivering the protons to the accelerators. The Large Hadron Collider (LHC) particle accelerator (later presented) speeds up groups of particles close to the speed of light, in opposite directions, inducing a controlled collision at the detectors core (the collision of two particles is referred as an "event"). The detectors record various characteristics of the resultant particles of each collision, such as energy and momentum, which originate from complex decay processes of the original particles. The purpose of these experiments is to test models and predictions in High Energy Physics (HEP), such as the Standard Model, by confirming or discovering new particles and interactions.

CERN started with a small low energy particle accelerator, the Proton Synchrotron [2] inaugurated in 1959, but soon its equipment was iteratively upgraded and expanded. The current facilities are constituted by the older accelerators (some already decommissioned) and particle detectors, as well as the newer Large Hadron Collider (LHC) [3] high energy particle accelerator, located 100 meter underground and with a 27 km circumference length. There are currently seven experiments running on the LHC: CMS [4], ATLAS [5], LHCb [6], MoEDAL [7], TOTEM [8], LHC-forward [9] and ALICE [10]. Each of these experiments have their own detector on the LHC and conduct HEP analysis, using distinct technologies and research

approaches. One of the most relevant researches being conducted at CERN is the validation of the Standard Model and discovery of the Higgs boson theory. The ATLAS experiment, a key project at CERN, aims to study the properties of the recently discovered Higgs boson [11], the search for new particles predicted by models of physics beyond the Standard Model like Susy, searches for new heavy gauge bosons and precision measurements where the top quark is of utmost importance. During the next year the LHC will be upgraded to increase its luminosity, e.g., the amount of energy of the accelerated particle beams.

Approximately 600 millions of collisions occur every second at the LHC. Particles produced in head-on proton collisions interact with the detectors of the ATLAS experiment, generating massive amounts of raw data as electric signals. It is estimated that all the detectors combined produce 25 petabytes of data per year [12, 13]. CERN does not have the financial resources to afford the computational power necessary to process all data from the detectors, which motivated the creation of the Worldwide LHC Computing Grid [14], a distributed computing infrastructure that uses the resources of the scientific community for data processing. The grid is organized in a hierarchy divided in 4 tiers. Each tier is made by one or more computing centres and has a set of specific tasks and services to perform, such as store, filter, refine and analyse all the data gathered at the LHC.

The Tier-0 is the data centre located at CERN. It provides 20% of the total grid computing capacity, and its goal is to store and reconstruct the raw data gathered at the detectors in the LHC, converting it into meaningful information, to be used by the remaining tiers. The data is received on a format designed for this reconstruction, with information about the event, detector and software diagnostics. The output of the reconstruction has two formats, the Event Summary Data (ESD) and the Analysis Object Data (AOD), each with different purposes, containing information of the reconstructed objects and calibration parameters, which can be used for early analysis. This tier distributes the raw data and the reconstructed output by the 11 Tier-1 computational centres, spread among the different member countries of CERN.

Tier-1 computational centres are responsible for storing a portion of the raw and reconstructed data and provide support to the grid. In this tier, the reconstructed data suffers more processing and refinement, to filter the relevant information and reduce its size, which is now in Derived Physics Data (DPD) format, to be then transferred to the Tier-2 computational centres. The size of the data for an event is reduced from 3 MB (raw) to 10 kB (DPD). This tier also stores the output of the simulations performed at Tier-2. The Tier-0 centre is connected to the 11 Tier-1 centres by high bandwidth optical fiber links, which form the LHC Optical Private Network.

There are roughly 140 Tier-2 computational centres spread around the world. Their main purpose is to perform both Monte-Carlo simulations and a portion of the events reconstructions, with the data received from the Tier-1 centres. The Tier-3 centres range from university clusters to desktop computers, and they are responsible for most events reconstruction and final data analysis. In the CERN

terminology, an application that is designed to process a given amount of data to extract relevant physics information about events, which may support a specific HEP theory, is called an analysis.

The Laboratório de Instrumentação e Física Experimental de Partículas (LIP) [15] is a portuguese scientific and technical association for research on experimental high energy physics and associated instrumentation. LIP has a strong collaboration with CERN as it was the first scientific organisation from Portugal that joined CERN, in 1986. It has laboratories in Lisbon, Coimbra and Minho and 170 people employed. LIP researchers have produced several applications for testing at ATLAS several HEP theoretical models that use Tier-2 and Tier-3 computational resources for data analysis. Most of the analysis applications use in-house developed skeleton libraries, such as the LipCbrAnalysis and LipMiniAnalysis.

There is the need to process more data, more accurately, in less time, which often leads to investments on larger computing clusters to improve the quality of the research results. However, most scientific code was not designed and/or developed for an efficient use of the available computational resources of modern platforms. If these applications were adequately designed (or tuned), the event analysis throughput could be massively increased. An efficient parallel application can significantly improve its performance at a much lower cost [16].

1.2 Problem and Motivation

With an increase in particle collisions and data being produced by the detectors at the LHC, research groups will need a larger budget to acquire and maintain the required computational resources to keep up with the analysis. Moreover, research groups working on the same experiment enforce positive competition to find and publish relevant results. The amount and quality of event processing has a direct impact on the research, meaning that groups with access to the most efficient computational resources become ahead of the competition.

Improving the accuracy of each event analysis benefits the quality of the physics properties being studied. Due to several intrinsic ATLAS experimental effects like energy and transverse momentum resolutions, the measured kinematic properties of particles produced in a collision may be shifted within a range of $\pm 1\%$, implying an uncertainty that is propagated through the event analysis. It is possible to improve the reconstruction quality by varying the values measured by the detector within that range, but with a significant impact to the analysis execution time, leading to a trade-off between the event processing throughput and their reconstruction quality.

Scientists at LIP developed the LipCbrAnalysis skeleton library to aid the development of data analysis applications. It contains a set of physics utilities, such as specific classes and functions, and removes the need to code the input file reading, memory allocation of each event data, and output creation for every data analysis application. With this, the programmer may focus on the specifics of the required

analysis, such as the filtering and reconstruction of events. An improved version was developed, LipMiniAnalysis, aiming to read a new type of input data files, and stripping the former skeleton of outdated features.

An efficiency study and optimisation of one of LIP production data analysis, also used as a case study for a background research on this issue, was presented in [16, 17]. It tackled the computational inefficiencies of the application on both homogeneous and heterogeneous platforms, and identified several limitations to performance scalability, specially when using hardware accelerators. The data analysis case study and the limitations identified with the LipMiniAnalysis skeleton are presented in a later section.

At the LHC, two proton beams are accelerated close to the speed of light in opposite directions, set to collide inside a specific particle detector. This head-on collision triggers a chain reaction of decaying particles, and most of the final particles interact with the detector, allowing to record relevant data. One of the searches being conducted at the ATLAS Experiment relates to the study of the Higgs boson couplings to top quarks. Figure 1.1 represents the final state topology of the associated production of two top quarks and one Higgs boson (that decays to $b\bar{b}$, two bottom quarks), known as $t\bar{t}H$ production. Figure 1.2 provides a schematic representation of the system to highlight the key features, such as the bottom quarks being jets of smaller particles, and the leptons (both l^+ and l^-) being a muon and electron in the t and $\bar{t}$ decays, respectively.

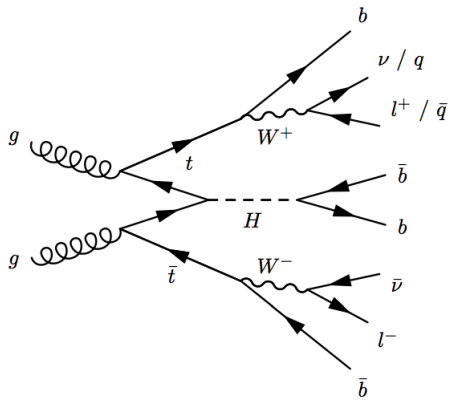

Figure 1.1: Feynman diagram of the $t\bar{t}$ and Higgs boson production.

Neutrinos ($v\bar{v}$) do not interact with the detector, so their characteristics are not recorded. Since the top quark reconstruction requires the neutrinos information, their characteristics are analytically determined with the remaining data, known as kinematical reconstruction. However, the $t\bar{t}$ system may not have a possible reconstruction: the reconstruction has an intrinsic uncertainty associated which determines its accuracy.

The amount of jets from bottom quarks and leptons present in the events may vary according to the decay channel of the W bosons produced in the top quark

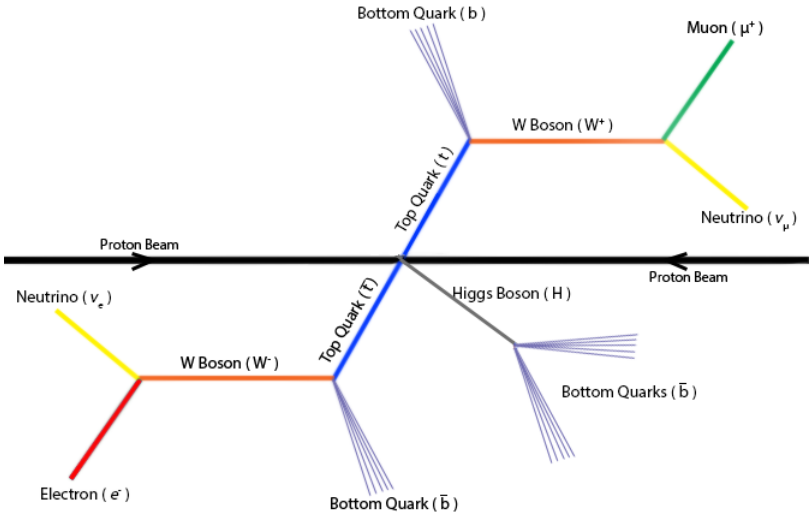

Figure 1.2: Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

decays. As shown in figure 1.2, four jets and two leptons are required to be present in the events. Two of the jets and two leptons are needed to reconstruct the $t\bar{t}$ system, and the remaining two jets are used for the Higgs boson reconstruction. For the kinematical reconstruction, every possible combination of jets and leptons must be evaluated and only the most accurate reconstruction is considered. If the $t\bar{t}$ system has a possible solution, the Higgs boson is reconstructed from the jets of the two remaining bottom quarks. The Higgs reconstruction does not use the jets that were associated to the best $t\bar{t}$ system reconstruction. The overall quality of the event processing depends on the combined accuracy of both reconstructions.

For the global event reconstruction, several solutions can be tested if we assume that the ATLAS detector has an experimental energy-momentum resolution of $\pm 1\%$, by varying these quantities within their uncertainty. This uncertainty is propagated into the $t\bar{t}$ system and Higgs analysis, affecting their accuracy. To improve the quality of the reconstructions several random variations are applied to the measured values, within a maximum range of $|1\%|$ next to the measured values, and apply the process explained previously for each variation. The quality of the event analysis and the application execution time is directly proportional to the amount of variations performed. The goal is to do as many variations as possible within a reasonable time frame.

The current version of `ttH_dilep`, the application developed to perform this analysis, is capable of reconstructing $\tilde{3}000$ events per second on a 2.60 GHz Intel Xeon CPU, without the accuracy improvement. Physicists consider that 1000 variations within the $\pm 1\%$ uncertainty would have a big impact on the analysis accuracy, but for `ttH_dilep` is only capable of processing 5 events per second with this precision. The reconstruction of the $t\bar{t}H$ system accounts for 99.8% of `ttH_dilep` execution time with this level of accuracy. To be viable to analyse the $t\bar{t}H$ system with 1000 variations it is crucial to improve the `ttH_dilep` efficiency.

1.3 Goals and Scientific Contribution

Dealing with scientific applications developed by scientists is not trivial due to the code structure and organisation. Several studies [18, 19, 20, 21] identified the causes that lead scientists to produce poor code:

- Most scientists are self-taught programmers with inadequate or outdated computer science background.
- Scientists disregard software engineering principles to produce efficient code that is also robust, modular, and long lasting.
- Scientists often iteratively develop over the same application, producing legacy code (some applications currently in production are iterated on for the last 20 years), and not documenting it so that it can be used by others.
- Scientists are seldom aware of profiling and debugging tools, as well as parallelization paradigms.
- Scientists cannot afford to get into the architectural details of the newer generations of computing systems, reducing the portability of the code they produce.

To improve the quality of the scientific code, scientists agree that it is crucial to create an interface between their field and computer science by having multidisciplinary teams. However, computer scientists often lack the expert field specific knowledge required to be acknowledged as an integral part of these teams. This often makes scientists sceptical to let others restructure, and even develop from scratch, legacy code that they have been using for years.

The goal of this PhD dissertation is to provide an efficient unified framework for the development of particle physics data analysis applications, designed in close cooperation with the LIP research group. It aims to give an abstraction to the current data analysis programming model, so that the user only codes the sections relative to each specific data analysis, while the framework guarantees portable efficiency for both homogeneous and heterogeneous platforms. The physics researchers will spend less time developing applications, while the framework ensures that the code is automatic parallelized and efficiently uses the computing power of both CPUs and accelerator devices, improving both the data analysis accuracy and throughput.

With a more agile development of high performance data analysis applications, researchers can spend more time improving the algorithms accuracy, which also require the extra computing power provided by the efficient use of multicore CPUs and manycore devices, and analysing larger amounts of data. These two factors have a significant impact on improving the quality of the physics research.

The specialised design of the framework for the specific field of particle physics data analysis allows to implement better automatic parallelization mechanisms than

the equivalent general purpose frameworks. On homogeneous platforms, it has been demonstrated in [17, 16] that a single shared or distributed memory parallel implementation may not provide the best efficiency when compared to an hybrid implementation. This framework will attempt to use hybrid parallel configurations in specific cases on a single computing system, while other frameworks assume that shared memory paradigm best suits all applications needs. On heterogeneous platforms, the framework will initially support automatic parallelization for both NVidia GPU and Intel Xeon Phi devices, with dynamic load balance among CPU and accelerator devices. The framework will use an efficient load balance library for heterogeneous platforms (most libraries only support GPUs), and possible extend its performance model to support the use of Intel Xeon Phi hardware accelerators.

1.4 Document Structure

This document is structured as follows:

Introduction: this chapter contextualises the work in section 1.1, and presents the scientific problem that motivates the PhD thesis proposal in section 1.2. Section 1.3 presents the goals and scientific contribution of this PhD.

State of the Art: this chapter presents the current hardware and software technology available for homogeneous and heterogeneous platforms in sections 2.1 and 2.2, respectively. Section 2.3 contextualises the current work on data analysis optimisation and available libraries for particle physics. Section 2.4 presents the available tools and libraries to profile parallel applications.

An Unified Particle Physics Framework: the current particle physics skeleton library used by the LIP research group is presented in section 3.1. Section 3.2 presents the conceptual design and preliminary prototypes of the efficient particle physics framework proposed for the PhD thesis.

Research Plan: the research plan for the PhD thesis work is presented in this chapter.

Chapter 2

State of the Art

Computing clusters are a common resource among scientific research groups. These massively parallel systems are usually constituted by racks of computing nodes interconnect by a low latency network, each running an individual instance of the operating system. The cluster operates on a distributed memory configuration, where shared data must be explicitly transferred among nodes. These cluster nodes may be dissimilar but use a common interface to communicate with each other.

Clusters are heterogeneous systems, since it may be constituted by nodes with different architectures, that use dedicated nodes to centralise the data storage and implement an abstraction layer to the user. When running an application, the user file system is mounted on the nodes that will perform the computation, but it is still needed to manually copy all necessary data to avoid unnecessary communication. The computing nodes architecture may be homogeneous or heterogeneous.

Both computer scientists and self-taught programmers are only used to code and design sequential applications, showing a lack of know-how to develop algorithms for parallel environments. This lack of expertise is even more evident when programming for heterogeneous systems, where programming paradigms shift among different hardware accelerators. The mainstream industry is still adopting the use of multicore architectures with the purpose of increasing their processing performance, which reflects in a lack of academic training of computer scientists on code optimisation and parallel programming. Self taught programmers have an increased obstacle due to the lack of theoretical basis when using these new parallel programming paradigms.

Programming for multicore environments requires some knowledge of the underlying architectural concepts of CPU devices and how they are interconnected. Shared memory, cache coherence and consistency, and data races are architecture-specific aspects that the programmer does not face in sequential execution environments. However, these concepts are fundamental not only to efficiently use the computational resources, but to ensure the correctness of applications.

Heterogeneous architecture of a single computing node combines the flexibility of multicore CPUs with the specific capabilities of manycore accelerator devices. How-

ever, most computational algorithms and applications are designed to the specific characteristics of CPUs. Even multithreaded applications cannot be easily ported to these devices expecting high performance. To optimise the code it is necessary a deep understanding of the architectural principles behind these devices design.

The workload balance between the cores of a single CPU chip is a key aspect to extract performance and get the most efficient usage of the available resources. An inadequate workload distribution may cause some cores of the CPU to be starved, unnecessarily increasing the application execution time. A good load balancing strategy ensures that all cores are used as much as possible. Considering a multi-CPU computing node in a Non Unified Memory Access space (NUMA), a key aspect is to manage the data in such a way that it is available in the memory bank of the CPU that will need it. If a task running on a CPU needs data stored on the other CPU memory bank, it must be communicated across a low latency channel. This communication increases the time that a task is stalled waiting for resources. The same concepts apply when balancing the load between CPU and hardware accelerators, with the increased complexity of the distributed memory environment and high latency data transfers.

Some computer science groups developed libraries that attempt to abstract the programmer from specific architectural and implementation details of these systems, providing an easy API as similar as possible to current sequential programming paradigms. The next subsections will present the architecture of a multi-CPU computing node, considered a homogeneous platform, and a multi-CPU node coupled with hardware accelerators, considered a heterogeneous platform, as well as software to aid the development of parallel applications in those environments.

2.1 Homogeneous Platforms Environment

Homogeneous systems are the most common computing platforms, constituted by one or more CPU devices with their own memory bank (RAM memory), and are interconnected by a specific interface. Although these systems use a shared memory model, where all data is addressable among multiple CPUs, each CPU has its own physical memory bank, which causes the system to have a NUMA organisation, as presented in figure 2.1. This means that the access time of a CPU core to a memory block in its memory bank will be faster than accesses to the other CPU memory bank. The threads of an application must have the data that they will use on the memory bank of their CPU device to avoid the increased communication costs of NUMA.

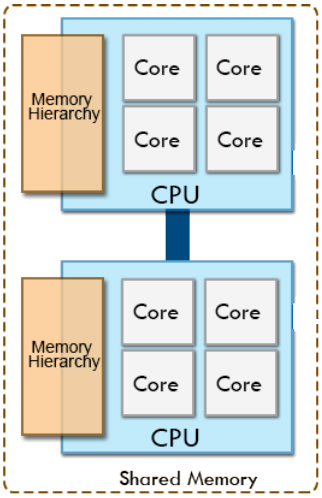

Figure 2.1: Schematic representation of a homogeneous NUMA platform.

2.1.1 The Hardware

CPU devices

Gordon Moore predicted, in 1965, that for the following ten years the number of transistors on the CPU chips would double every 1.5 years [22]. This was later known as the Moore's Law and it is expected to remain valid at least up to 2015. Initially, this allowed the increase in CPU chips clock frequency by the same factor as the number of transistors. Software developers did not spend much effort optimising their applications and they only relied on the hardware improvements to make the code faster.

The clock frequencies of CPU chips started to stall in 2005 due to thermal dissipation issues. Manufacturers shifted from making CPUs faster to increasing their throughput by adding more processing elements (known as cores) to a single chip, reducing their energy consumption and operating temperature. This marked the beginning of the multicore and parallel computing era, where every new generation of CPUs get wider, while their clock frequencies remain steady.

CPU devices are designed as general purpose computing units, and may contain multiple cores, each based on a simple structure of small processing units attached to a very fast hierarchical memory (cache, whose purpose is to hide the high latency access to global memory), and all the necessary data load/store and control units. They are capable of delivering a good performance in a wide range of operations, from executing simple integer arithmetic on scalar values to complex branching and vector instructions. A single CPU core implements various mechanisms to improve the execution performance of the code, at the hardware level. The more relevant ones are explained:

ILP instruction level parallelism (ILP) is the overlapping of instructions, performed

at both the hardware and software level, which otherwise would run sequentially. At the software level, ILP is implemented as static parallelism, as compilers try to identify which instructions are data independent, meaning that the outcome of one does not affect the execution of the other, and schedules them to execute simultaneously, if the hardware has resources to do so. At the hardware level, ILP can be referred as dynamic parallelism, since the hardware dynamically identifies which instructions execution can be overlapped while the application is running.

Vector instructions are a special instruction set based on the SIMD model (Single Instruction-stream, Multiple Data-stream), where a single instruction is simultaneously applied to a large set of data. CPUs offer special registers to allow executing an operation on a chunk of data in a special arithmetic unit. One of the most common examples is the addition of two vectors, where the hardware can simultaneously add several elements . This optimisation is often performed at compile time.

Multithreading is the hardware support for the execution of multiple threads in a CPU core. This is possible by replicating part of the CPU resources, such as registers, and can lead to a more efficient utilisation of the CPU core hardware. If one thread is waiting for data, other thread can resume execution while the former is stalled. It also allows a better usage of resources that would otherwise be idle during the execution of a single thread. If multiple threads are working on the same data, multithreading can reduce the synchronisation costs between them, as they both operate on the same CPU core, and may lead to a better cache usage.

2.1.2 The Software

Homogeneous systems often operate in a shared memory environment. Using multiple CPU devices may cause the memory banks to be physically divided but hardware mechanisms, such as specialised CPU interconnections, allow for a common addressing space. Libraries and frameworks for parallelizing this environment are presented next.

pThreads

Threads are the most simple parallel task that can be scheduled by the operating system. POSIX Threads (pThreads) are the standard implementation for UNIX based operating systems with POSIX conformity, such as most Linux distributions and Mac OS. The pThreads API provides the user with primitive for thread management and synchronisation. Since this API requires the user to deal with several low level implementation details, such as data races and deadlocks, the industry demanded the development of high abstraction level libraries, which are usually based on pThreads.

OpenMP, TBB, and Cilk

OpenMP [23], Intel Threading Building Blocks (TBB) [24], and Cilk [25] are the most popular high level libraries for parallel programming in homogeneous systems.

The OpenMP API is designed for multi-platform shared memory parallel programming in C, C++, and Fortran, for most CPU architectures available. It is portable and scalable, and aims to provide a simple and flexible interface for developing parallel applications, even for the most inexperienced programmers. It is based in a work sharing strategy, where a master thread spawns a set of slave threads and compute a task in a shared data structure. The latest specification also supports GPU hardware accelerators, with the integration of offloading directives. However, there is no efficient load balance among CPU and accelerator simultaneously.

Intel TBB employs a work stealing heuristic, where, after the initial load distribution, if the task queue is empty, a thread attempts to steal a task from other busy threads. It provides a scalable parallel programming task based library for C++, independent from architectural details, and only requires a Intel C++ compiler. It automatically manages the load balancing and some cache optimisations, while offering parallel constructors and synchronisation primitives for the programmer. However, it requires knowledge of the object oriented programming paradigm.

Cilk is a runtime system for multithreaded programming in C++. It maintains a stack with the remaining work, employing a work stealing heuristic similar to the Intel TBB.

2.2 Heterogeneous Platforms Environment

A new type of computing platform is becoming increasingly popular, with the emergence of specialised hardware designed to efficiently solve a specific set of computing problems. This marks the beginning of heterogeneous systems, where one or more CPU devices operate in a shared memory environment as in homogeneous systems, presented in section 2.1, and are coupled with one or more hardware accelerators. CPUs and accelerators operate in a distributed memory environment, meaning that data must be explicitly passed between the CPU and the accelerator by the programmer. Computing clusters are also considered as heterogeneous systems. However, the first stage of this work is focused on individual heterogeneous computing nodes and, if a parallelization for clusters proves viable for this specific problem, further research on this platforms will be performed. In this work, the concept of heterogeneous platform addresses a single computing node.

Figure 2.2 presents a schematic representation of a heterogeneous system. Multiple CPUs on the system use the same high latency interface to communicate with the hardware accelerators. This high latency PCI-Express interface is usually a potential bottleneck for applications that use accelerators.

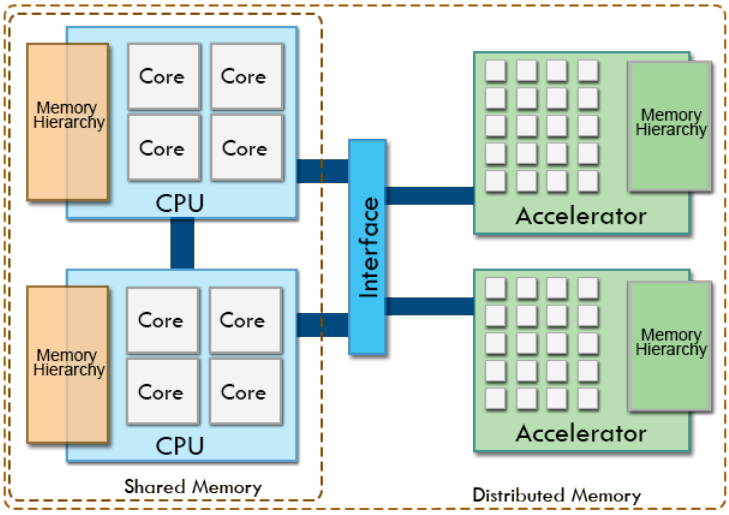

Figure 2.2: Schematic representation of a heterogeneous system.

Computing accelerators are usually constituted of a large number of small and simple processing units, aimed to achieve the most performance possible on specific massively parallel problems, as opposed to general purpose CPUs. This massive data parallel processing (SIMD execution model) offered by these accelerators, where a single operation is performed simultaneously on large quantities of independent data, have the purpose of offloading the CPU from such data intensive operations. Several manycore accelerator devices are currently available, with me most popular being the general purpose GPUs and the Intel Many Integrated Core line, with its production device known as Intel Xeon Phi [26]. An heterogeneous platform may have one or more accelerator devices of the same or different architectures.

As of June 2014, 62 of the TOP500's list [27] are computing clusters that use hardware accelerators, which indicates an exponential growth of these devices popularity compared to previous years. The Intel Xeon Phi is becoming increasingly popular, being the accelerator device of choice in 17 clusters of the TOP500, with 2 of those clusters on the top 10 (the fastest cluster, Tianhe-2, uses this device). NVidia GPUs remain as the most used accelerator, on a total of 44 clusters with 2 on the top 10, while the AMD devices are steadily losing their share. A similar list focused on consumption efficiency (computing power per Watt) Green500 [28], includes at the top 15 only clusters with NVidia GPU accelerators. The most popular hardware accelerators will be presented in depth in the next subsections.

2.2.1 Computing Accelerators

Graphics Processing Unit

The Graphics Processing Units (GPU) were one of the first hardware accelerators on the market. Their initial purpose was to speedup computer graphics applications, which started of as simple pixel drawing and evolved to support complex 3D scene rendering, such as transforms, lighting, rasterisation, texturing, depth testing, and display. Due to the industry demand for customisable shaders, this hardware later allowed some flexibility for the programmers to modify the image synthesising process. This also allowed using this GPUs as a hardware accelerator for wider purposes beyond computer graphics, such as scientific computing, as some researchers saw the potential to use these devices to boost the performance of numerical computation.

The GPU architecture is based on the SIMD execution model. Image synthesising is, from the computational point of view, the processing of a large set of values that represent pixels. The processing of each individual pixel usually does not depend on the processing of its neighbours, or any other pixel on the image, so, in the best case scenario, the computation has no data dependencies, which allows to process all pixels simultaneously. The massive data parallelism is the most important characteristic that was considered when designing the GPU architecture.

As GPU manufacturers allowed more flexibility to program their devices, the High Performance Computing (HPC) community started to use these devices to solve specific massively data parallel problems, such as numerical computation problems. However, the highly specialised architecture of GPUs affected the performance of many other different problem domains. Due to the increased demand for these devices by the HPC community, manufacturers began to generalise more of the GPUs features, such as adding support for double precision floating point arithmetic, and later began producing accelerators specifically oriented for scientific computing. NVidia is the main GPU manufacturer for scientific computing GPUs, with a wide range of available hardware boards known as Tesla. These devices characteristics differ from the original GPUs, as they have more GDDR RAM, a different structural design to fit in cluster nodes, and different cooling options. The chip itself is different, offering more processing units and larger memory caches. Kepler [29] is the latest GPU architecture released by NVidia, and some of its key features are detailed below.

Figure 2.3 shows the Kepler architecture organisation in two main components: the set of Streaming Multiprocessors (SMX) and the internal memory hierarchy. The focus of this architecture was not only on improving the performance but also the energy efficiency, offering up to to 3x more performance per watt than Fermi (the previous architecture). To achieve this efficiency, Kepler has implemented several features to improve the computational resource usage:

Dynamic Parallelism: a kernel (algorithm coded in CUDA) running on the GPU is capable of calling itself recursively, which allows to dynamically generate new

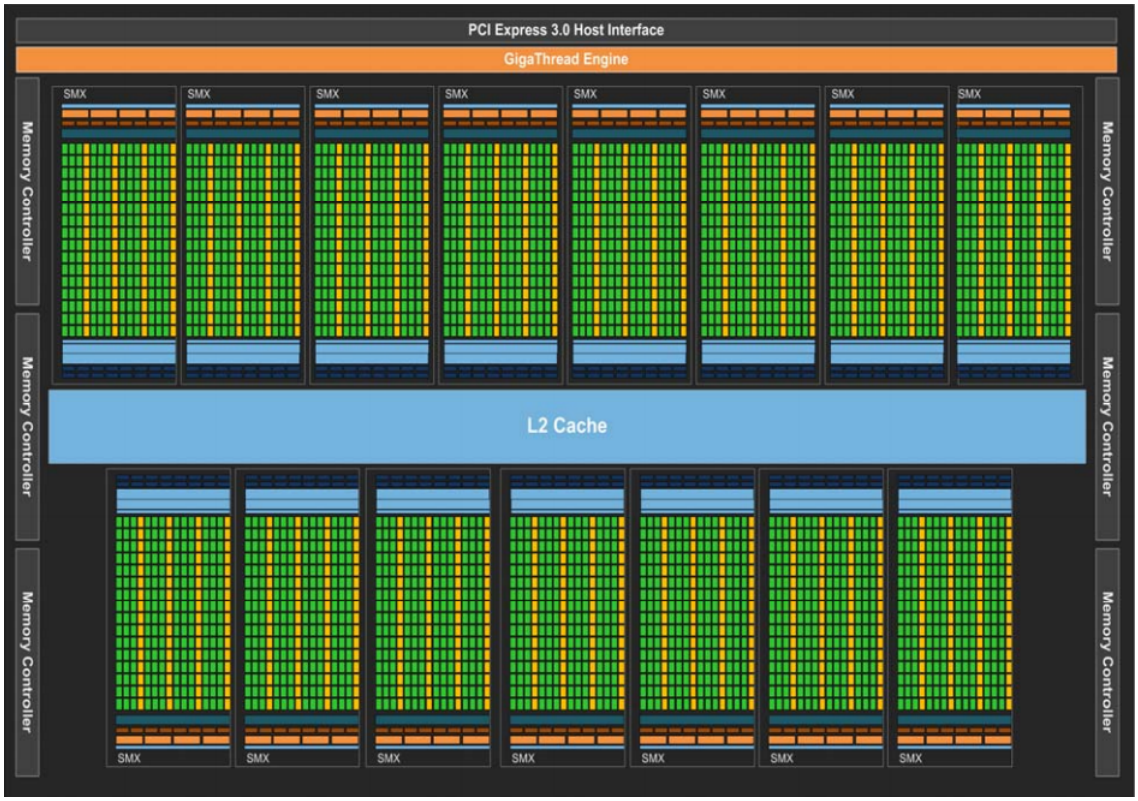

Figure 2.3: Schematic representation of the NVidia Kepler architecture.

workload to process without the CPU interference. This improves irregular algorithms performance on the GPU and reduces the communications to the CPU as the GPU is capable of managing the workload.

Hyper-Q: this technology increases the amount of work queues to 32 simultaneously hardware managed connections. It allows for multiple CPU cores to launch different kernels on the GPU simultaneously, improving the device resource usage. Multiple threads of the same application are able to share the GPU resources, reducing the amount of synchronisations.

Grid Management Unit: to allow for dynamic parallelism a new grid (a collection of threads of a kernel, explained in more detail in subsection 2.2.2) management system is required. The new system also allows to schedule multiple grids simultaneously, which allows for different kernels, from possibly different threads, to run concurrently (Hyper-Q).

NVidia GPUDirect: this feature allows GPUs in a single system, or in a interconnected network, to share data without the interference of the CPU and system memory, creating a direct connection to Solid State Drives and other similar devices, reducing the communication latency.

The SMX are complex processing units responsible for performing all computations on the GPU, and there may be up to 15 in a single chip. Each SMX has 192 single precision and 64 double precision CUDA cores, small processing units capable of performing basic arithmetic, 32 special function units, to perform complex computations such as trigonometric operations, and 32 load and store units. These computing units operate at the GPU main clock rate. The SMX features 4 warp schedulers (warps are presented in subsection 2.2.2) and 8 instruction dispatchers.

Each SMX has 64K 32-bit registers, with a maximum of 255 registers per CUDA thread, a 64 KB very fast memory for L1 cache and shared memory, and a similar fast 48 KB memory cache for read-only data. Finally, the Kepler architecture provides 1.5 MB of L2 cache shared among all SMX units. The high end available Tesla K40 has a memory bandwidth of 280 GB/s to its main memory. Since the GPU is connected by PCI-Express interface, the bandwidth for communications between CPU and GPU is restricted to only 12 GB/s (6 GB/s in each direction of the channel). Memory transfers between the CPU and GPU must be minimal as they may greatly restrict the performance.

A kernel is executed by a given amount of parallel workers named CUDA threads. They are grouped into blocks, to be scheduled among SMX and the threads inside a block can only run in a given SMX, and these are grouped into a grid, which contains all CUDA threads (up to $2^{31} - 1$) for a given kernel. The CUDA threads are grouped in batches of 32, called warps, to be dispatched by a warp scheduler. The scheduler has a scoreboard with up to 48 entries to manage which warps are stalled waiting for resources or data and which are ready to be executed.

Intel Many Integrated Core architecture

The Intel Many Integrated Core (MIC) architecture, with the current production device being the Intel Xeon Phi, is an emerging technology adopted by various clusters in the TOP500 list. It has a design different from the NVidia GPUs presented previously, opting to have fewer computing units but capable of performing more complex operations, and heavily relying on code vectorisation to extract performance. Figure 2.4 presents a schematic representation of the architecture. The current high end model, the Intel Xeon Phi 7120p, has 61 cores and 16 GB GDDR5 RAM. The device has three operating modes:

Native: the device acts as an independent system itself, with one core reserved for the operating system execution. The application and all libraries must be compiled specifically to run on the device, and later copied to the its memory along with the necessary input data, prior to its execution. No further interaction with the CPU is required until the application has executed.

Offload: the device acts an accelerator, such as a GPU. Only part of the application is set to run on the Xeon Phi, and data required by the code must be explicitly passed between CPU and the device. All library functions called inside the device must be specifically compiled for it.

Message passing: the device acts as an individual computing system in the network. Memory transfers are explicitly and the device can be programmed using the Message Passing Interface (MPI) [30]. The restrictions mentioned in the previous point are also applicable.

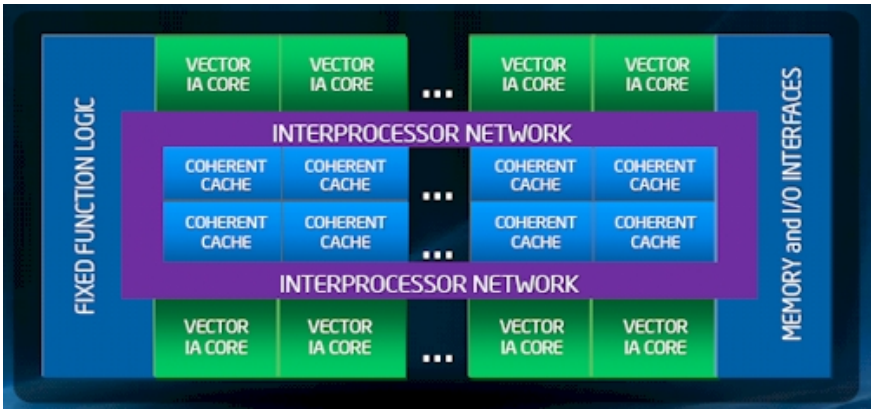

Figure 2.4: Schematic representation of the Intel Many Integrated Core architecture.

Each core is able to run 4 threads simultaneously, and most of the massive parallelism is obtained by using the vectorisation capabilities provided by the available 32 vector registers 512-bit wide. However, only a small set of vector operations are implemented in the hardware, and the most complex are emulated by the compiler.

Each core has 64 KB + 64 KB for data + instructions L1 cache, and 512 KB L2 cache. There is no shared cache among the 61 cores of the chip, and no cache consistency and coherence is automatically guaranteed among them. The cores are interconnected by a bidirectional ring network. MIC does not support out of order execution, which greatly compromises the use of ILP. Also, the clock frequency is limited to 1.1 GHz, which is less than half of the current CPUs.

Since it uses the same instruction set as conventional x86 CPUs, Intel claims that current applications can be easily ported to run on the device. This may be true for common matrix arithmetic and similar applications, efficient ports of complex applications that require the use of many external libraries is very difficult, or even infeasible [16].

The next iteration of the MIC architecture, known as Knights Landing, will provide out of order execution, better branch prediction, and implement all AVX vector operations in hardware, as in current Intel CPUs. It will also use a new instruction set, more similar to x86, to allow an easier port of most C++ features to the device.

Other hardware accelerators

Many alternative hardware accelerators are currently on the market due to the increasingly popularity of GPUs and Intel MIC among the HPC community. Texas Instruments developed their new line of Digital Signal Processors, best suited for general purpose computing while very power efficient. Their capable of delivering 500 GFlop/s (giga floating point operations per second), consuming only 50 Watts [31].

ARM processors are now leading the mobile industry and, alongside the new NVidia Tegra processors [32] that are steadily increasing the market share, are likely to be adopted by the HPC community[1] due to their low power consumption while delivering a significant performance [33]. Due to the increased complexity of mobile applications, the shift from 32 bit to 64 bit mobile processors has already happened, which will greatly benefit computing clusters using this type of hardware.

2.2.2 The Software

Heterogeneous systems use distributed memory address space to share data between multicore CPU devices and accelerator units. Even though the CPU devices work on a shared memory space, data must be explicitly passed to the accelerators. General purpose frameworks for parallelizing on the devices and on the heterogeneous platforms as a whole are presented next.

[1] e.g. the ARM based Montblanc project will replace the MareNostrum in the Barcelona Supercomputing Center (BSC)

Message Passing Interface

The Message Passing Interface (MPI) [30], designed by a consortium of both academic and industry researchers, aims to provide a simple API for process based parallel programming in distributed memory environments, typically in clusters. It relies on point-to-point and group messaging communication, and is available in Fortran and C.

MPI is often used in conjunction with a shared memory parallel programming API, such as OpenMP, for work sharing among computing nodes in a cluster environment, with the latter ensuring a more efficient parallelization inside each cluster node. There are studies on the performance of these hybrid implementations [34, 35], but only for Symmetric Multiprocessor systems (SMP). In these systems all nodes have similar characteristics, so OpenMP the parallelization inside each node is the same. For heterogeneous clusters, where each node may have different characteristics, this approach does not guarantee the best results.

Intel adapted an MPI version to work across their CPUs and Xeon Phi, considering the device as an individual computing node. Communications between the CPU and the device are explicitly handled by the programmer by calling specific functions. The other alternative to program this device with MPI is to use compiler *pragma* directives for data communication and code parallelization.

CUDA

The Compute Unified Device Architecture (CUDA) is a computing model for hardware accelerators launched in 2007 by NVidia and aims to provide a framework to program devices with an architecture similar to NVidia GPUs. It has a specific instruction set architecture (ISA) and allows programmers to use C extensions to program GPUs in scientific computing.

NVidia considers that a parallel task is constituted by a set of CUDA threads, which execute the same instructions coded in the kernel but on different data. For instance, in the sum of two vectors each CUDA thread will be responsible for the addition of a single element of each vector.

The CUDA thread is the most basic data independent parallel task, which can run simultaneously with other CUDA threads, and it is organised in a hierarchy presented in figure 2.5. A block is a set of CUDA threads that is allocated by the global scheduler to a specific multiprocessor. The thread blocks are organised in a grid, which represents the whole parallel kernel. Note that both the blocks and the grid sizes must be defined by the programmer, according to the algorithm, before calling the kernel, within the maximum values allowed by the GPU architecture. A warp is a subset of CUDA threads from a block that is set to run simultaneously on the multiprocessor.

Conditional jumps are a special type of instructions that must be avoided as they cause different CUDA threads within the same warp to diverge. Since a mul-

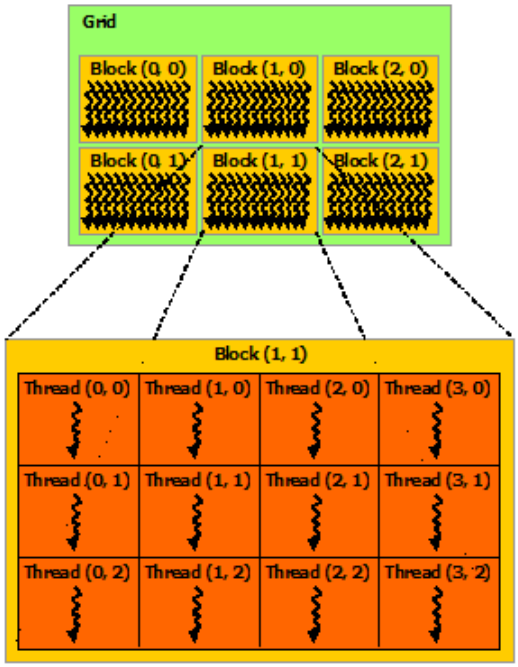

Figure 2.5: Schematic representation of CUDA thread hierarchy.

tiprocessor does not allow threads to execute different instructions simultaneously, the divergent branches will execute sequentially, doubling the warp execution time.

OpenACC

OpenACC [36] is a framework for heterogeneous platforms with accelerator devices. It is designed to simplify the programming paradigm for CPU/GPU systems by abstracting the memory management, kernel creation, and GPU management. Like OpenMP, it is designed for C, C++ and Fortran, it provides both an API and compiler directives, and allows the parallel task to run on both CPU and GPU at the same time. However, it does not schedule the load between the CPU and GPU, as it is only designed to offload the workload to the accelerators. The current specification addresses both NVidia and AMD GPUs, as well as the Intel Xeon Phi.

This framework focus on creating an abstraction of the hardware accelerator used, focusing on portability across heterogeneous platforms, rather than abstracting the intrinsic complexities of these systems.

OpenHMPP

OpenHMPP [37] is a standard similar to OpenACC, designed by CAPS [38] to develop parallel applications for heterogeneous platforms. It attempts to abstract the complexities of GPU accelerators by providing a set of compiler directives for

efficient parallelization. In the current specification, OpenHMPP uses a superset of the OpenACC directives for offloading code to the GPU and managing the data transfers, in both C and Fortran.

Although it provides asynchronous execution of the offloaded kernel, it is not possible to use this framework to manage simultaneous execution and load balance of the same kernel in both CPUs and GPUs. Moreover, it is only possible to use this specification with the CAPS compilers and PathScale ENZO Compiler Suite [39].

StarPU

StarPU [40] is a unified runtime system consisting on both compiler directives and a runtime API that aims to allow programmers to efficiently map parallel code into heterogeneous platforms by abstracting the architecture details of these systems. This framework frees the programmer of the workload scheduling and data consistency inherent from the distributed memory environment of heterogeneous platforms. Task submissions are handled by the StarPU task scheduler, and data consistency is ensured via a data management library.

StarPU attempts to increase performance by carefully considering and attempting to reduce memory transfer costs. This is done using history information for each task and, accordingly to the scheduler decision of where a task shall be executed, as it asynchronously deals with data dependencies while the system is busy computing the tasks that are ready. The task scheduler can take this into account, and determine where a task should be executed by considering not only the execution history, but also the estimation of data transfers latency.

StarPU employs a task based approach to the programming model, where a kernel is considered a parallel task. Based on the scheduler and available implementations for the kernel (i.e., can only run on CPU, GPU, or both), the framework handles where and how much load each task will compute. It provides a set of different schedulers for the programmer to chose.

The performance model differs among the schedulers implemented in StarPU, but most track the tasks execution time on the devices. All the schedulers use a user defined calibration to start the execution, and after 10 executions of each task it starts to perform a real-time calibration with the available statistics. This may translate in an inefficient usage of the system resources at the start of the application, but ensures that it tends to improve as the application runs.

The memory consistency is automatically ensured by the framework, as it transfers the data asynchronously without the programmer interaction. The data dependencies are determined by the scheduler, with some interaction of the programmer, when declaring if a data structure is read/write or both. The granularity of the tasks must be statically defined by the user.

DICE

The DICE framework aims to provide the tools to help building efficient and scalable applications for heterogeneous platforms with accelerator devices that support CUDA. It creates an abstraction layer between the architectural details of heterogeneous platforms and the programmer, aiding the development of scalable parallel applications. Its main focus is to obtain the best performance possible on irregular applications, as opposed to StarPU, and to achieve this goal it does not abstract all the architecture details from the programmer. It is still required to the programmer to have some knowledge of each different architecture and respective programming paradigms, and the framework needs to be instructed of how tasks should be divided in order to fit the requirements of the different devices.

Instead of relying in pre-partitioned work (the approach used by StarPU), the programmer defines a function for dicing the dataset and the framework creates different sized chunks of data to distribute among the CPU and GPUs. The framework frees the programmer from managing the workload distribution, memory usage and data transfers among the available devices, hiding the latency as StarPU, but requires that the application is built according to its strict specifications. The programmer is able to tune specific details related to the memory transfers and load balance, if he has the required expertise with the framework.

The scheduler uses the statistics provided by each job (a kernel set to run on a device) to adjust the scheduling policy and the granularity of the tasks. This dynamic granularity management allows to better suit the uneven execution times of irregular jobs. DICE uses a variant of the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm [41], which uses the computation and communication costs of each task, in order to assign every task to a device in such a way that minimises the estimated finish time of the overall task pool. This variant of HEFT attempts to make a decision every time it is applied to the task pool, so that tasks on the multiple devices take the shortest possible time to execute [42].

DICE assumes a hierarchy composed of multiple devices (both CPUs and GPUs, in its terminology), where each device has access to a private address space (shared within that device), and a distributed memory system among devices. To abstract this distributed memory model, the framework offers a global address space. However, since the communication between different devices is expensive, DICE uses a relaxed memory consistency model, where the programmer can use a synchronisation primitive to enforce memory consistency. DICE implements a shared software cache so that every device has the data as close as possible, using the local memory of each device. It also ensures that each device has a copy of a given data partition, which otherwise would only be stored in the CPU memory.

Legion

The Legion framework [43] is a programming model for heterogeneous platforms. It relies on an extensive specification of the data structure of an application to

provide a high performance parallelization and load balancing using both CPUs and hardware accelerators. It is targeted for users with extensive programming experience for heterogeneous platforms with MPI, OpenCL, and CUDA, and for users that aim to create high level languages and libraries optimised separately for each architecture.

The programmer needs to use a set of specific Legion data structures to ensure a subset of the provided data properties, such as partitioning and coherence. These properties need to be explicitly managed so that the framework is able to achieve an efficient load balance. With the specified data properties, Legion uses automated mechanisms to perform the execution parallelization, load balance, and communications. The user can provide the same algorithm coded for various architectures and the Legion mapping directives ensure their allocation to the proper device. The framework uses logical regions to abstract the data handling to the users. By using these regions, the programmer can enforce dependencies among tasks, slice the region space to increase the granularity of the tasks, and improve the scheduling and scalability of the algorithms.

Legion handles parallel execution of irregular loads by adapting the workload of each computing device and improving the task size during the application execution. It dynamically partitions the data according to the history of the execution time of the tasks for each device. The algorithm for slicing the data structure is coded by the programmer, similar to DICE. This benefits the scheduling on these distributed memory environments and in multiprocess parallelization among several computing nodes.

Applications can be coded in the provided Legion language or in C++, which integrates with its runtime API. The mappers and slice algorithms are coded in special classes, to later be queried by the runtime system during the application execution. Default mappers are provided in the current Legion package. The framework also provides a low level C++ API to allow programming for each specific architecture. It can be used in a shared memory mode, or in a distributed memory configuration that supports execution on large heterogeneous clusters. Currently, Legion only supports NVidia and AMD GPUs hardware accelerators.

2.3 The Computational Particle Physics

The data analysis applications developed by all CERN experiments contributors have common functionalities and use the same input data file format. This section presents an efficient study of a data analysis application that uses the LipMiniAnalysis skeleton library, as well as two libraries, ROOT and TopROOTCore, which are used in the particle physics field to aid the development of these applications.

The `ttH_dilep` Data Analysis Application Optimisation

A performance and optimisation study of a particle physics data analysis application of the $t\bar{t}H$ system was performed in [16, 17]. The `ttH_dilep` application was developed by LIP researchers, using the LipMiniAnalysis skeleton library, with dependencies on some ROOT and TopROOTCore features, to study and prove the $t\bar{t}H$ system properties in the Standard Model, which was presented in section 1.2. The flow of the application execution is presented in figure 2.6. The reconstruction algorithm implements the variation of the particles parameters, to overcome the $\pm 1\%$ uncertainty of the ATLAS detector. It is able to reconstruct around 3000 events per second, considering the detectors data as accurate, and 5 events per second when performing 1000 variations.

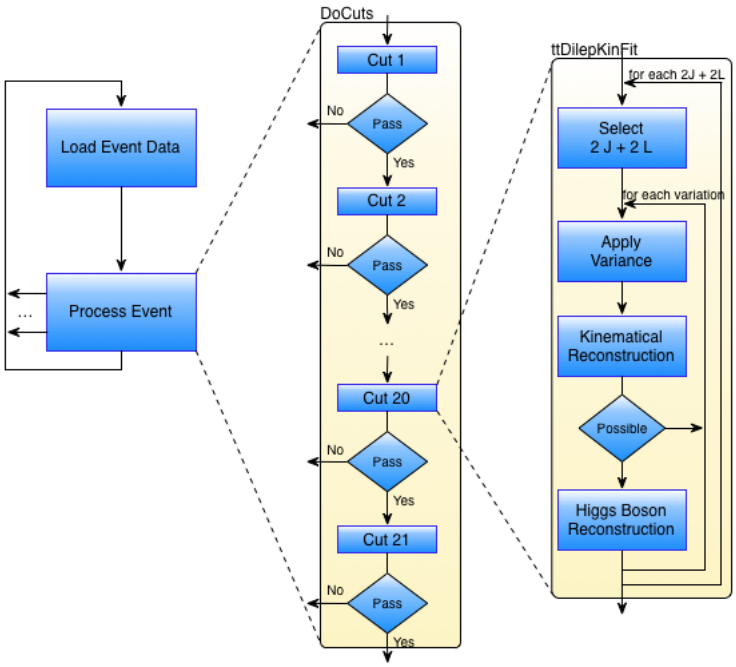

Figure 2.6: Schematic representation for the `ttH_dilep` application flow.

Each event data on an input file is individually loaded into a single global state, shared between the data analysis code and LipMiniAnalysis, which is overwritten for each new event loaded. The event is then submitted to a series of cuts, which filter events that are not suited for the kinematical reconstruction. When an event reaches the cut 20, the $t\bar{t}$ system and Higgs boson are reconstructed, which is expected to be the most computing demanding section of the code. If the $t\bar{t}$ system reconstruction fails, the current jet/lepton combination is discarded and the next is processed. If an event has a possible reconstruction it passes the final cut and its final information is stored.

The variation of the particles parameters was very inefficient. For 256 variations, it amounted to 63% of the `ttH_dilep` execution time, from which 23% was spent on

resetting the seed of the pseudo-random number generator. An analysis of the code revealed that the application uses the Mersenne Twister algorithm [44], resetting the seed for every 18 numbers generated. The seed reset is unnecessary as the Mersenne Twister period is approximately $4.3 * 10^{6001}$, while the maximum amount of pseudo random numbers generated by the application, for a given test input data file and 1024 variations, is $3 * 10^9$. The removal of this inefficiency granted a 71% performance improvement.

The best parallelization approach is to process concurrently several events from the same input file. However, the function that loads events from a file into memory in LipMiniAnalysis assigns the information to a single global space. This data structure contains information that is modified during the event reconstruction process, and it is overwritten for every event loaded. Changing the data structure to support multiple events in memory simultaneously, and loading all events in the input file at the beginning of the data analysis, would allow the parallel processing of events with low overhead. However, changes to the LipMiniAnalysis code were out of the scope of this work, so parallelization alternatives were designed and tested on a homogeneous platform[2]. A hybrid process/thread parallelization provided the best efficiency on that platform, with an overall speedup of 112. The event throughput increased from 5 to 560 events per second for 1024 variations.

A preliminary parallelization for CUDA capable GPUs was performed. Due to the dependencies of the code in functionalities of external libraries, only the kinematical reconstruction was offloaded to GPU. The marshalling and unmarshalling of the data and communication overhead greatly restricted the performance using the device. With only one event on memory at a time, it was not possible to hide the communication overhead by using multiple CPU cores and GPU simultaneously. For 256 variations on the detector data the CPU was idle 31% of the time, and the computations that were not offloaded to the accelerator had to be performed sequentially due to the LipMiniAnalysis data structure inefficiencies.

ROOT

ROOT [45] is a complex framework designed by particle physicists to aid all data analysis application development of the physics experiments conducted at CERN. It has all functionality required to process large amounts of data, by providing specific data storage formats, C++ classes for elemental particles, various physics algorithms, and histogram creation functions. The framework also provides a built-in C++ interpreter, Cling, to allow testing simple instructions and macros, without the need to compile and link the code.

PROOF is a subset of the framework to support the development of data analysis applications in distributed memory environments. However, it is only designed to work with a set of computing nodes, on a master/slave process hierarchy, without

[2] A dual Intel Xeon E5-2670v2 cluster node, each Xeon with 10 cores and hardware support for 2 simultaneous threads.

the support for hardware accelerators. Also, it does not focus on the efficient usage of the available computational resources, as it only distributes the load on demand among the processes.

Currently, ROOT does not provide any features parallelized, but the developers already shown interest to improve the performance of some of the core routines of the framework by parallelizing them on a shared memory environment. However, it may not translate in massive performance gains of the data analysis applications, as their critical regions are usually the reconstruction of the events, which do not rely on those complex ROOT functionalities, but rather on a large set of simple routines and classes.

TopROOTCore

TopROOTCore is an extension of ROOT for top quark physics, developed by CERN associate research groups, which adds features and physics algorithms to the existing framework. It is responsible for producing the last input data format at the last CERN computational tiers, before the final analysis and event reconstruction. In the data analysis applications, it is often used due to some physics algorithms it implements.

2.4 Profiling Tools and Libraries

Efforts to improve the efficiency of an application must be directed towards the bottleneck, i.e., the section of the code that takes the most time to compute. The identification of bottlenecks can be performed by analysing the application execution with specialised profilers, which then deliver a report with relevant information, such as each function execution time and memory usage. There are also profilers that provide an API to integrate with the code, performing a more refined analysis of the execution of specific parts of the code.

Performance API

The Performance API (PAPI) [46] specifies an API to access hardware performance counters in most modern processors. It allows programmers to measure the performance counters for specific regions of an application, evaluating metrics such as cache misses, operational intensity or even power consumption. This analysis helps classifying the algorithms and identify possible bottlenecks at a very low abstraction level.

PAPI recently supports hardware counters for both NVidia GPUs, using the NVidia CUPTI driver interface, and Intel Xeon Phi. It also supports counters to measure the energy efficiency of the hardware.

NVidia CUPTI

The NVidia CUDA Profiling Tools Interface (CUPTI) [47] is a performance analysis interface available in the NVidia drivers for CUDA capable GPUs. It provides a callback API to integrate with the code, at the entry and exit of a kernel call, which monitors the interaction of the code with the CUDA runtime and drivers. CUPTI has a second API to monitor the performance of a kernel on the GPU by analysing the hardware counters on the device, which allows for a in-depth assessment of the code behaviour in memory transactions, cache accesses and misses, and much more.

TAU and HPCToolkit

TAU [48] and HPCToolkit [49] are performance analysis tools, with static and dynamic functionalities, to evaluate the performance of HPC applications. The static APIs are low level and, while providing higher control of the areas to profile and specific metrics, require the programmer a deeper knowledge of these tools and how to integrate them with the existing code. The dynamic functionalities provide general metrics but do not require any changes to the application code.

Both tools provide statistical visualisation GUIs, to build graphs and comparisons of the different metrics profiled during the application execution time. Note that both tools support the analysis of parallel code in shared and distributed memory environments, but the HPCToolkit still does not support hardware accelerators. Unlike VTune, these tools only present the statistics but do not attempt to identify the bottlenecks, leaving that task to the programmer.

VTune

Intel VTune profiler [50] is a proprietary tool for performance analysis of parallel applications. It provides an easy to use interface to analyse applications, automatically identifying its bottlenecks, without requiring any change to the source code. It intercepts the system calls to assess the execution time and behaviour, such as efficient cache usage, of the routines of an application. VTune also provides visualisation functionalities to make the profiling of parallel applications a simple task for developers with small experience. It works with both Intel and GNU compilers.

VampirTrace

VmpirTrace [51] is a open source library to analyse an application execution on both shared and distributed memory environments, with support for CUDA capable GPUs through the CUPTI driver interface. It is capable of analysing the CPU hardware counters per thread/process by resorting to the Performance API. It has a low level API to integrate with the code to measure specific metrics and

regions of the code, and a more abstract interface that allows tracing the application execution without the need to change the code.

Additionally, VampirTrace allows to analyse the I/O interactions of an application, such as access times, types, and patterns to the hard drives.

NVidia Nsight

The NVidia Nsight [52] is a development platform for heterogeneous computing. It is available for both Visual Studio and Eclipse and aids the development of code for CUDA capable GPUs, with easy integration with current official production libraries. It has real time debugging functionalities to test code running on both CPU and GPU simultaneously. The built-in profiler allows to perform analysis to the kernels execution time on GPU, load and store efficiency (related to the coalesced accesses of CUDA threads to memory), multiprocessor occupancy rate, and memory usage. The profiling metrics are the same as the ones provided by the Performance API, as they both use the NVidia CUPTI interface.

Chapter 3

An Unified Efficient Particle Physics Framework

Developing parallel code for heterogeneous platforms is more complex than for homogeneous platforms. In a shared memory context, the data is always accessible by the programmer, as the different memory bank accesses on multi-CPU systems are managed by the compiler and hardware. Data dependencies and concurrent memory accesses still need to be managed by the programmer, which may require a significant level of expertise. To efficiently use the computational resources, dealing with problems such as false sharing and efficient cache usage, the programmer must have an advanced expertise on both the coding and architectural knowledge of homogeneous platforms.

Heterogeneous platforms are organised in a distributed memory environment, where the CPUs share the memory among each other but not with the hardware accelerators, which rises a new set of challenges. All communications of data between CPU and accelerator must be explicitly coded by the programmer, and has associated an added latency. The balance of the workload for each computing device to process becomes harder as it must take into account the data transfers and different characteristics of the devices.

Each different hardware accelerator has its own architectural design principles, as presented in section 2.2.1, which constrain their programming paradigm and the characteristics that both the algorithm and the code must have to efficiently use the computational resources. This implies that the programmer must be able to learn the hardware intrinsic characteristics and adapt to a new programming paradigm. Even for experienced programmers, porting current applications to run on heterogeneous platforms may be infeasible without redesigning the core features, which does not happen when designing a new application from scratch specifically for these platforms. Legacy code suffers the most from these issues.

Scientists are usually self-taught programmers that only develop applications as they are a necessary tool for the research in their field. Several studies, referred in section 1.3, identified a set of problems with the scientists coding practices and

scientific computing. Most of their code is in constant development, which may last for decades, adding and changing functionalities in each development iteration, disregarding most software engineering principles and not adapting the code to the hardware evolution. The few scientists that worry about performance attempt to optimise the code regions that they think are the bottleneck, without exploring the potential offered by profiling tools.

Since most scientists develop applications with the help of specialised frameworks of their research field, they expect that the tools coded efficiently and use the multicore capabilities of modern CPUs. However, the bottleneck is often on the developed code rather than in the framework, and these tools are not designed to automatically parallelize the bottlenecks and balance the workload among the available computing units.

Scientists opt to spend most of the research time on studying the problem and enhancing the algorithms rather than improving their programming skills to efficiently use the platforms available on modern computing clusters. They are even more reluctant to learn the new programming paradigms required to code for hardware accelerators on heterogeneous platforms. Several automatically parallelization and load balancing frameworks were developed for these systems by computer scientists to help the scientific community, as presented in subsection 2.2.2.

These general purpose frameworks usually have a steep learning curve, even for computer scientists. One significant setback of these frameworks is that, even if it is not explicitly required, the application must be designed to the framework characteristics, rather than the framework adapt to the application. As scientists are usually reluctant to redesign the very complex legacy code, which is difficult for computer scientists to understand without the expertise of the science field, an integration with these frameworks is infeasible. This problem also applies to the most of the external libraries used by these applications, as their functions are not coded to run on hardware accelerators and adapting the source code may not be possible.

Scientists are not willing to endure the steep learning curve of these frameworks to integrate with the development of future applications, and, as required by some frameworks, do not want to code two versions of an algorithm to run on the CPU and accelerator. The architectural complexity of these systems and the lack of guarantees to that the code performance will improve, due to poor implementation or algorithm characteristics, drives the scientists away from these frameworks. Finally, scientists attempt to produce applications with few external libraries dependencies, as they are not guaranteed to be supported through the application lifetime.

The existence of general purpose load balancing frameworks for heterogeneous platforms is useful, specially for computer scientists, but the scientific community lacks frameworks that both address the intrinsics of their scientific field, in which scientists can trust and rely, and the efficient usage of the computational resources, on both homogeneous and heterogeneous platforms. Frameworks such as these sacrifice the abstraction required to interact with any scientific field, but are more adapted

to the addressed scientific problem. A more concise framework orientation leads to an easy interaction of the scientist with the tool (and better abstraction of the parallelization complexities) and increases the computational efficiency of the code, when compared with general purpose frameworks. The main bottlenecks are usually known *a priori* and the framework is designed around the problem characteristics. The development of such frameworks may lead to a better interface between computer scientists and other scientific researchers, which may improve their codes and increase the quality of the research.

3.1 The LipMiniAnalysis Skeleton Library

The LipCbrAnalysis was a skeleton developed by two researchers of LIP in 2005, with the purpose of aiding the development of data analysis applications within the research group. Initially it served as an interface for dealing with the I/O of the data, converting ROOT formatted input data into variables in the skeleton global memory. Function prototypes for the common features among data analysis applications were declared on the skeleton. The programmer would code the functions to each analysis specific needs, knowing that all data was addressable from a global context.

Along the years the code was successively iterated to add support for new data file formats, physics functionalities, and general features, such as the support to pass options and arguments to the executable code. Some of the functions that the programmer needed to code are now fully implemented, with the option of being override by the user.

The LipMiniAnalysis is the latest development iteration of LipCbrAnalysis currently on production. It discarded features that were no longer necessary, and was adapted to read the new Mini Ntuple data format, hence its name. It is not a standard ROOT format, but results from a refinement and partial filtering of the events to improve the data quality.

Figure 3.1 presents the structure of LipMiniAnalysis, where all orange sections need to be coded by the programmer. When the application starts it sets the default values for all control information needed by the event filtering and reconstruction. The *Set User Values* section allows for the user to set its own control parameters, for both information defined by *Set Default Values*, overwriting the existing configuration, and other parameters not yet set. The *Get Command Line Options* section is responsible for defining and interpreting the options defined by the user when starting the executable. It is partially defined with standard options for every analysis, such as the definition of the systematics file and output directory, but the programmer can add new options as needed. The systematics parameters are automatically configured based on the input systematics file, and no user interaction is necessary. The preparation of both the input and output files is done automatically, but the declaration of the histogram vectors must be coded by the programmer, as it depends on the event type, filtering and reconstruction techniques applied.

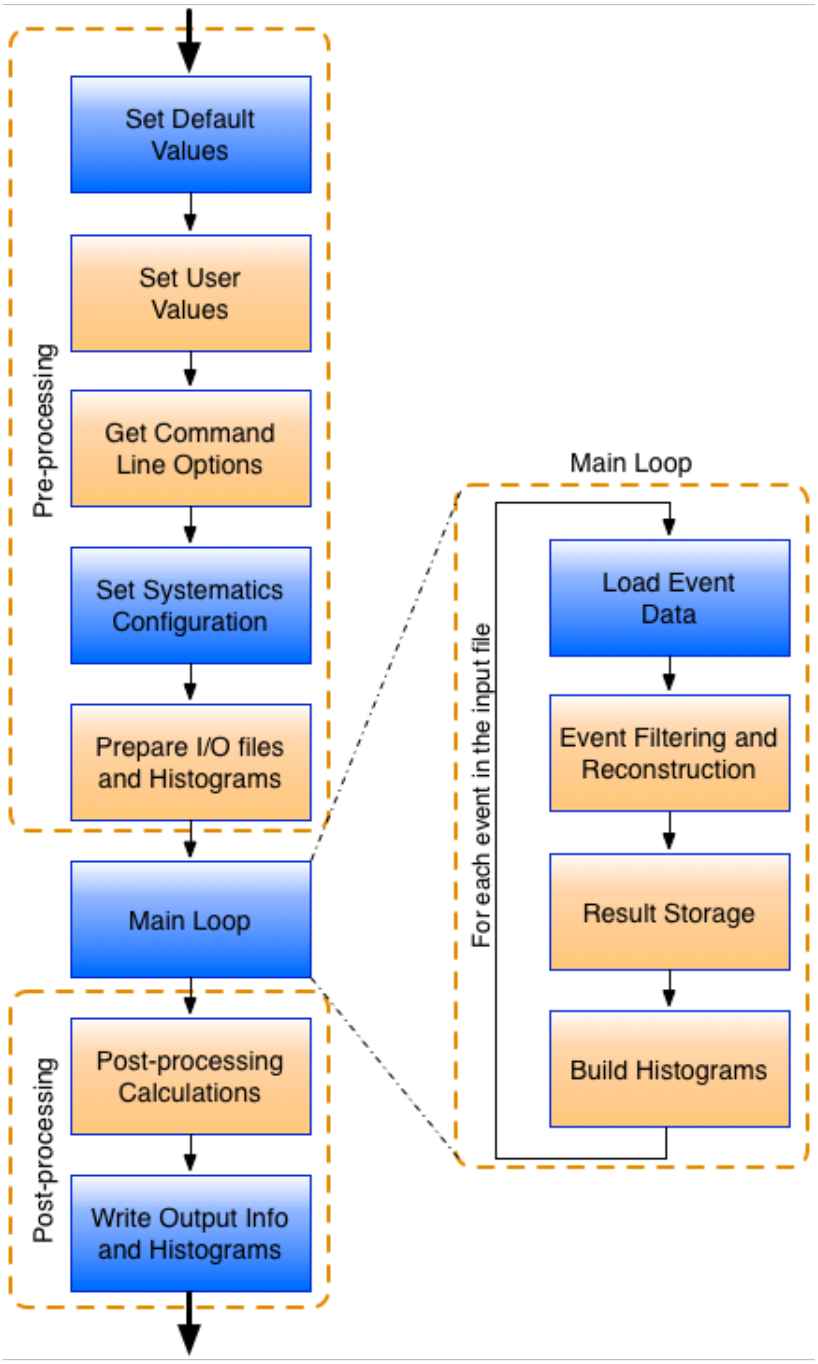

Figure 3.1: Schematic representation of the LipMiniAnalysis skeleton structure. The programmer is required to code the orange sections of the flow.

The *Main Loop* has a loop over all events on the input data file that loads a single event, applies the filters, reconstructs the event if it passes the required filters, stores the results, and builds the histograms for each filter and final reconstruction. Only the event loading is automatic, so the programmer is required to code all remaining sections, as they vary among different analysis. The final post-processing also depends on the analysis. Then, LipMiniAnalysis automatically writes all output information. Note that this is a logical structure of the code; in the current implementation most features are not properly organised and LipMiniAnalysis would have great benefits from a reorganisation of its code.

Studies presented in [16, 17] targeted the computational efficiency issues of a specific data analysis application of LIP, related to the reconstruction of the $t\bar{t}H$ system. This data analysis was developed using the LipMiniAnalysis and, although the goal of the work was to address only the inefficiencies of the data analysis itself, problems with the main data structure of the skeleton library restricted the performance scalability, specially on heterogeneous platforms.

The LipMiniAnalysis was designed to store only one event in the application global memory for the *Main Loop* to load and process. The data for an event is composed of hundreds of variables, ranging from simple scalars to complex vectors of ROOT classes. With only a single event in memory, it is more difficult to create an efficient parallelization with only the event reconstruction tasks, due to the low amount of work to balance, specially on distributed memory environments as it will require more communications in the application runtime. The filtering of events cannot be performed in parallel as the filters have data dependencies among them. With all events from an input data file on the application global memory, a more efficient parallelization for both shared and distributed memory systems can be achieved. It would also help the implementation of automatic parallelization and load balancing in LipMiniAnalysis.

3.2 The Proposed Framework for Particle Physicists

A physics framework is proposed to replace LipMiniAnalysis in aiding the development of data analysis applications. Its design and specification will include several software engineering concepts to provide a stable and robust tool that will increase the researchers productivity, spending less time coding and more time analysing data and improving physics algorithms. It will ensure the generation of efficient data analysis applications by providing automatic parallelization and load balancing mechanisms. It will include both redesigned features of LipMiniAnalysis and new particle physics functionalities.

Improving both the researchers coding productivity and data analysis computational efficiency has a direct impact on the research quality. To achieve this goal, the proposed framework has to utilise the capabilities of modern computing systems and

software. This implies that the LipMiniAnalysis code design, based on the LipCbr-Analysis developed in 2005, cannot be used as it does not fit the characteristics of modern hardware. The new framework will implement all required features present in LipMiniAnalysis but designed to modern hardware specifications.

Figure 3.2 presents the organisation and dependencies of the different modules of the proposed framework. A modular organisation and implementation allows for the framework to be robust and easily extensible in the future. The *Physics-related* and *Histograms* modules will be implemented using redesigned features currently available at LipMiniAnalysis, as they are used by most data analysis applications. The framework will have to support both ROOT and TopROOTCore libraries without requiring any specific configuration by the user. TopROOTCore installation with the framework may be an option to the user, since only a subset of data analysis applications use its functionalities.

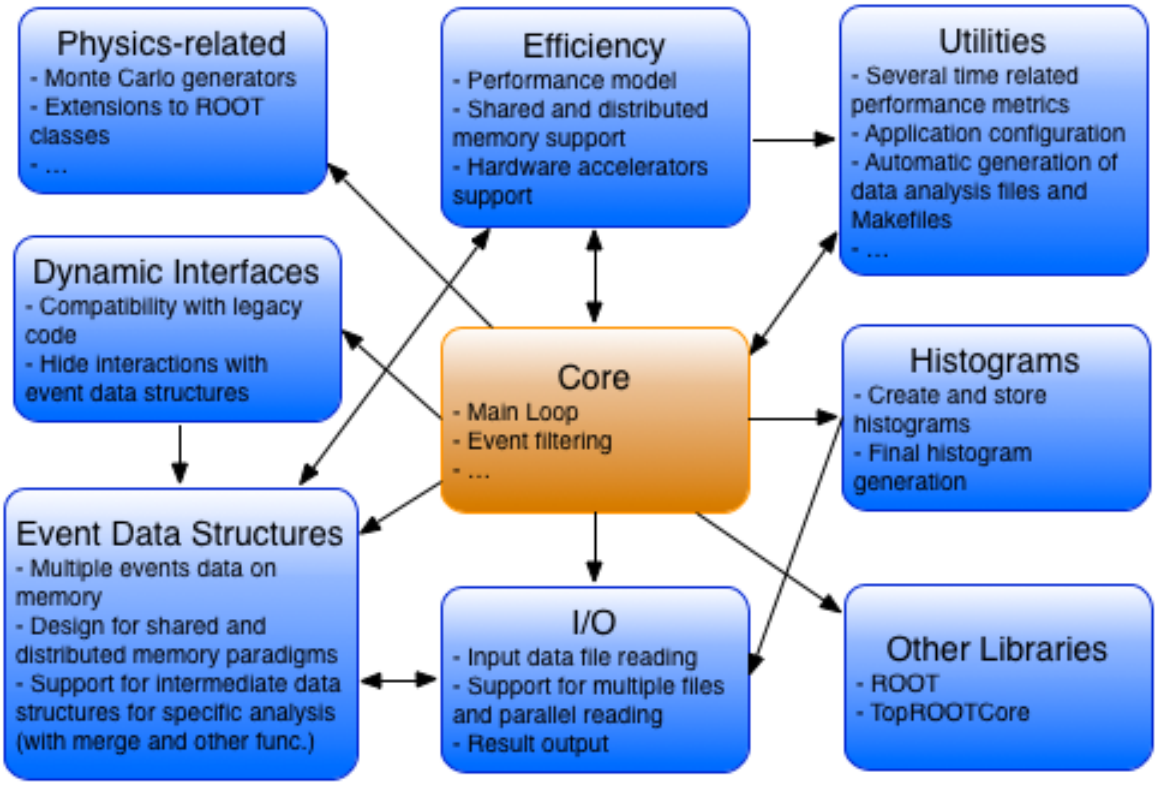

Figure 3.2: Schematic representation of the proposed framework modules and their dependencies.

The *I/O* module loads the input data files to process the events. The LipMini-Analysis reads only MiniNtuples, which are preprocessed events at the previous computational tiers, used by some applications, but other file formats will also be supported, after analysing the current formats used by researchers. Different file formats have different structures and auxiliary parameters, although the core event information is similar. The file reading must support parallel file descriptors, so

that it is possible that each thread/process reads its own event data, reducing the communications cost and initial load distribution overhead. Since the size of an input data file is around 1 GB, the framework will support a batch of input files and will manage their execution.

The *Event Data Structures* module will be dependent on the file type to be read. Its purpose is to have a C++ class, or set of classes, to hold all the information of an event on memory in a structured way, opposed to the current implementation on LipMiniAnalysis. The module will use a collection (vector, map, etc) to store the instantiations of the event class, enclosing all events in one or multiple input data files. An assessment of the different file formats used at LIP will be required to evaluated the benefits of having a single abstract event class or specific classes for each file format. Both the event class design and store collection must be suitable for both shared and distributed memory parallelization, with good performance on data structure splits (to balance the load, specifically in distributed memory paradigms), transverses (to iterate through the events on each split segment), and low overhead on (un)marshalling computations for communicating the data among processes. The module may also contain specific data structures for intermediate processing, as explained in detail in subsection 3.2.2, suitable to run on hardware accelerators.

The *Dynamic Interfaces* module will hold the interface generator and respective interfaces for each different event data structure. Its purpose is to make the framework compatible with legacy code, allowing for current data analysis applications to benefit from the automatic parallelization and improved efficiency of the proposed framework. Also, it must hide the interaction of the user with the data structures, to provide a simpler programming interface, which is explained in more detail in subsection 3.2.1. Since there may be several different data structures, and their code may be changed periodically, the interface generator must be capable of parsing the data structure source files and generate the interfaces at compile time.

The *Utilities* module implements several auxiliary features usable in both the framework and data analysis code. It will have simple performance statistics, such as execution time of the application, communications time, event processing throughput, etc. A more robust command line options reader will be available, with all required options for executing all different data analysis, which can be extended by experienced users.

The *Efficiency* module will have all necessary functionality required to create parallel tasks and manage the load and data distribution for both homogeneous and heterogeneous platforms with hardware accelerators. The performance model must be capable of assessing if an hybrid thread/process parallelization provides better efficiency than a simple multithreaded implementation, which was proven to happen in some data analysis [17]. There is no automatic parallelization tool that attempts hybrid implementations such as this, only opting for a shared or distributed memory paradigm (usually the last is only used when the hardware forces to), but it may prove beneficial to have this mix of processes and threads in some specific cases. This may require an extra overhead to obtain the best process/thread

configuration, but since these data analysis run for several hours the initial overhead is minimum. It can also produce a configuration file when the setup is performed for a given data analysis on a given computing system to avoid executing the initial setup every time the application starts. This module will use current efficient load balance frameworks, such as StarPU, DICE, or Legion, presented in section 2.2.2, as a middleware to manage part of the framework execution. This integration is explained in more detail in section 3.2.3.

This module must also be able to assess which sections of the data analysis code can be executed in the hardware accelerators, and if they have the required characteristics to efficiently use these devices. While this might seem complex for general purpose parallelization frameworks, dealing with only a specific problem of particle physics data analysis applications helps to design and adapt the performance model around this field features. A new framework implementation designed to this specific problem, with components adapted from current frameworks, may provide a better and simpler integration, and improve the efficiency of data analysis applications, which would not be possible by using existing libraries.

The *Core* module will integrate all previous modules, implementing the major routines responsible for the event analysis, ranging from the filtering to the final output data storage. It assembles all the specific bits of each module to process the events, as well as the code sections that are for the user to code. Note that the user will not edit the code in the framework, but rather create a data analysis source file that will link to the framework.

Relevant features for most compute intensive sections of data analysis applications will be implemented to run on CPU and accelerator devices and will be provided in the framework API. For example, when selecting an pseudo-random number generator to use in a compute intensive section, such as the event reconstruction, the user must avoid the `TRandom` available in ROOT and use the one provided by the framework. The framework compiles the code to run on CPU, which uses `TRandom`, and on GPU, which may use the `cuRand` (with the same PRNG algorithm). This only applies for features that produce the same result on any computing accelerator. Otherwise, an alternative is suggested for each computing device that the user may chose to accept.

3.2.1 Usage and Workflow

One of the goals of the framework is to provide a kernel-like programming model to the user, abstracting most parallel programming complexities. This eases the development of new data analysis applications and increases their portability across platforms. Its flow is presented in figure 3.3. Note that LipMiniAnalysis, presented in section 3.1, has a similar logical structure, but it does not reflect the code organisation and structure. In the new framework, the code will follow this structure to improve its modularity and reliability.

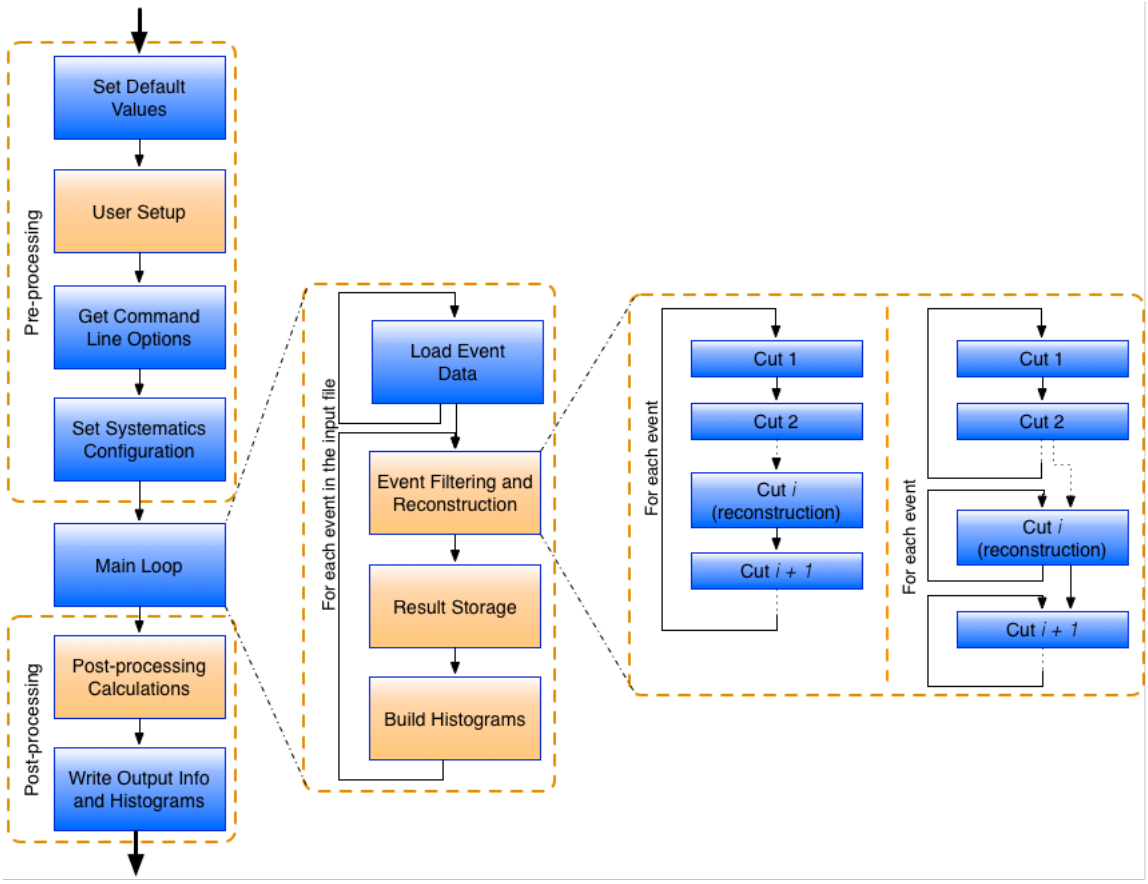

Figure 3.3: Schematic representation of the proposed framework flow. The programmer must code the orange sections of the flow.

The user intervention is required at four stages of the event processing (in terms of the implementation, the *Result Storage* and *Build Histograms* are coded together as *Filter Post-process*). The data analysis is expected to be implemented as a class, which will extend the `DataAnalysis` class provided by the framework, and must contain a set of predefined methods that represent each of the stages. The user codes the filters and reconstruction for a single event assuming that the data is on global memory. The user does not interact directly with the event data structure, but through the respective interface, which gives the illusion that only a single event is on memory, as physicists were used to program so far. The framework will be responsible for applying the code in parallel to all events on memory, not necessarily following the pipeline order but respecting the dependencies.

The data analysis stages are characterised as follows:

***User Setup*:** this is only executed at the beginning of the data analysis execution, where the user specifies the initial *User Values*, as in the LipMiniAnalysis flow, output files, and histogram configurations. Expert users can also specify any additional parameters to read by both command line and environment variables, using the respective framework utilities, and a small set of configurations to help the parallelization setup.

***Event Filtering and Reconstruction*:** this section filters and reconstructs a single event. As shown in figure 3.3, two templates are provided. The first is the traditional pipeline for the event processing, oriented for novice users. The second provides three sections: pre-filtering, reconstruction, and post-filtering. The user codes in the filters applied before the reconstruction, the reconstruction, and then the final filters (if applicable), in three separate methods. This is useful when the reconstruction takes most of the execution time as it allows for better load balance, specially when the framework can use accelerators for the reconstruction.

***Result Storage* and *Build Histograms*:** this is coded as a single method, since both operations are closely dependent in most implementations. The user store the results of the previous filtering to later be printed in the output file.

***Post-processing Calculations*:** this stage is executed once, after processing all events. The user codes all final calculations before the results and histograms output.

The design of the stages required by the `DataAnalysis` class will be refined after the requirements elicitation and an usability study with the LIP researchers. All major features initial design and testing was performed in close cooperation with a subset of LIP physicists, working on top quark and Higgs boson research.

The programmer will have some guidelines to avoid producing an inefficient data analysis. Class variables in `DataAnalysis` must be avoided because of the amount of communications required to maintain the consistency and coherence of that data

on a distributed memory environment. To maintain portability, the data analysis must only depend on the libraries provided by the framework (at the moment, there is not any data analysis in LIP that depends on other libraries). The section of the data analysis that is parallelized among multicore CPUs and manycore accelerator devices must not depend in any external libraries not supported by the framework. This lets the user code the algorithm so that it is able to execute on any computing device. Otherwise, the code will be restricted to the device that can compute it.

3.2.2 Preliminary Prototypes

Some features of the framework modules were already developed. These prototypes were integrated and tested with the current version of LipMiniAnalysis, using the $t\bar{t}H$ data analysis application presented in section 2.3, but have a modular design to be properly merged into the final framework. The proposed prototypes of these features are presented next.

A new event data structure

The information of an event is loaded in the *Main Loop* into a single global memory state. The hundreds of variables of an event are spread among a set of files of LipMiniAnalysis. To create a new event data structure all these variables were merged into a single C++ class, which represents a single event, named *EventData*, and use a standard collection, such as a STL vector, to store all events of an input file. However, the current LipMiniAnalysis version performs a set of data preparation routines, named *FillAllVectors* and *Calculations* (the latter needs to be coded by the user). As their implementation is only prepared to access the data as it was in the global memory, they will not be compatible with the new data structure.

The implementation of *Calculations* could be modified access the values stored in the new data structure. However, the user would have to be aware of the data structure interaction and characteristics to properly code the required data preparation. Since *Calculations* only accesses the data of an event, it makes much more sense to code it as a method of the event class. The current implementation of *EventData*, the *FillAllVectors* routine, which performs the initialisation of some of the components of an event, is coded as a method, and the *Calculations* is declared as a virtual function, so it can be coded by the user in the analysis source file, without the need to interact with LipMiniAnalysis.

A new *Main Loop* design

Having a new data structure that allows multiple events to be simultaneously on memory implies changes to the way that LipMiniAnalysis handles the input data files. Instead of loading an event at a time, the *Main Loop* implementation was changed to load all events in an input data file at once, and store them in the new

specialised structure, as presented in figure 3.4. Now, it is possible to parallelize the execution of the *Event Filtering and Reconstruction*, performed in the *DoCuts* function. In physics terminology, a filter is addressed as a cut.

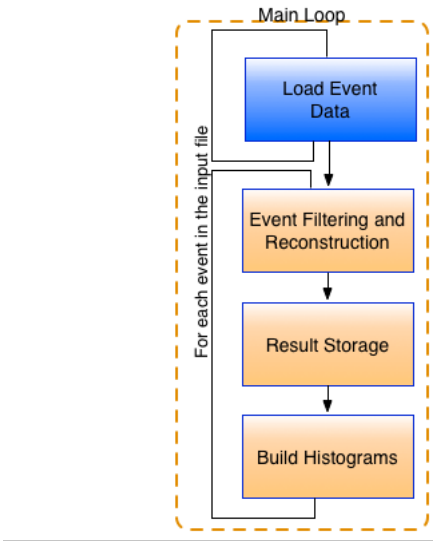

Figure 3.4: Schematic representation of the new *Main Loop* implementation.

The interaction of the LipMiniAnalysis with the input data file is performed using the ROOT file reader classes. Since the event loading assumes a set of operations, such as storing MonteCarlo information on *EventData* and the *FillAllVectors* initialisation, this task would benefit if it was executed in parallel, as the I/O itself only amounts to a small part of the computation. However, up to version ROOT v6, released in early June, it was not possible to have parallel file descriptors reading information of the same input file. With the new ROOT version it may be possible but it was not yet tested.

The purpose of the filters is to separate the signal (interesting events) from the background (uninteresting events). This notion depends on the physics that the analysis studies as, for example, the background of a $t\bar{t}H$ system analysis may contain events useful for the search of heavy quarks. As it may vary among different analysis, the *DoCuts* must be coded by the user, but its structure is well defined. Implementation-wise, an analysis has a set of filters. Each filter is logically composed by a set of computations, more or less complex, and a test to the results of those computations. If the test fails, the event is discarded. An example of a simple filter is to check if the mass of the event system (aggregate masses of all or a subset of particles detected) is greater than a given value.

The reconstruction of an event is considered a filter in *DoCuts*, as only the events capable of being reconstructed pass the filter, which is usually the last one. The reconstruction is usually the most complex and computational intensive task in the whole data analysis application. One example is the $t\bar{t}H$ system reconstruction, where the performance depends on a trade-off between reconstruction accuracy and

execution time. The event reconstruction is a task that may require a closer analysis to improve the efficiency of the application, through a more specific parallelization on both homogeneous and heterogeneous platforms. Its irregular workload, very complex algorithms, and few vectorised numerical computations create a complex bottleneck, but has the potential of greatly improving the performance when properly optimised [17]. Figure 3.5 presents the implemented prototype for a new *DoCuts* flow that exposes the event reconstruction to more efficient parallelization mechanisms.

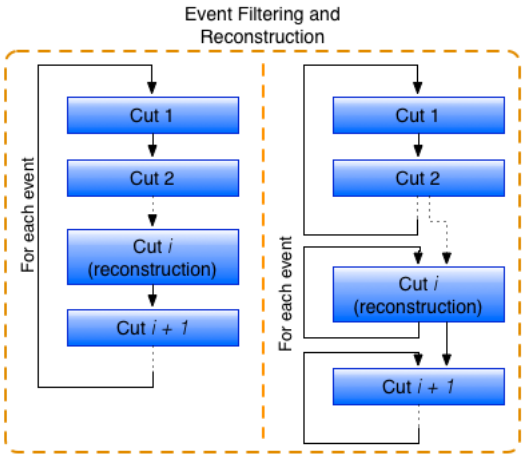

Figure 3.5: Schematic representation current (left image) and proposed (right image) *Event Filtering and Reconstruction* flows.

Besides exposing the parallelism of a complex computational intensity section, this change also increases the events to be simultaneously reconstructed. When the application reaches the reconstruction, all events that passed the previous cuts can be processed in parallel. Having all the necessary data outside of the *Main Loop* loop allows for a better load balance of the possible parallelization approaches using hardware accelerators. It also helps to reduce the parallelization overhead associated with the communications in heterogeneous systems: instead of passing required the data of a single event multiple times, the data of all events is passed once. There is a third section of cuts (post-reconstruction) that are also coded by the user. To reduce the amount of loops, this third section shares the loop over all events with all the remaining post-processing of the *Main Loop*. Note that both *Main Loop* options will be available to maintain compatibility with the legacy data analysis applications.

Interfacing with legacy code

Current data analysis use the data of an event as it is on the global memory, with no structure. The integration of the new event data structure requires that the accesses to the data be made through the STL collection used to store the event and

the *EventData* class. An interface is proposed to avoid rewriting the legacy code of these data analysis, providing a kernel-like programming approach (where the code for processing an event is applied to all simultaneously). The interface must abstract the access to both the *EventData* variables and methods.

The input data file format occasionally suffers some changes due to the increase in information given by the previous preprocessing on the CERN computational tiers. The file reading and the *EventData* classes have to be adapted to this increase in event parameters. A static interface would need to be rewritten every time such change occurred. A parser was developed to solve this problem by receiving the *EventData* source files and retrieving the name of the methods and parameters. It then creates a header composed of *define* clauses that translate the accesses to these variables on the STL collection to simple global accesses. This header is then included in the main LipMiniAnalysis so that the user does not need to interact with the interface. This interface is automatically created every time the skeleton is compiled.

With this, the user can code the data analysis assuming a single event is stored in memory. For example, to access an event luminosity the user would write `int var = LumiBlock`. However, when compiling the application the compiler preprocessor uses the interface to replace that statement with `int var = events.get(currentEvent).getLumiBlock()`, without the user knowing how the event is stored. The framework manages the counter that assigns the event to process in every loop that iterates through the event data structure.

Other features

The framework will have an utility module where general features will be included. Some of the already implemented range from automatic execution time measurement of the application and event processing throughput, definition of the number threads to execute, and setting the accuracy of some reconstructions. It was opted to use environment variables to set these features to reduce the clutter of options currently passed to the data analysis applications (usually more than 10 parameters) and separate general purpose features from physics functionalities.

One complex feature was implemented specifically for the $t\bar{t}H$ data analysis application. During the event reconstruction, several different variations of the system are reconstructed and only the best is of interest. For that a class was developed that encloses the resultant data of the reconstruction and performs a parallel merge through all the threads. This feature might prove useful for other data analysis once the structure of the class is general enough to hold the result of different types of reconstructions.

3.2.3 The Underlying Middleware for Heterogeneous Environments

Subsection 2.2.2 presents various middleware frameworks for efficient load balancing in heterogeneous platforms. The most relevant are StarPU, DICE, and Legion, which provide efficient parallelization and load balance mechanisms for irregular workloads, but only support GPU hardware accelerators. Since these frameworks are target for users with expertise in parallel programming for heterogeneous platforms, and they require that the applications to be developed according to their specifications, most non-computer scientists are reluctant to use them.

The high performance and portability offered by these frameworks might be useful when integrated with the proposed particle physics framework. However, these libraries do not fill all the framework requirements: efficient load balancing and parallelization using various multithreaded processes and support for GPUs and Intel Xeon Phi hardware accelerators. StarPU and DICE only support a multithreaded parallelization using CUDA capable GPUs, while Legion also supports OpenCL and multiprocess parallelizations. DICE seems to be better than StarPU, as it dynamically adapts the granularity of the workload using each device execution history, while the latter relies on a static grain size. Legion has the advantage of supporting multiprocess parallelization, which was proven to benefit specific data analysis applications, even within a single computing node. A detailed comparative efficiency study will be performed to assess which one provides the best support for this specific problem, as they will be required to balance complex heterogeneous code and data structures that possibly cannot be ported to CUDA and OpenCL.

One crucial characteristic that the library must have is the capability of being extended by third party developers to support other hardware accelerators. This implies changes to the performance model and implementation of other features required to run the parallized code on the new accelerators. The integration of the Intel Xeon Phi with the library has to be feasible, as it is the accelerator that best suits the execution of complex code of the data analysis applications. The proposed framework design will be adapted to the library specifications, to ensure efficient execution on heterogeneous platforms.

Chapter 4

Research Plan

Efficient load balancing in heterogeneous platforms is a complex task due to the different programming models among computing devices, specially for non expert programmers. This dissertation work proposes a automatic load balancing framework for the development of particle physics data analysis applications. To achieve this goal, the framework has to integrate, and possibly extend, a specialised load balancing framework as middleware, and redesign the current data analysis programming model of particle physicists. Both these tasks are individually complex and cannot be addressed separately.

Prototypes of some of the proposed framework core features were already developed and tested to validate the current design. However, it is crucial to perform the final requirements elicitation of the physics and computational features, together with the LIP researchers, to refine the framework design. The new data analysis programming model offered by the framework also needs to be refined and validated with the LIP researchers. This tasks will take approximately 1 month. With the clear design of the framework, the prototypes need to be updated and extended, to improve the data structure and parallel I/O features, and refine the automatic parallelization and load balancing in multi-CPU shared memory environments. An assessment of the performance of DICE, StarPU, and Legion will be performed in parallel to the framework improvement. These tasks are expected to take 4 months. A preliminary version of the framework is expected to be presented in March 2015.

After an efficient load balancing framework is chosen it is necessary to integrate it with the particle physics framework as middleware. It will be refined during the integration, as the programming constraints imposed by GPU accelerators will have a considerable impact on the framework design. It is needed to assess if the middleware can be extended to support Intel Xeon Phi accelerators and then perform the required changes to the framework to support the device. This stage of the work is expected to take 6 to 8 months.

The framework middleware will be improve by refining the Intel Xeon Phi support and implement support for efficient load balancing among several computing nodes in a cluster environment. The latter will explore innovative approaches for

hybrid OpenMP/MPI/CUDA parallelization. The framework physics features will be extended to update the input data file support and other functionalities. This stage is expected to take 10 months. In the last 12 months, the performance of the framework must be assessed in terms of efficient use of the computational resources in heterogeneous platforms, and by quantifying the productivity improvements of developing data analysis applications. At this stage, the framework must be validated by integrating it with current data analysis applications used by the LIP research group. The relevant results will be compiled into one or more scientific papers.

References

[1] European Organization for Nuclear Research. *CERN European Organization for Nuclear Research*. Nov. 2012. URL: http://public.web.cern.ch/public/ (cit. on p. 6).

[2] European Organization for Nuclear Research. *The Proton Synchrotron*. July 2013. URL: http://home.web.cern.ch/about/accelerators/proton-synchrotron (cit. on p. 6).

[3] European Organization for Nuclear Research. *The Large Hadron Collider*. Nov. 2012. URL: http://public.web.cern.ch/public/en/lhc/lhc-en.html (cit. on p. 6).

[4] European Organization for Nuclear Research. *Compact Muon Solenoid experiment*. Nov. 2012. URL: http://cms.web.cern.ch/ (cit. on p. 6).

[5] European Organization for Nuclear Research. *ATLAS experiment*. Nov. 2012. URL: http://atlas.ch/ (cit. on p. 6).

[6] European Organization for Nuclear Research. *The Large Hadron Collider beauty experiment*. Nov. 2012. URL: http://lhcb-public.web.cern.ch/lhcb-public/ (cit. on p. 6).

[7] European Organization for Nuclear Research. *The Monopole and Exotics Detector at the LHC*. Nov. 2012. URL: http://moedal.web.cern.ch/ (cit. on p. 6).

[8] European Organization for Nuclear Research. *Total Cross Section, Elastic Scattering and Diffraction Dissociation at the LHC*. Nov. 2012. URL: http://totem.web.cern.ch/Totem/ (cit. on p. 6).

[9] European Organization for Nuclear Research. *The Large Hadron Collider forward experiment*. Nov. 2012. URL: http://home.web.cern.ch/about/experiments/lhcf (cit. on p. 6).

[10] European Organization for Nuclear Research. *A Large Ion Collider Experiment*. Nov. 2012. URL: http://aliceinfo.cern.ch/ (cit. on p. 6).

[11] Georges Aad et al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Phys.Lett.* B716 (2012), pp. 1–29 (cit. on p. 7).

[12] European Organization for Nuclear Research. *Computing*. July 2013. URL: http://home.web.cern.ch/about/computing (cit. on p. 7).

[13] European Organization for Nuclear Research. *Animation shows LHC data processing*. July 2013. URL: http://home.web.cern.ch/about/updates/2013/04/animation-shows-lhc-data-processing (cit. on p. 7).

[14] European Organization for Nuclear Research. *The Worldwide LHC Computing Grid*. July 2013. URL: http://wlcg.web.cern.ch/ (cit. on p. 7).

[15] Laboratório de Experimentação e Física Experimental de Partículas. *Laboratório de Experimentação e Física Experimental de Partículas*. Nov. 2012. URL: http://www.lip.pt/ (cit. on p. 8).

[16] André Pereira. "Efficient Processing of ATLAS Events Analysis in Homogeneous and Heterogeneous Platforms". MA thesis. University of Minho, Sept. 2013 (cit. on pp. 8, 9, 12, 23, 29, 39).

[17] A. Onofre A. Pereira and A. Proença. "Removing Inefficiencies from Scientific Code: the Study of the Higgs Boson Couplings to Top Quarks". In: *The International Conference on Computational Science and its Applications* (July 2014) (cit. on pp. 9, 12, 29, 39, 41, 47).

[18] Zeeya Merali. "ERROR... Why scientific code does not compute". In: *Nature, pp. 775-777* (467 Oct. 2010) (cit. on p. 11).

[19] Jo Erskine Hannay et al. "How do scientists develop and use scientific software?" In: *Proceedings of the 2009 ICSE workshop on Software Engineering for Computational Science and Engineering*. 2009, pp. 1–8 (cit. on p. 11).

[20] Prakash Prabhu et al. "A survey of the practice of computational science". In: *State of the Practice Reports*. 2011, p. 19 (cit. on p. 11).

[21] Jeffrey C Carver et al. "Software development environments for scientific and engineering software: A series of case studies". In: *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. 2007, pp. 550–559 (cit. on p. 11).

[22] Gordon E. Moore. "Cramming more components onto integrated circuits." In: *Electronics, 38(8)* (Apr. 1965) (cit. on p. 15).

[23] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Tech. rep. July 2013 (cit. on p. 17).

[24] James Reinders. *Intel Threading Building Blocks*. Tech. rep. 2007 (cit. on p. 17).

[25] Robert Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 207-216* (July 1995) (cit. on p. 17).

[26] Intel. *The Intel® Xeon Phi Datasheet*. Tech. rep. Apr. 2014 (cit. on p. 18).

[27] TOP 500. *June 2014*. June 2014. URL: http://www.top500.org/lists/2014/06/ (cit. on p. 18).

[28] GREEN 500. *June 2014*. June 2014. URL: http://www.green500.org/lists/green201406 (cit. on p. 18).

[29] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Tech. rep. 2012 (cit. on p. 19).

[30] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: (Sept. 2004), pp. 97–104 (cit. on pp. 22, 24).

[31] Texas Instruments. *Digital Signal Processors*. May 2014. URL: http://www.ti.com/lsds/ti/dsp/overview.page (cit. on p. 23).

[32] NVIDIA Corporation. *Tegra*. May 2014. URL: http://www.nvidia.com/object/tegra.html (cit. on p. 23).

[33] Sixto Ortiz Jr. "Chipmakers ARM for Battle in Traditional Computing Market." In: *Computer, 44(4):14-17* (Apr. 2011) (cit. on p. 23).

[34] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. Feb. 2009, pp. 427–436 (cit. on p. 24).

[35] Martin J. Chorley and David W. Walker. "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters". In: *Journal of Computational Science* 1.3 (2010), pp. 168–174 (cit. on p. 24).

[36] OpenACC Corporation. *OpenACC*. Nov. 2012. URL: http://www.openacc-standard.org/ (cit. on p. 25).

[37] Romain Dolbeau, Stéphane Bihan, and François Bodin. "HMPP: A hybrid multi-core parallel programming environment". In: *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*. Citeseer. 2007 (cit. on p. 25).

[38] Caps Enterprise. *CAPS: The Many-Core Programming Company*. June 2014. URL: http://www.caps-entreprise.com/ (cit. on p. 25).

[39] PathScale. *ENZO 2014*. June 2014. URL: http://www.pathscale.com/ENZO (cit. on p. 26).

[40] Cédric Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurr. Comput. : Pract. Exper.* 23.2 (Feb. 2011), pp. 187–198. ISSN: 1532-0626 (cit. on p. 26).

[41] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing". In: *IEEE Trans. Parallel Distrib. Syst.* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219 (cit. on p. 27).

[42] Artur Mariano. "Scheduling (ir)regular applications on heterogeneous platforms". MA thesis. University of Minho, Sept. 2012 (cit. on p. 27).

[43] Michael Bauer et al. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. IEEE Computer Society Press, 2012, 66:1–66:11 (cit. on p. 27).

[44] Makoto Matsumoto and Mutsuo Saito. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". In: *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation* (1998) (cit. on p. 30).

[45] F. Rademakers and P. Canal and B. Bellenot and O. Couet and A. Naumann and G. Ganis and L. Moneta and V. Vasilev and A. Gheata and P. Russo and R. Brun. *ROOT*. June 2014. URL: http://root.cern.ch/drupal/ (cit. on p. 30).

[46] S. Browne et al. "PAPI: A Portable Interface to Hardware Performance Counters". In: *Proceedings of Department of Defense HPCMP Users Group Conference* (June 1999) (cit. on p. 31).

[47] NVIDIA. *NVIDIA CUPTI User's Guide*. Tech. rep. Feb. 2014 (cit. on p. 32).

[48] Sameer S. Shende and Allen D. Malony. "The Tau Parallel Performance System". In: *Int. J. High Perform. Comput. Appl.* 20.2 (May 2006), pp. 287–311 (cit. on p. 32).

[49] L. Adhianto et al. "HPCTOOLKIT: tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010) (cit. on p. 32).

[50] Intel. *Profiling Runtime Generated and Interpreted Code with Intel VTune Amplifier*. Tech. rep. Jan. 2013 (cit. on p. 32).

[51] Andreas Knüpfer et al. "The Vampir Performance Analysis Tool-Set". In: *Tools for High Performance Computing*. Ed. by Michael Resch et al. Springer Berlin Heidelberg, 2008, pp. 139–155 (cit. on p. 32).

[52] NVidia Corporation. *NVIDIA Nsight Visual Studio Edition 3.2 User Guide*. 2014. URL: http://docs.nvidia.com/nsight-visual-studio-edition/Nsight_Visual_Studio_Edition_User_Guide.htm (cit. on p. 33).