MAPi DOCTORAL PROGRAMME IN COMPUTER SCIENCE

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

July 2014

Contents

1 Introduction **3**

1.1 Motivation, Goals, and Scientific Contribution 5

1.1.1 The Top Quark and Higgs Boson Decay 6

1.1.2 Goals and Scientific Contribution 8

2 State of The Art **11**

2.1 Hardware 11

2.1.1 Homogeneous Systems 11

2.1.2 Heterogeneous Systems 13

2.2 Software 17

2.2.1 Shared Memory Environments 18

2.2.2 Distributed Memory Environments 19

2.2.3 Particle Physics Frameworks 22

2.2.4 Profiling Tools and Libraries 23

3 An Unified Efficient Particle Physics Framework **25**

3.1 The LipMiniAnalysis Skeleton Library 26

3.2 The Proposed Framework 28

3.2.1 Usage and Workflow 30

3.2.2 Preliminary Prototypes 33

4 Research Plan **37**

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 fully used by an application may require deep knowledge of the underlying architecture details of both CPUs and hardware accelerators and extensive tuning of each individual application. It is important to understand the resources on a CPU, such as the computing units organisation, benefits and limitations of using multiple cores, and cache memory hierarchy, to avoid underusing the full computational potential of the device. The architecture design of many-core hardware accelerators have significant differences from device to device with no standard yet defined, unlike current CPUs. The programmer must know the architectural details of each hardware accelerator to produce efficient code. Moreover, the communications among the CPU and hardware accelerators must be managed by the programmer and may significantly affect the efficiency of an application.

From the hardware point of view, efficiency has a different meaning: engineers consider it to be the ratio between power usage and computational throughput. This is a subject of extensive research known as "Green Computing", where the goal is to reduce power consumption of the hardware with little impact to its performance. This is not only important for mobile computing but also to reduce the cost of maintaining huge computing clusters and data centres.

Computing clusters are the most popular High Performance Computing (HPC) platform and are constituted of many computing nodes, with possibly different characteristics, interconnected by specialised communication channels in a distributed memory environment. The computing nodes may be homogeneous or heterogeneous platforms, where the first has only one or more CPUs in a shared memory environment, and the latter has hardware accelerators connected by a PCI-Express interface to the CPUs, in a distributed memory environment. This means that the data is always visible to the CPUs, but must be explicitly transferred to the accelerator devices. The management of the data may affect the performance and efficiency of an application. Code parallelism is a must to take advantage of the multiple cores in both the CPUs and the hardware accelerators, with the programming model differing from shared to distributed memory environments. Data races, resource contention and, in heterogeneous systems, explicit memory transfers are complex issues that the programmer must tackle. Also, each accelerator manufacturer uses their own frameworks and compilers for programming their devices. With the current computational systems rapidly changing, scientists restrain from investing on academic formation in computer science, opting for self-learning these complex principles, and often avoid developing code for hardware accelerators. These factors reinforced the collaboration of multidisciplinary teams of scientists from various fields with computer scientists to develop high performing, efficient and robust applications.

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 experiments, 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 the data, which motivated the creation of the Worldwide LHC Computing Grid [14], a distributed computing infrastructure that uses the resources of 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, usable 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 reprocessing, refining and filtering the relevant information and reducing the size of the data, now in Derived Physics Data (DPD) format, 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 small personal computers, and they perform most of the events reconstruction and final data analysis. In the CERN terminology, an analysis is the denomination of an application which is designed to process a given amount of data in order to extract relevant physics information about events that may support a specific HEP theory.

These factors enforce 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. 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 [15].

The Laboratório de Instrumentação e Física Experimental de Partículas (LIP) [16] 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 organization 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-3 computational resources for data analysis. Most of the analysis applications use in-house developed skeleton libraries, such as the LipCbrAnalysis and LipMiniAnalysis.

1.1 Motivation, Goals, and Scientific Contribution

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. Adding to this increase, 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 more efficient computational resources become ahead of the competition.

Better physics are not only obtained by increasing the amount of events analysed; it is important to take into account the quality of each event analysis. 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 intrinsic uncertainty which is propagated through the event analysis. It is possible to improve the reconstruction quality by varying the values measured by the detector within the said range, but with a significant impact to the analysis execution time, requiring a trade-off between the event processing throughput and their reconstruction quality.

To aid the development of these data analysis applications, scientists at LIP created a skeleton library named LipCbrAnalysis. 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. With this, the programmer only needs to code the specific bits of the analysis filtering and reconstruction of events. An iteration of this skeleton was developed, named LipMiniAnalysis, with the purpose of reading a new structure of the input data files, and stripping the former skeleton of outdated features.

An efficiency study and optimisation of one of LIP production data analysis, used also as case study for some preliminary studies of this PhD pre-thesis work, was presented in [15, 17]. It tackled the computational inefficiencies 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 subsection 1.1.1.

Dealing with scientific code is no trivial task due to the code structure and organization. Several studies [18, 19, 20, 21] identified the causes that lead scientists to produce poor code:

- Most scientists are self-taught programmers with no computer science background;
- Scientists disregard software engineering principals to produce long lasting, extendible, and efficient code;
- Scientists often iterate through 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 usually are not aware of profiling and debugging tools, as well as parallelisation paradigms;
- Scientists do not understand the architectural details of computing systems, reducing the portability of the code they produce.

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

1.1.1 The Top Quark and Higgs Boson Decay

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 top quark and Higgs boson couplings. Figure 1.1 represents the final state topology of the associated production of two top quarks and one Higgs boson (that decays to two b-quarks), labelled from now on as $t\bar{t}H$ production. Figure 1.2 provides a schematic representation of the system to highlight certain features, such as the bottom quarks being jets of particles, and the leptons (both l^+ and l^-) being a muon and electron in the t and $\bar{t}$ decays, respectively.

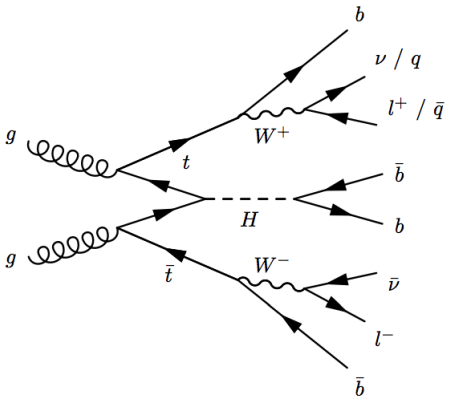

Figure 1.1: Feynman diagram of the $t\bar{t}$ and Higgs boson production.

Neutrinos do not interact with the detector, so their characteristics are not recorded. Since the top quark reconstruction requires the neutrinos, their characteristics are analytically determined with the known information of the system, through a 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.

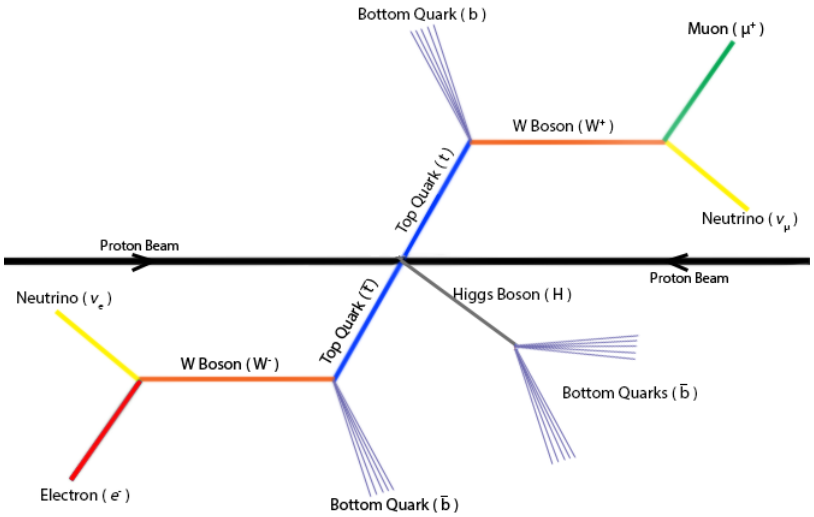

Figure 1.2: Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

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 decays. As shown in figure 1.2, four jets and two leptons are required to be present in the events. Two of the jets, together with two leptons are required 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 of each event is considered. In a first step, the $t\bar{t}$ system reconstruction is tried. If it 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 which were associated to the best $t\bar{t}$ system reconstruction. The overall quality of the event processing depends on the quality 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 reconstructions, 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. The quality of the reconstructions and the application execution time is directly proportional to the amount of variations performed per combination. The goal is to do as many variations as possible within a reasonable time frame.

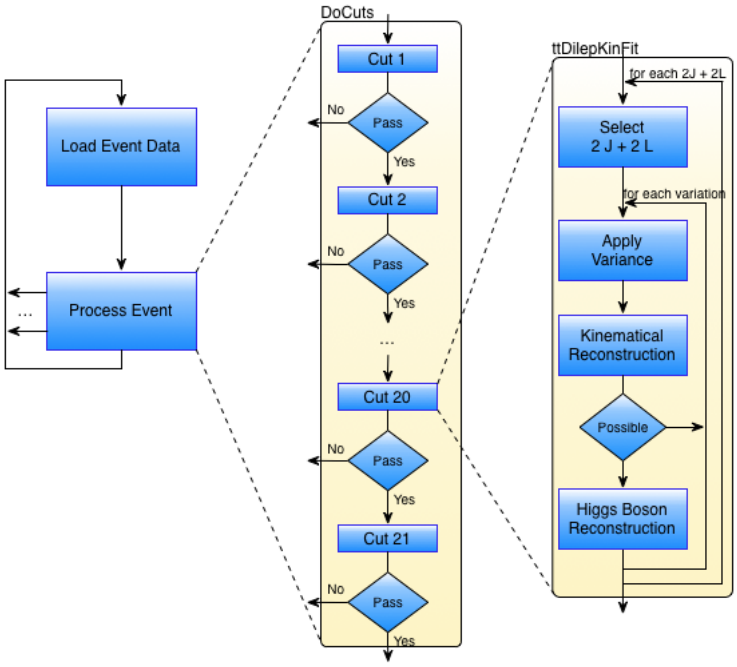

Figure 1.3: Schematic representation for the `ttH_dilep` application flow.

To reconstruct the $t\bar{t}H$ system a data analysis application was developed, the `ttH_dilep`. The application flow is presented in figure 1.3. Each event data on an input file is individually loaded into a single global state, shared between the data analysis code and the LipMiniAnalysis, and it is overwritten every time a new event is loaded. The event is then submitted to a series of cuts, which filters events that are not suited for reconstruction. When an event reaches the cut 20, the $t\bar{t}$ system and Higgs boson are reconstructed in the function `ttDilepKinFit`, which is expected to be the most computing demanding. If the $t\bar{t}$ system reconstruction fails, the current 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.

1.1.2 Goals and Scientific Contribution

The goal for this PhD dissertation is to provide an efficient unified framework for the development particle physics data analysis applications. It aims to give a kernel-like programming model, where 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 parallelised and uses the computing power of both CPUs and accelerator devices.

With a more agile development of high performance data analysis applications, researchers can spend more time improving the algorithms accuracy, which usually requires the extra computing power provided by the efficient usage of multicore CPUs and manycore devices, and analyse a

larger amounts of data. These two factors have a big 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 parallelisation mechanisms, for both homogeneous and heterogeneous platforms, than general purpose frameworks. On homogeneous platforms, it has been demonstrated in [17, 15] 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 parallelisation for both NVidia GPU and Intel Xeon Phi devices, with dynamic load balance among CPU accelerator devices.

Chapter 2

State of The Art

2.1 Hardware

Most scientific research groups have access to computing clusters. These highly parallel systems usually are constituted by racks of computing nodes interconnect by a specialised network, but each with an individual instance of the operating system. The cluster has a distributed memory configuration, where the data must be explicitly transferred among nodes. The nodes may have different characteristics and configurations as long as they use the same interface to communicate with the others.

Cluster often nodes dedicated to centralise the data storage, with an abstraction layer to the user. However, when running an application, the user file system is mounted on the nodes that will perform the computation, copying all the data needed to avoid unnecessary communication. These computing nodes may be homogeneous or heterogeneous systems.

2.1.1 Homogeneous Systems

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

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 transistors. Software developers did not spend much effort optimising their applications and only relied on the hardware improvements to make them faster.

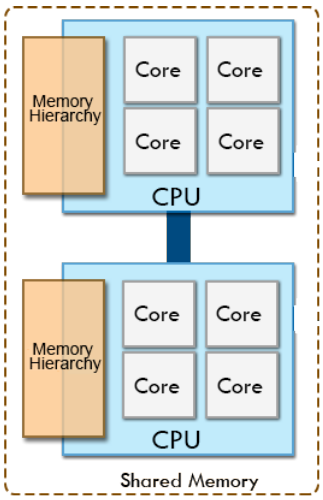

Figure 2.1: Schematic representation of a homogeneous system.

Due to thermal dissipation issues, the clock frequencies of CPU chips started to stall in 2005. Manufacturers shifted from making CPUs faster to increasing their throughput by adding more 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 with a very fast hierarchical memory attached (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 to complex branching and SIMD (single instruction multiple data, later explained) instructions. A single CPU core implements various mechanisms for improving the performance of applications, at the hardware level, with the most important explained next:

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 it is denominated as static parallelism, where compilers try to identify which instructions are independent, meaning that the outcome of one does not affect the execution of the other, and schedules them to be executed at the same time, if the hardware has resources to do so. At the hardware level, ILP can be referred as dynamic parallelism as the hardware dynamically identifies which instructions execution can be overlapped while the application is running.

Vector instructions are a special set of instructions based on the SIMD model, where a single instruction is simultaneously applied to a set of data. CPU instruction sets offer special registers and instructions that allow to execute a operation on a chunk of data in a special arithmetic unit. One of the most common examples is addition of two vectors. The hardware is capable of adding a given number of elements of the vectors . This optimisation is often done 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 between them and lead to a better cache usage.

2.1.2 Heterogeneous Systems

With the emerging use of hardware specifically designed for some computing domains, denominated hardware accelerators, whose purpose is to efficiently solve a given problem, a new type of computing platform is becoming increasingly popular. This marked the beginning of heterogeneous systems, where one or more CPU devices, operating in a shared memory environment similar to homogeneous systems presented in subsection 2.1.1, are coupled with one or more hardware accelerators. In current heterogeneous systems, CPUs and accelerators operate in a distributed memory model, meaning that data must be explicitly passed from the CPU to the accelerator, and vice-versa.

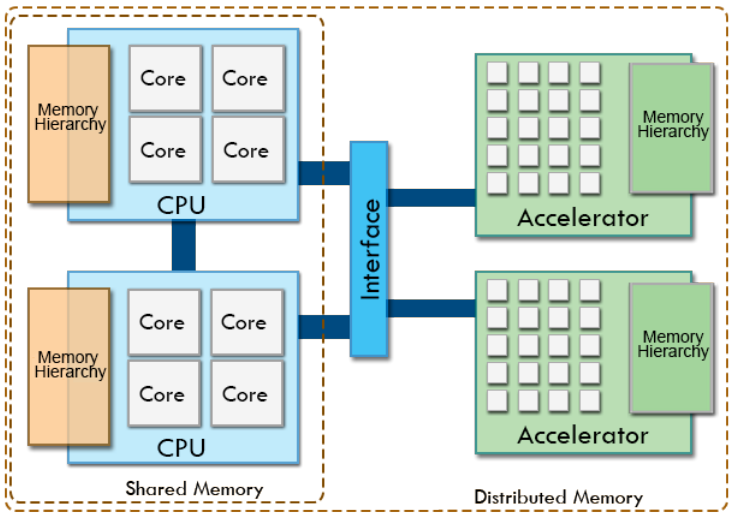

Figure 2.2: Schematic representation of a heterogeneous system.

Figure 2.2 presents a schematic representation of a heterogeneous system. Note that both CPUs must use the same interface to communicate with the hardware accelerators, which may cause contention. This high latency interface, PCI-Express being the most common, usually is a potential bottleneck for applications that use accelerators.

Computing accelerators are usually built with a large number of small and simple processing units, aimed to achieve the most performance possible on specific problem domains, as opposed to general purpose CPUs. They are oriented for massive data parallelism processing (SIMD architectures), where a single operation is performed on huge quantities of independent data, with the purpose of offloading the CPU from such data intensive operations. Several manycore accelerator devices are available, with me most popular being the general purpose GPUs to the Intel Many Integrated Core line, currently known as Intel Xeon Phi [23]. An heterogeneous platform may have one or more accelerator devices of the same or different architectures.

As of November 2013, over 50 of the TOP500's list [24] are computing systems with hardware accelerators, which indicates an exponential growth in usage compared to previous years. The Intel Xeon Phi is becoming increasingly popular, being the accelerator device of choice in 13 clusters of the TOP500, with one of them being the Tianhe-2 which is the faster system on the list. NVidia GPUs remain as the most used accelerator, with the AMD steadily losing their share.

Graphics Processing Unit

The Graphics Processing Units (GPU) were one of the first hardware accelerators on the market. Their purpose is to accelerate 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 model. Its original purpose was to process and synthesise images, which are, from the computation point of view, a large set of numbers representing pixels. The processing of each pixel usually does not depend on the processing of its neighbours, or any other pixel on the image, so the computation has no data dependencies in the best case scenario. This allows to process all pixels simultaneously. The massive data parallelism is the most important characteristic that was considered when designing the GPU architecture.

As the GPU manufacturers allowed more flexibility to program their devices, the High Performance Computing (HPC) community started to use them to solve specific massively data parallel problems, such as numerical computation problems. However, the highly specialised architecture of GPUs affects the performance of many other 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 number one GPU manufacturer for scientific computing GPUs, with a wide range of available hardware known as Tesla. These devices characteristics differ from the general purpose 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 caches. The latest architecture released by NVidia is named Kepler [25], and its relevant architecture details are explained next.

As figure 2.3 shows, the Kepler architecture is organised into two main components: the Streaming Multiprocessor (SMX) and the memory module. The focus of this architecture was not only the performance but the energy efficiency, offering up to to 3x more performance per watt than Fermi (the previous architecture). In addition, the Kepler has implemented several features to improve the usage of resources:

Dynamic Parallelism: a kernel (algorithm coded to run on the GPU) on the GPU is capable of being called recursively, which allows to dynamically generate new load to process, without the CPU interfering. This allows for less regular algorithms to run on the GPU and reduces the communication between CPU and GPU as it is capable of managing the workload.

Hyper-Q: this system increases the amount of work queues to 32 simultaneously hardware managed connections. With this, multiple CPU cores can launch new different kernels on the CPU at the same time, increasing the resource usage. Now, multiple threads of the same

Figure 2.3: Schematic representation of the NVidia Kepler architecture.

application are not required to have exclusive usage of the GPU, reducing the amount of synchronisations.

Grid Management Unit: to allow for dynamic parallelism a new grid (a collection of threads of a kernel, later explained in more detail) management system is required. The new system also allows to schedule multiple grids, which allows for different kernels, from possibly different threads, to run simultaneously (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, and reducing the communication latency.

The SMX are the units responsible for performing all computations on the GPU, and a chip may have up to 15. 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 explained next) and 8 instruction dispatchers.

Memory wise, each SMX has 65536 32-bit registers, with a maximum of 255 registers per CUDA thread, a 64 KByte very fast memory for L1 cache and shared memory, and a similar fast 48 KByte memory cache for read-only data. Finally, the Kepler architecture provides 1536 KB 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 it greatly restricts 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 known as Intel Xeon Phi, is an emerging technology becoming 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. 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 must be explicitly passed between CPU and device as required by the code that it will compute. All library functions called inside the device must be specifically compiled for it. Note that it is not possible to have an entire library compiled for the Xeon Phi and CPU simultaneously.

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) [26]. The restrictions mentioned in the previous point are also applicable.

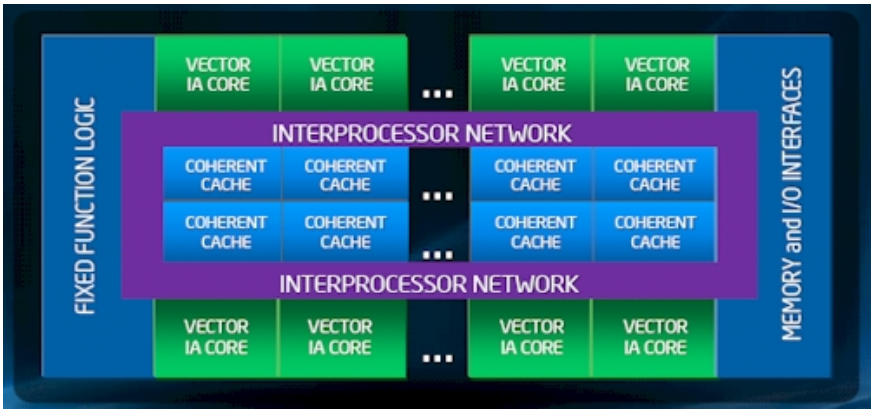

Figure 2.4: Schematic representation of the Intel Many Integrated Core architecture.

Each core is able to run 4 threads simultaneously, and most of the massive parallelism is obtained by he 32 512 bit wide vector registers available. The core as 64 KB for data and 64 KB

for instruction 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 1.33 GHz, which is less than half of the modern 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 [15].

Other hardware accelerators

More hardware accelerators are coming to 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 [27].

ARM processors are now leading the mobile industry and, alongside the new NVidia Tegra processors [28] 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 [29]. 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 Software

Most programmers (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 greater when programming for heterogeneous systems, where programming paradigms shift when considering different hardware accelerators. The mainstream industry is still adopting the use of multicore architectures with the purpose of increasing their processing power, reflecting in a lack in the 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 ensure efficient use of the computational resources, but most importantly the correctness of the application.

Heterogeneous systems combine the flexibility of multicore CPUs with the specific capabilities of manycore accelerator devices. However, most computational algorithms and applications are designed with the specific characteristics of CPUs in mind. Even multithreaded applications

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

usually cannot be easily ported to these devices expecting high performance. To optimise the code for these devices it is necessary a deep understanding of the architectural principles behind their design.

The workload balance between the cores of a single CPU chip is an important aspect to extract performance and get the most efficient usage of the available resources. A 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 the cores are used as most as possible. Considering a multi-CPU system, it is important to manage the data in such a way that it is available in the memory bank of the CPU that will need it to avoid the NUMA latency. 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 frameworks to aid the development of parallel applications for homogeneous and heterogeneous systems.

2.2.1 Shared Memory Environments

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 parallelising for 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 forces 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 [30], Intel Threading Building Blocks (TBB) [31], and Cilk [32] 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 all available CPU architectures. 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.

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 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.

2.2.2 Distributed Memory Environments

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

Message Passing Interface

The Message Passing Interface (MPI) [26], designed by a consortium of both academic and industry researchers, has the objective of providing a simple API for process based parallel programming in distributed memory environments. It relies on point-to-point and group messaging communication, and is available in Fortran and C. It is often used in conjunction with a shared memory parallel programming API, such as OpenMP, for work sharing between computing nodes, with the latter ensuring a more efficient parallelisation inside each node.

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

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 for programming devices similar architecture to the NVidia GPUs. It has a specific instruction set architecture (ISA) and allows programmers to use GPUs for 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 (conditional jumps are a special case that will be later explained) but on different data. Considering the sum of two vectors, each CUDA thread will be responsible for adding a single element of the vectors.

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 matched by the global scheduler to run on a specific SMX. A grid is a set of blocks, representing 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.

Conditional jumps must be avoided as they may cause different CUDA threads within the same warp execute different instructions. Since inside an SMX it is not possible to have 2 threads simultaneously executing different instructions, the divergent branches will be executed sequentially, doubling the warp execution time.

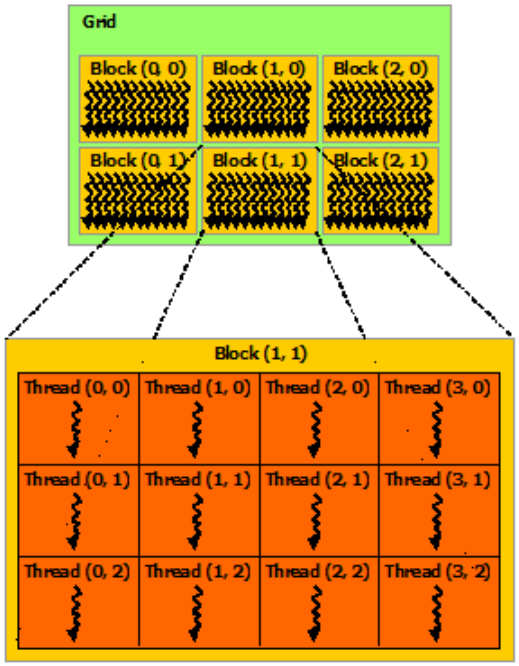

Figure 2.5: Schematic representation of CUDA thread hierarchy.

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 portable and scalable parallel applications. Its main focus is to obtain the best performance possible on irregular applications, rather than abstracting all the architecture details from the programmer. The programmer still needs to have some knowledge of each different architecture and programming paradigms, and he needs to instruct the framework about how tasks should be divided in order to fit the requirements of the different devices.

Instead of relying in pre-partitioned work, 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. Apart from this, the framework frees the programmer from managing the workload distribution, memory usage and data transfers between the available devices, but requires that the application is built according to the strict specifications of the framework. The programmer is able to tune specific details related to the memory transfers and load balance, if he is comfortable enough 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 [33], which uses the computation and communication costs of each task, in order to assign every task to a device in such a way that minimizes 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 [34].

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 between 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.

StarPU

StarPU [35] is a unified runtime system consisting on both compiler directives and a runtime API that aims to allow programmers to efficiently extract parallelism from 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.

However, one of the main differences to DICE is that 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, asynchronously prepare data dependencies, while the system is busy computing other tasks. The task scheduler itself can take this into account, and determine where a task should be executed by taking into account 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 from the schedulers implemented in StarPU, but most resort to a history of the tasks execution 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 a not efficient 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, as he must declare if a data structure is read/write or both. The granularity of the tasks must be defined by the user, as opposed to the DICE dynamic adjustment.

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. The current specification addresses both NVidia and AMD GPUs, as well as the Intel Xeon Phi.

However, this framework offers more of 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, developed 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 parallelisation. 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 these 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].

2.2.3 Particle Physics Frameworks

ROOT

ROOT [40] is a very complex framework coded by 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, with a master/slave process hierarchy, without support for hardware accelerators. Also, it does not focus on the efficient usage of the available computational resources, only distributes the load on demand among the processes.

Currently, ROOT does not provide parallel functionalities, specially for shared memory environments, but the developers already shown interest to improve the performance of some core routines of the framework. However, it may not translate in massive performance gains of the data analysis applications, as the critical region is usually on the reconstruction of the events, which does not heavily rely on a small set of ROOT functionalities, but rather on a large set of non-complex routines and classes.

TopROOTCore

TopROOTCore is an extension of ROOT for top quark physics, developed by CERN associate research groups,that adds more functionalities 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.

2.2.4 Profiling Tools and Libraries

Performance API

The Performance API (PAPI) [41] 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, trough the NVidia CUPTI 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) [42] is a performance analysis interface available through 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 [43] and HPCToolkit [44] 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 more general metrics and 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 both shared and distributed memory environments, but the HPCToolkit still does not support any hardware accelerator. 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 [45] is a proprietary tool for performance analysis of applications. It provides an easy to use interface to analyse applications, automatically identifying its bottlenecks, without 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 [46] is a open source library to analyse an application execution on both shared memory and distributed memory environments, with support for CUDA capable GPUs through the CUPTI 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 to other similar tools, 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 [47] is a developer more of a platform rather than a framework 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), SMX 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

Programming for homogeneous platforms poses a series of challenges not faced when coding parallel applications due to the shared memory paradigm. In this context, the data is always easily accessible when coding, as the different memory bank accesses are managed by the compiler and hardware. Data dependencies and races still need to be managed by the programmer, which may required a significant level of expertise. To efficiently use the computational resources, dealing with problems such as false sharing or efficient cache usage, the programmer must have an advanced expertise on both the coding and architectural details of homogeneous platforms.

Since an heterogeneous platform is a distributed memory environment, where the CPUs shared the memory with each other but not with the hardware accelerators, a series of new challenges arise. All communications of data between CPU and accelerator must be explicitly coded by the programmer, and has an added latency associated. The balance of the work 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 show in section 2.1, which constrain the way they are programmed 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, adapting current applications to run on heterogeneous platforms may be infeasible without redesigning all major algorithms, as opposed to code a new application specifically for these platforms. This issue has an higher impact on legacy code.

Scientists are usually self-taught programmers that only consider coding as a necessary tool to perform their research. Several studies, referred in section 1.1, identified a set of problems with scientists coding practices and scientific computing. Most of their code is in constant development, up to decades long, only adding or changing functionalities in each iteration, not considering any software engineering principles and not adapting the code to the changes in hardware. The few that worry about performance attempt to address the code regions that they think are the bottleneck, not knowing of the existence of profiling tools and even compiler optimisations.

Since most scientists develop applications with the help of specialised frameworks of their research field, they expect them to be efficient, by resorting to parallelisation or other techniques. However, the bottleneck is often on the scientists code rather than in the framework, and these tools are not designed to automatically extract parallelism from their code.

Scientists usually do not have any training for programming efficient applications for homogeneous systems or cluster environments, as programming is just a necessity for their field of research. They are even more reluctant to learn the new programming paradigms required to work with hardware accelerators on heterogeneous platforms. With this in mind, several automatically parallelisation frameworks for these systems were developed by computer scientists, 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.

Even though, scientists are not willing to endure the steep learning curve of these frameworks to integrate with future applications, and do not want to code two versions of the core algorithms to run on the CPU and accelerator. Their complexity and the lack of guarantees to that they will increase the code performance, due to poor implementation or algorithm characteristics, puts the scientists further away from these frameworks. Also, they attempt to have few dependencies of an application with external libraries, as the external tools are not guaranteed to be supported through the application lifetime.

The existence of general purpose automatically parallelisation frameworks 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 to interact with any scientific field, but are more adapted to the scientific problem that is addressed. A lower abstraction level leads to an easy interaction of the scientist with the tool (and even abstract them of any parallelisation complexities) and increases the computational efficiency of the code when compared with general purpose frameworks, as the main bottlenecks are usually known *a priori* and the framework is designed around their characteristics. The development of such frameworks may lead to a better interface of computer scientists and researchers, causing and improvement of their codes with the implementation of software engineering concept, increasing 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, transforming it from the format supported by ROOT at the time to variables in the global memory of the skeleton. All other standard functions needed for the analysis had their prototypes declared, but it was the programmers job to code them to his needs, knowing that the required data for each event was on memory.

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 for passing options and arguments to the executable. Some of the functions that the programmer needed to code were now fully implemented, with the option of being override by the user.

The LipMiniAnalysis is the latest development iteration on a production environment. It discarded outdated features that were no longer necessary, and reads the new Mini Ntuple data format, hence its name. It is not a standard ROOT data format, but rather a set of events that suffered some preprocessing to improve the quality of the data and filter events that do not show any interesting physics.

Figure 3.1 presents the structure of LipMiniAnalysis, where all sections marked with a * 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 parameters not set yet. 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, so 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.

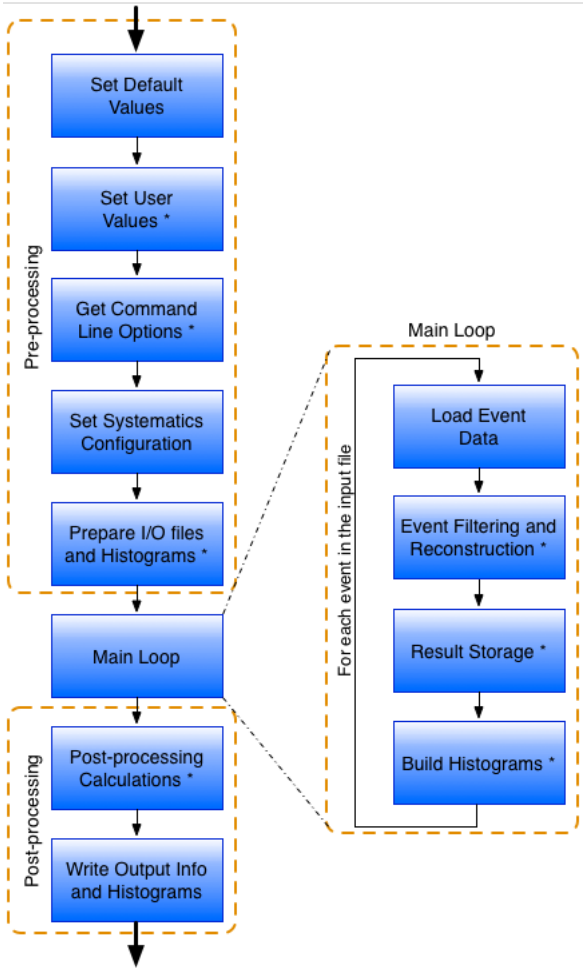

Figure 3.1: Schematic representation of the LipMiniAnalysis skeleton structure. The * mark represents the sections of the skeleton that the programmer must code.

The *Main Loop* is responsible for individually loading an event from the input file, apply the filters, reconstruct it if it passes the required filters, store the results, and build the histograms

for each filter and final reconstruction. Only the event loading is automatic, so the programmer must code all remaining sections, as they vary among different analysis. The final post-processing also depends on the analysis, so it is not coded. 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 structure, code-wise.

Studies presented in [15, 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 was to address only the inefficiencies of the data analysis itself, problems with the main data structure of the skeleton 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, from simple scalars to complex vectors of ROOT classes. With only a single event in memory, it is more difficult to create an efficient parallelisation 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 through the application lifetime. With all events from an input data file on the application global memory, a more efficient parallelisation for both shared and distributed memory systems can be achieved. It would also help the implementation of automatic parallelisation in LipMiniAnalysis.

3.2 The Proposed Framework

A physics framework to replace LipMiniAnalysis in aiding the development of data analysis applications is proposed. 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, and ensure the generation of efficient data analysis applications by automatically parallelising the code. It will include both redesigned features of LipMiniAnalysis and new physics functionalities.

Improving both the researcher 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 LipCbrAnalysis developed in 2005, does not suit the characteristics of current hardware. The new framework will implement all required features present in LipMiniAnalysis but designed to modern specifications of hardware, software, and modularity to interact with new features.

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* and *Histograming* modules will be implemented using redesigned features currently available at LipMiniAnalysis, as they have standard features used by most data analysis applications. The framework will have to support both ROOT and TopROOTCore libraries without any specific configuration by the user. TopROOTCore installation with the framework may be an option to the user, since only a part of event data analysis applications use its functionalities, which are more useful for data preparation.

The *I/O* module is responsible for loading the input data files for processing the events in the data analysis applications. The LipMiniAnalysis reads only MiniNtuples, which are preprocessed events at the previous computational tiers, which are used by some applications, but other file

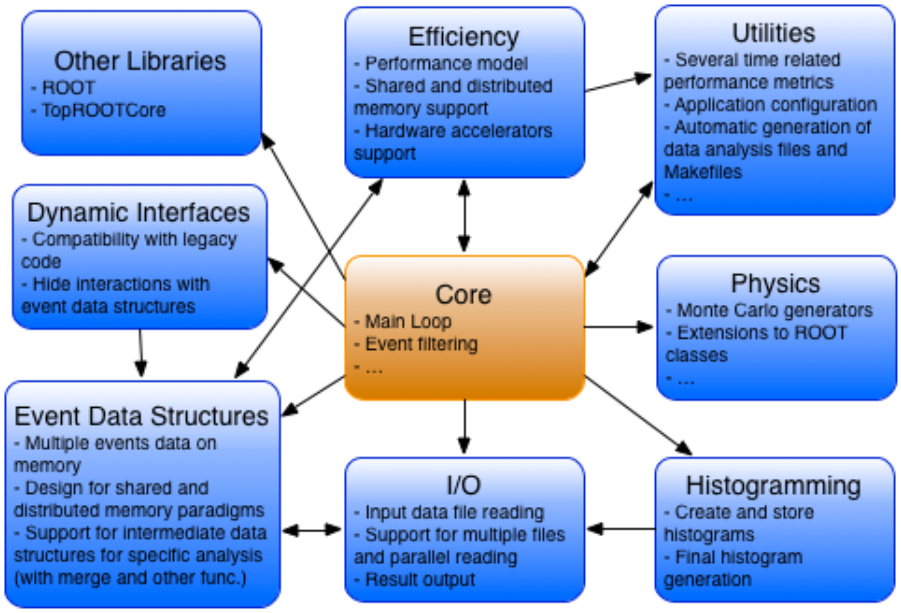

Figure 3.2: Schematic representation of the proposed framework modules and their dependencies.

formats may also be supported, as requested by researchers. Different file formats have different structures and auxiliary event parameters stored, 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.

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. Then, it must hold a STL collection (vector, map, etc) to store the instantiations of the event class, enclosing all events in the input data file. 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 storing collection must be suitable for both shared and distributed memory parallelisation, with good performance for 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 (un)marshalling computations for communicating the data. 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, considering the limitations that they incur.

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 to benefit from the automatic parallelisation 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, later explained in more detail. 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 dynamically generate the interfaces.

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 distribution for both homogeneous and heterogeneous platforms with hardware accelerators. The performance model must be capable of assessing if an hybrid multithreaded process parallelisation provides better efficiency than a simple implementation, which was proven to happen in some data analysis [17]. There is no automatic parallelisation 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) and uses an efficient amount of parallel tasks, but it may prove beneficial to have this mix of processes and threads in some specific cases. This might require an higher overhead to obtain the best process/thread configuration, but since these data analysis run for several hours the initial setup cost is minimum. It also can produce a configuration file when the configuration is performed for a given data analysis on a given computing system, which can avoid performing the initial setup every time the application is executed.

This module must also be able to assess, with some help of the user, what sections of the data analysis code can be executed in the hardware accelerators, and if it has the required characteristics to efficiently use these devices. While this might seem complex for general purpose parallelisation frameworks, dealing with only a specific problem such as physics data analysis applications helps the design of the performance model around these features. Implementation-wise, these features require heavy modifications to current automatic parallelisation frameworks. However, a new implementation designed to this specific problem, with components adapted from current frameworks, may provide a better and more simple integration and performance efficiency of data analysis applications.

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 will have assembled all the specific bits of each module to process the events, as well as the code sections that is for the user to code. Note that the user will not edit the code in the framework, but rather on the data analysis source code that will later link the missing parts into the framework.

Features that are very important for the most compute intensive sections of data analysis applications will be implemented to run on CPU and accelerator devices. These features 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 uses 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, with as minimum interaction as possible. This eases the development of new data analysis applications and increases their portability. Once the framework is installed on a system, the user just needs to copy their code and it is expected to compile, as all external dependencies are handled by the framework. 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.

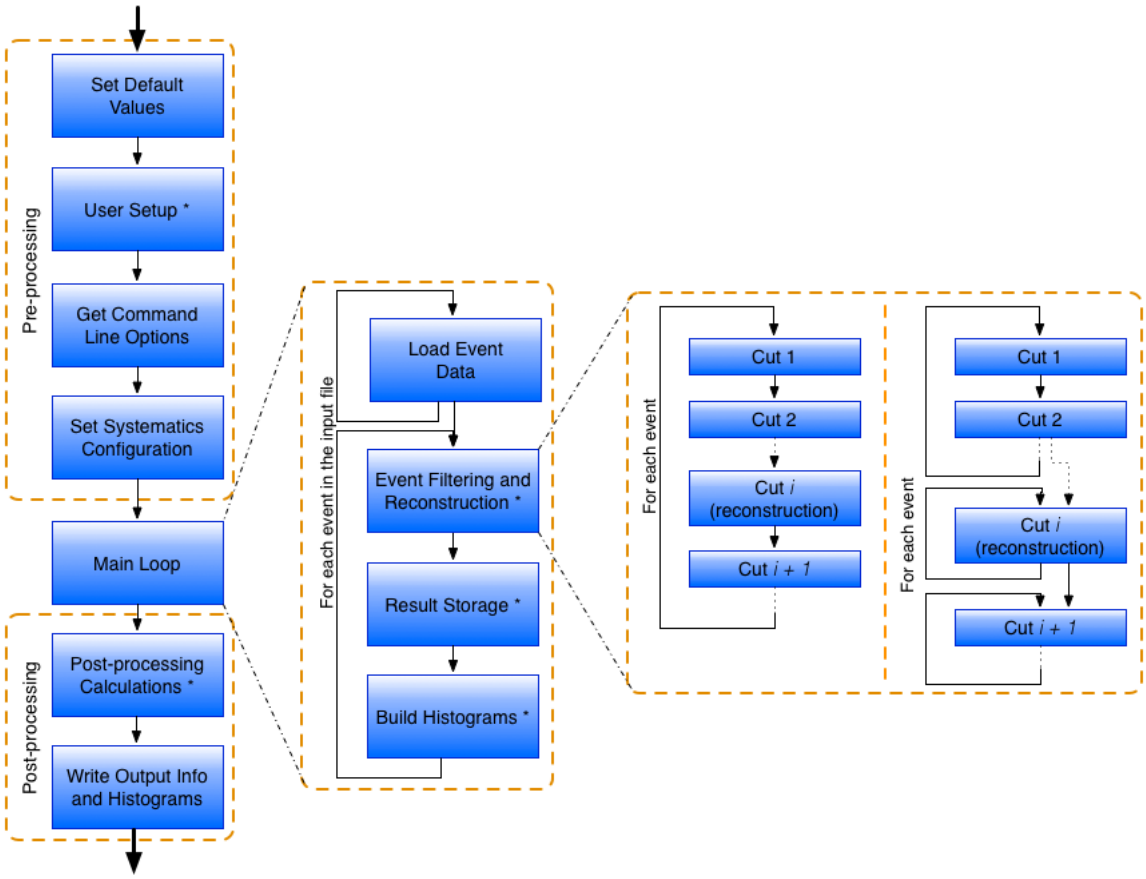

Figure 3.3: Schematic representation of the proposed framework flow. The * mark represents the sections of the workflow coded by the programmer.

The user intervention is required at four stages of the event processing (implementation-wise, 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 at least a set of predefined methods that represent each of the stages that the user must code. The coding is very intuitive, as the user must implement the set of features required to process a single event, in the logical pipeline reasoning. 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. 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. The user can also easily 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 parallelisation setup.

***Event Filtering and Reconstruction*:** this holds the code required to filter and reconstruct 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 pre-filtering 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 and this provides more parallelism (as it is applied simultaneously to all events) and better load management, specially when using heterogeneous platforms.

***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.

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

However, 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 none data analysis in LIP that depends on other libraries). To run on heterogeneous platforms, the analysis critical region must only depend on features that the framework has implemented for both CPU and accelerator devices. This allow for the user to code the data analysis once and the critical regions are able to execute on any computing device. Otherwise, the code will be restricted to the device that can execute it.

3.2.2 Preliminary Prototypes

Some of the work towards the creation of the new framework was 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 1.1.1, but have a modular design to be properly merged into the final framework, and they may suffer structure or implementation changes throughout its development. Proposed prototypes for several features required by the framework 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 must be 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 on an input file. However, the current LipMiniAnalysis version performs a set of data preparation routines, named *FillAllVectors* and *Calculations*, performed automatically and coded by the user, respectively. 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 changed in order to 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. Also, since *Calculations* only access 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 easily coed by the user in the analysis source file, without the need of modifying the LipMiniAnalysis code and recompile the skeleton.

A new *Main Loop* design

Having a new data structure that allows multiple events to be on memory simultaneously 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 (note that an input file has a size of 1 GBytes), as presented in figure 3.4. Now, it is possible to parallelise the execution of the *Event Filtering and Reconstruction*, performed in the *DoCuts* function. In physics terminology, a filter is addressed as a cut.

The interaction of the LipMiniAnalysis with the input data file is performed using the ROOT file reader classes. Since the event loading assumes also 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 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.

As previously stated, the *DoCuts* function is responsible for filtering and reconstructing the events. The purpose of these filters is to separate the signal (interesting events) from the background (uninteresting events). The notion of an interesting event depends on the physics that the analysis studies, for example the background of a $t\bar{t}H$ system analysis may contain events useful

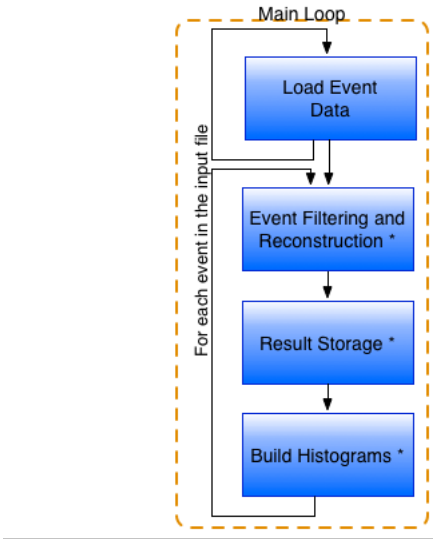

Figure 3.4: Schematic representation of the *Main Loop* implementation modifications.

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 not submitted to the rest of the filters. 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, discarding events that do not provide interesting physics.

The reconstruction of an event is considered as a filter in *DoCuts*, as only the events capable of being reconstructed pass the filter, which is usually the last. The reconstruction may be the most complex and computational intensive task in the whole data analysis application. One example is the $t\bar{t}H$ system reconstruction, which performance depends on a trade-off between reconstruction accuracy and execution time. The event reconstruction is a task that may require a closer look to improve the efficiency of the data analysis, through a more specialised parallelisation on both homogeneous and heterogeneous platforms. Its irregular load and execution time, very complex algorithms, and few vectorised numerical computations creates a bottleneck complex to deal with, but with the potential of bringing exceptional performance gains when addressed with detail [17]. Figure 3.5 presents a implemented prototype for a new *DoCuts* flow that exposes the event reconstruction to more efficient parallelism approaches.

Besides exposing the parallelism of a complex computational intensity section, this change also increases the workload to be simultaneously processed. When the application reaches the reconstruction, all events that passed the previous cuts can be processed in parallel. Having all the data necessary outside of the *Main Loop*, as the *Event Filtering and Reconstruction* as its own loop over all events on memory, but still hiding that fact from the user that codes *DoCuts* to abstract the user as explained in 3.2, allows for a better load balance of the possible parallelisation approaches, for homogeneous and heterogeneous platforms. For the latter, which operate in a distributed memory environment, it also helps to reduce the parallelisation overhead associated with the communications: instead of passing required the data of a single event multiple times, the data of all events is passed once. Then 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 current data analysis applications.

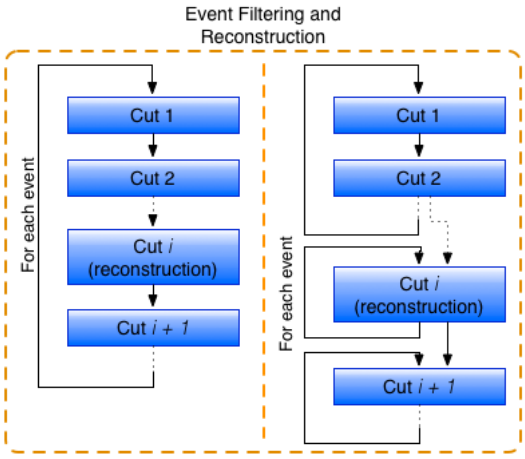

Figure 3.5: Schematic representation current (left image) and proposed (right image) *Event Filtering and Reconstruction* flows.

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, a way to abstract the user from programming for a set of events by providing a kernel-like approach (where the code for processing an event is automatically applied to all). The interface must abstract both the access to the *EventData* variables and methods, while being completely transparent to the user.

The input data files occasionally suffer some changes do 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. To avoid this problem, a parser was developed that receives the *EventData* source files and retrieve the name of the methods and parameters. It then creates an header composed of *define* clauses that translate the accesses to these variables on the STL collection to the previous simple accesses. This header is then included in the main LipMiniAnalysis header so that the user does not need to interact with the interface. This interface is automatically created every time the skeleton is compiled.

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. Note that the counter that assigns the event to process is automatically managed in every loop that iterates through the event data structure, hidden from the user.

Other features

The final framework will have an utilities module where general features will be coded. 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, set 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 implemented is specific 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.

Chapter 4

Research Plan

levantamento de requisitos

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. 4).

[2] European Organization for Nuclear Research. *The Proton Synchrotron*. July 2013. URL: http://home.web.cern.ch/about/accelerators/proton-synchrotron (cit. on p. 4).

[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. 4).

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

[5] European Organization for Nuclear Research. *ATLAS experiment*. Nov. 2012. URL: http://atlas.ch/ (cit. on p. 4).

[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. 4).

[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. 4).

[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. 4).

[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. 4).

[10] European Organization for Nuclear Research. *A Large Ion Collider Experiment*. Nov. 2012. URL: http://aliceinfo.cern.ch/ (cit. on p. 4).

[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. 4).

[12] European Organization for Nuclear Research. *Computing*. July 2013. URL: http://home.web.cern.ch/about/computing (cit. on p. 4).

[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. 4).

[14] European Organization for Nuclear Research. *The Worldwide LHC Computing Grid*. July 2013. URL: http://wlcg.web.cern.ch/ (cit. on p. 4).

[15] André Pereira. "Efficient Processing of ATLAS Events Analysis in Homogeneous and Heterogeneous Platforms". MA thesis. University of Minho, Sept. 2013 (cit. on pp. 5, 6, 9, 17, 28).

[16] 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. 5).

[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. 6, 9, 28, 30, 34).

[18] Zeeya Merali. "ERROR... Why scientific code does not compute". In: *Nature, pp. 775-777* (467 Oct. 2010) (cit. on p. 6).

[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. 6).

[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. 6).

[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. 6).

[22] Gordon E. Moore. "Cramming more components onto integrated circuits." In: *Electronics, 38(8)* (Apr. 1965) (cit. on p. 11).

[23] Intel. *The Intel® Xeon Phi Datasheet*. Tech. rep. Apr. 2014 (cit. on p. 13).

[24] TOP 500. *November 2013*. May 2014. URL: http://www.top500.org/lists/2013/11/ (cit. on p. 14).

[25] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Tech. rep. 2012 (cit. on p. 14).

[26] 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. 16, 19).

[27] Texas Instruments. *Digital Signal Processors*. May 2014. URL: http://www.ti.com/lsds/ti/dsp/overview.page (cit. on p. 17).

[28] NVIDIA Corporation. *Tegra*. May 2014. URL: http://www.nvidia.com/object/tegra.html (cit. on p. 17).

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

[30] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Tech. rep. July 2013 (cit. on p. 18).

[31] James Reinders. *Intel Threading Building Blocks*. Tech. rep. 2007 (cit. on p. 18).

[32] 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. 18).

[33] 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. 20).

[34] Artur Mariano. "Scheduling (ir)regular applications on heterogeneous platforms". MA thesis. University of Minho, Sept. 2012 (cit. on p. 20).

[35] 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. 21).

[36] OpenACC Corporation. *OpenACC*. Nov. 2012. URL: http://www.openacc-standard.org/ (cit. on p. 21).

[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. 22).

[38] Caps Enterprise. *CAPS: The Many-Core Programming Company*. June 2014. URL: http://www.caps-entreprise.com/ (cit. on p. 22).

[39] PathScale. *ENZO 2014*. June 2014. URL: http://www.pathscale.com/ENZO (cit. on p. 22).

[40] 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. 22).

[41] 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. 23).

[42] NVIDIA. *NVIDIA CUPTI User's Guide*. Tech. rep. Feb. 2014 (cit. on p. 23).

[43] 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. 23).

[44] 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. 23).

[45] Intel. *Profiling Runtime Generated and Interpreted Code with Intel VTune Amplifier*. Tech. rep. Jan. 2013 (cit. on p. 23).

[46] 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. 24).

[47] 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. 24).