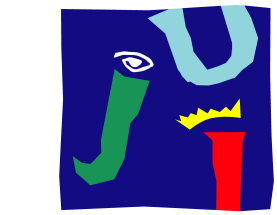


UNIVERSITAT JAUME·I

DOCTORAL THESIS

Multi-level Parallelization in ROOT: New Patterns, Libraries and Utilities

Author:
Xavier Valls Pla

Supervisor:
Dr. Enrique S. Quintana Ortí

A thesis submitted in fulfillment of the requirements for the Doctoral Programme in Computer Science

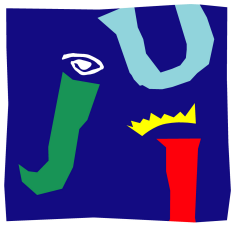
in the

Doctoral School
Universitat Jaume I

developed in collaboration with the

European Organization for Nuclear Research (CERN)

July 2018



UNIVERSITAT JAUME·I

Programa de Doctorat en Informàtica / Programa de Doctorado en Informática

Escola de Doctorat de la Universitat Jaume I / Escuela de Doctorado de la Universitat Jaume I

Multi-level Parallelization in ROOT: New Patterns, Libraries and Utilities

Memòria presentada per Xavier Valls Pla per a optar al grau de doctor per la Universitat Jaume I
Memoria presentada por Xavier Valls Pla para optar al grado de doctor por la Universitat Jaume I

Autor:
Xavier Valls Pla

Director:
Enrique S. Quintana Ortí

Castelló de la Plana, Juliol de 2018 / Castellón de la Plana, Julio de 2018

Finançament rebut
Agències finançadores del doctorand:

- Al CERN l'ajuda concedida per a la realització d'una estada predoctoral al CERN que ha possibilitat la sol.licitud de la "Menció de Doctorat Internacional" per a la present tesi doctoral.

Agències finançadores del projecte de recerca o dels recursos materials específics del grup de recerca.

- CERN.

Financiación recibida
Agencias financiadoras del doctorando:

- Al CERN la ayuda concedida para la realización de una estancia predoctoral en el CERN que ha posibilitado la solicitud de la "Mención de Doctorado Internacional" para la presente tesis doctoral.

Agencias financiadoras del proyecto de investigación o de los recursos materiales específicos del grupo de investigación:

- CERN.

Acknowledgements

This thesis wouldn't have been possible without the supervision at CERN of Dr. Lorenzo Moneta, whose deep knowledge, kindness and patience helped me reach new heights. Bureaucracy unfairly stripped him of official recognition, but let this serve as an acknowledgment that he was the guiding light. Thank you.

I would like to express my gratitude towards the ROOT team, who nurtured me during the last three years. Special thanks to the performance group, instrumental in the development of this thesis: Guilherme, Enrico, Enric, Philippe, Axel, Pere, Gerri and most of all, Danilo, whose high standards and honest feedback compelled me to grow this much.

Thanks, Megan, for your company, patience, corrections, teachings, and understanding. Thanks for this peace of mind. Thanks for all the love.

Finally, I would like to thank my family, to whom this thesis is dedicated. Thanks for the privilege and the opportunity to be here today. Love you.

Abstract

The Large Hadron Collider (LHC) is generating Physics data at an unprecedented rate that is expected to continue increasing in the future. This situation results in increasing demands in computation and processing power for analyzing the LHC data. At the same time, modern architectures have switched from a sequential processing paradigm to a parallel one.

In order to take full advantage of new computer architectures, to improve performance with increasing amounts of data to analyze, and to reduce development time and complexity, ROOT, the official LHC analysis framework, is undertaking a modernization effort to cope with the computational challenges posed by the ambitious physics program of the LHC.

We contribute to this modernization effort by implementing patterns, introducing libraries and developing utilities for the parallelization of ROOT at multiple levels, improving the programming model and providing backward compatibility.

We contribute to task-level parallelism with the executors, ROOT classes implementing the parallel MapReduce pattern and hiding concurrent programming complexity. We adapt a fork-based multiprocessing executor, develop a TBB-based multithreading executor, and study the impact task granularity on their performance. With TNUMAExecutor, we provide a solution for suboptimal performance in NUMA architectures by combining libnuma and the executors to improve memory accesses. Finally, we design ROOT's centralized thread pool manager, which tackles undesired behaviour arising from a mixed implicit and explicitly multithreaded execution of ROOT.

We introduce the VecCore library in ROOT to exploit data-level parallelism, providing efficient vectorization on a variety of platforms by offering an abstraction layer on top of other SIMD libraries. We thoroughly benchmark its performance with a complex example, the generation of Julia sets. We introduce two new VecCore-based SIMD types in ROOT and we provide an example of their integration by benchmarking the SIMD evaluation of TF1, a fundamental class for mathematical operations in ROOT.

We demonstrate how we can combine parallelism at multiple levels by parallelizing and vectorizing the fitting in ROOT, parallelizing at task-level the objective functions of the fit, and explaining the adaptation of ROOT's fitting classes and interfaces for a safe and efficient evaluation of the fitting on SIMD types.

Finally, we study the performance of some of the most relevant deployments of our work on ROOT's codebase, showcasing the versatility of the executors and the convenience of VecCore.

With this thesis, we demonstrate that we can parallelize an established 20-year old code with a large userbase by choosing our battles, improving its performance and programming model without breaking backward compatibility.

Abstract

El Gran Col·lisionador d'Hadrons (LHC) està generant dades de Física a un ritme sense precedents, que es preveu que continue augmentant en un futur. Aquesta situació es tradueix en demandes creixents en computació i poder de processament per analitzar les dades obtingudes al LHC. Al mateix temps, les arquitectures modernes han evolucionat des d'un paradigma de processament sequencial a un paral·lel.

Per aprofitar al màxim les noves arquitectures de computadors, millorar el rendiment amb quantitats cada vegada majors de dades per analitzar i reduir el temps i la complexitat del desenvolupament, ROOT, el framework d'anàlisi oficial de LHC, està duent a terme un esforç de modernització per fer front als desafiaments computacionals plantejats per l'ambiciós programa de física del LHC.

Contribuïm a aquest esforç de modernització mitjançant la implementació de patrons, la introducció de llibreries i el desenvolupament d'utilitats per la paral·lelització de ROOT en múltiples nivells, millorant el model de programació i proporcionant compatibilitat amb versions anteriors del programari.

Contribuïm al paral·lelisme a nivell de tasca amb els executors, classes de ROOT que implementen el patró MapReduce en paral·lel i oculten la complexitat de la programació concurrent. Adaptem un executor multiprocés basat en fork, desenvolupem un executor multifil basat en TBB i estudiem l'impacte que la granularitat de les tasques té en el seu rendiment. Amb TNUMAExecutor, proporcionem una solució pel rendiment subòptim en arquitectures NUMA, combinant libnuma i els executors per millorar els accessos a la memòria. Finalment, dissenyem el gestor de grups de fils centralitzat de ROOT, que aborda els comportaments no desitjats derivats d'una execució mixta dels modes multifil implícit i explícit de ROOT.

Introduïm la llibreria VecCore a ROOT per explotar el paral·lelisme a nivell de dades, proporcionant una vectorització eficient multiplataforma gràcies a oferir una capa d'abstracció sobre altres llibreries SIMD. Analitzem minuciosament el seu rendiment amb un exemple complexe, la generació de conjunts de Julia. Introduïm dos nous tipus de dades SIMD basats en VecCore en ROOT i proporcionem un exemple de la seva integració analitzant l'avaluació SIMD de TF1, una classe fonamental per a la realització d'operacions matemàtiques amb ROOT.

Demostrem com podem combinar el paral·lelisme en múltiples nivells paral·lelitzant i vectorizant l'ajust de distribucions proba-

bilístiques en ROOT, paral·lelitzant a nivell de tasca les funcions objectiu de l'ajust i explicant l'adaptació de les classes i interfícies d'ajust existents en ROOT per una avaluació segura i eficient de l'ajust en tipus de dades SIMD.

Finalment, estudiem el rendiment d'algunes de les implementacions més rellevants del nostre treball al codi font de ROOT, evidenciant la versatilitat dels executors i la conveniència de VecCore.

Amb aquesta tesi demostrem que podem paral·lelitzar un codi establert de 20 anys de vida amb una gran base d'usuaris triant les nostres batalles, millorant el rendiment i el model de programació, i sense trencar la compatibilitat en versións anteriors del programari.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Large Hadron Collider (hereinafter referred to as the LHC), the world's most powerful particle accelerator, is an essential tool for the study of High Energy Physics (HEP). By accelerating particles to high energies and then colliding them, the LHC aims to answer fundamental questions of Physics, from validating current theories to testing hypotheses exploring the existence of new physics models.

By monitoring and recording these particle collisions, the LHC and its experiments generate data at an unprecedented rate, currently of about 1 billion collisions per second. This data is then stored, processed and analyzed. As the official LHC analysis framework, ROOT [11] plays a fundamental role in this process.

ROOT has been a core component of the HEP software stack for several decades, providing a fairly complete and cohesive framework for HEP analysis. However, due to the increase in requirements for processing power, and the switch in modern architectures from a sequential processing paradigm to a parallel one, ROOT needs to evolve to cope with the computational challenges posed by the ambitious Physics program of the LHC.

In order to tackle these challenges, ROOT is undergoing a significant update, adapting its codebase for the efficient exploitation of current and future hardware resources, and placing an emphasis on the programming model. This thesis is part of ROOT's modernization effort, demonstrating that it is not only possible to adapt a two-decade old, highly complex, established codebase in current use with thousands of users to new processing paradigms, but also to do so without affecting existing code and functionality, in a transparent way for the user, and with a simple yet powerful programming model. This is done by identifying and implementing reusable patterns as building blocks and including libraries for the exploitation of parallelism, providing both users and ROOT developers with a set of tools and libraries that act as computational bricks for the future.

1.1 Motivation and objectives

The next iteration of the the LHC, the High Luminosity LHC (HL-LHC), is expected to increase the amount of collision data gathered in the accelerator by a factor of 10. Traditionally, the additional processing power necessary to analyze an increasing amount of data has been solved either by purchasing more computational resources, as the cheaper option, or by exploiting more efficiently the available hardware, as the more expensive option. However, the amount of data the HL-LHC will generate does not allow us to rely solely on one of these two options.

With this situation in mind, and with the objective of continuing to be the reference framework to process this unprecedented amount of data, ROOT approaches its first 25 years of life by undertaking an extensive modernization effort that will culminate with the next major version of the framework, ROOT 7.

The motivation behind this effort is to reduce the time physicists spend analyzing data. We can achieve this in two ways: improving the execution time of the analysis, which allows us to process an increased amount of data per time unit; and improving the programming model of the analysis, reducing the time physicists spend dealing with the complexity of the tools (programming language, particularities of the memory model), and putting the spotlight on the analysis instead of its implementation. In addition, the modernization of ROOT's codebase will allow for faster development cycles, safer and more efficient utilization of hardware resources, and reduced time spent on code maintenance.

Among other improvements, ROOT is looking for new classes and interfaces that improve the current code in the following aspects:

- User-friendliness: Physicists should be able to focus on the analysis and avoid spending time dealing with unnecessary complexity.
- Performance: New developments should tackle performance bottlenecks, in order to improve the analyses' execution time significantly.
- Reliability: Each new development has to be tested and benchmarked extensively.
- Reusability: If possible, we should develop new solutions as building blocks that can be deployed multiple times, reducing the code's complexity and maintenance costs. For instance, identifying recurrent analysis patterns and providing generalized interfaces implementing them.

The objective of this thesis is to validate the feasibility of implementing solutions in ROOT that incorporate all principles above, and which can act as building blocks for the future without affecting existing code and functionality. This thesis contributes to ROOT's modernization effort with several well-defined objectives:

- Providing an updated description of ROOT and its most relevant features, highlighting new developments in the area of parallelism, to which this thesis contributes fundamentally, and programming model.
- Building a set of tools in ROOT to provide parallelization at task-level that can be applied recurrently throughout ROOT's codebase.
- Introducing data-level parallelism in ROOT's mathematical libraries, applying the necessary changes in ROOT's classes and interfaces to exploit Single Instruction Multiple Data (SIMD) vectorization efficiently.
- Using the utilities developed to parallelize the fit minimization process at task-level and data-level.
- Deploying and leveraging these tools in ROOT codebase and analyzing their impact on performance.

1.2 Structure of the thesis

This thesis is divided into 6 chapters:

- Chapter 2 gives an overview of the processing cycle for LHC data. First, we introduce the LHC chain of accelerators and the LHC's main experiments, describing the lifetime of particles in the accelerator, from injection until collision. Then, we describe the analysis chain, from the acquisition of the data recorded from the collisions, to the simulation and event reconstruction, and the final Physics analysis. Finally, we introduce the computing infrastructure put in place to process this amount of data.
- Chapter 3 describes the ROOT data analysis framework, walking the reader through its most relevant components and features and offering an overall idea of its magnitude and complexity.
- Chapter 4 focuses on task-level parallelism, highlighting the executors, highly flexible parallel MapReduce solutions deployed extensively throughout ROOT that improve both performance and the concurrent programming model.
- Chapter 5 analyzes the performance of the library introduced in ROOT to exploit data-level parallelism, VecCore.
- Chapter 6 applies the improvements in parallelism at task-level and data-level to one of the most computationally demanding operations in ROOT's mathematical libraries, the fitting.

- Chapter 7 exemplifies how the utilities developed have been deployed in other parts of ROOT, showcasing their reusability and relevance.
- Chapter 8 summarizes the results and contributions obtained from this work, and describes future lines of work related to this thesis.

Chapter 2

Physics analysis in the LHC era

The LHC is a circular particle accelerator located at the European Organization for Nuclear Research (CERN) and one of the most sophisticated machines ever built by humanity. In order to recreate conditions similar to those just after the Big Bang, the LHC circulates a beam of charged particles over its 27 km of circumference, increasing their momentum until they reach the appropriate energy, and then collides them at one of the 4 particle detectors in its ring: A Large Ion Collider Experiment (ALICE), A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS) and Large Hadron Collider beauty (LHCb).

Collectively, the LHC detectors, or experiments, produce about 40 petabytes of raw data each year from collisions. This massive amount of data must be stored, processed, and analyzed very efficiently. This in an extremely complex challenge, and it requires several optimization steps of data selection, identification, filtering and reduction until the data is manageable by the data analysis framework of choice, typically ROOT.

This chapter introduces the end-to-end process of generating and analyzing data, from the production of charged particles out of hydrogen, to the final plotting of a physics process generated by a physicist using his personal computer. In Section 2.1 we give an overview of the LHC chain and the different experiments. Section 2.2 describes the different steps involved and the challenges to overcome in the analysis chain, from data collection until the plot obtained as result of the analysis. Finally, in Section 2.3 we expose the computing infrastructure put in place to support the physics analysis and data distribution.

ROOT plays a fundamental role processing LHC data and is involved across all stages of the analysis chain. Understanding the full process of generating and analyzing LHC data provides us with a better comprehension of the source, the magnitude and the complexity of the data processed by ROOT, and the heterogeneity of the analyses programmed with it. Moreover, the massive computational infrastructure currently in place suggests how much we can benefit from maximizing ROOT's ef-

ficiency in exploiting modern hardware resources (e.g. by leveraging parallelism at multiple levels), as the amount of data generated will increase dramatically in the coming years.

2.1 The LHC chain

CERN is a scientific laboratory created in 1954 as an intergovernmental organization in order to spark the study of particle physics, and it is most importantly known by the complex of accelerators it hosts.

A particle accelerator is an extremely complex machine built out of a large number of electromagnetic devices that guide a beam of particles through the path described by its shape. An accelerator aims to increase the energy of charged particles by accelerating them using electromagnetic fields. Depending on the path of an accelerator, we classify it as a linear or circular accelerator.

At CERN, linear accelerators (LINAC) are mainly used to inject the beam into circular accelerators at the right momentum. The circular accelerators constitute a chain in which the newer accelerators that were built for experimenting at higher energies require the beam to perform previous passes through the smaller, older, circular accelerators to ensure it is is injected into the higher energy accelerators at the appropriate energy. The circular topology of these accelerators allows for as many laps as necessary to increase the momentum of the particles until the desired energy is reached.

A schematic drawing of the CERN accelerator complex, including the LHC chain, is given in Figure 2.1. There are two beams circulating simultaneously on the accelerator chain, each consisting of a large number of bunches of $O(10^{11})$ protons. For the LHC the chain starts by extracting protons from hydrogen gas and injecting them into the LINAC2. LINAC2 will accelerate the protons to 50 MeV and then inject them into the PS Booster, which will bring them up to 1.4 GeV. Then both beams are injected into the Proton Synchroton (PS) and accelerated to 25 GeV, after which they are transferred to the last link of the chain—and the second biggest accelerator at CERN—the Super Proton Synchroton (SPS), where they will reach the required injection energy for the LHC, 460 GeV. Finally, in the LHC, the beams will be able to reach up to 7 TeV of energy, in preparation for collision in one of the four main detectors placed at strategic points in the LHC cavern.

Main detectors in the LHC

Figure 2.1 shows the four main detectors, also called experiments, installed in the LHC cavern: CMS, ATLAS, LHCb and ALICE. Detectors are highly complex machines, built out of a large number of state of the art materials and components,

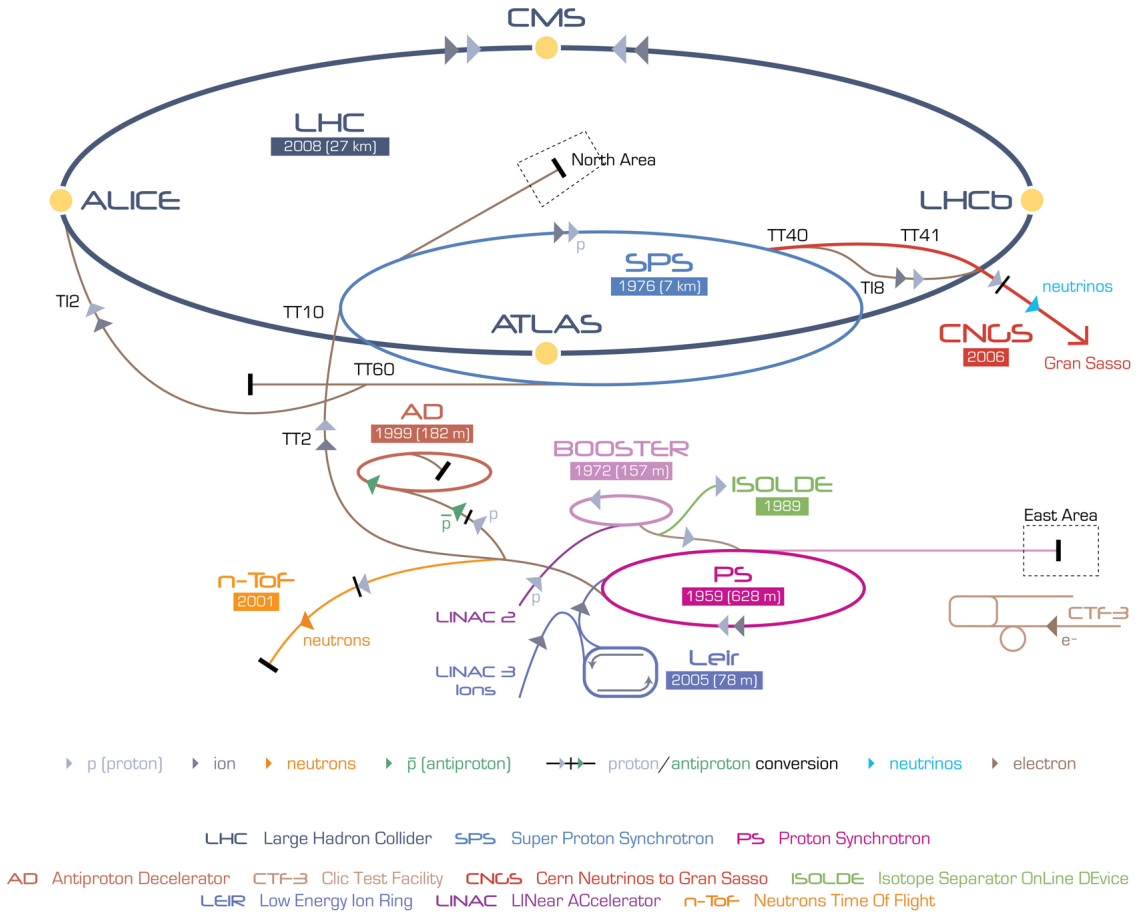


Figure 2.1: The CERN accelerator complex, including the LHC chain. The yellow dots in the LHC represent each one of the four big experiments. Taken from [14].

with the objective of recording the results of the collision of two particles moving at high energies. Figure 2.2 depicts the ATLAS detector and its most important sections, comparing it with the size of an average human.

CMS and ATLAS are two general-purpose detectors used to investigate a wide range of physics. They have complementary characteristics, and they are often used to validate each other's experimental results. LHCb is a detector specialized in b-physics, and its objective is to look for possible indications for new physics by studying b-decays. ALICE performs heavy-ions experimentation to detect quark-gluon plasma, a state of matter thought to have formed just after the Big Bang.

Particle detectors are situated at the collision points of the LHC beam. We collide two particles at extremely high energy to decompose them into smaller more fundamental particles and observe their behaviour, hoping to improve our understanding of physics. The trajectory of the particles resulting from the collision is

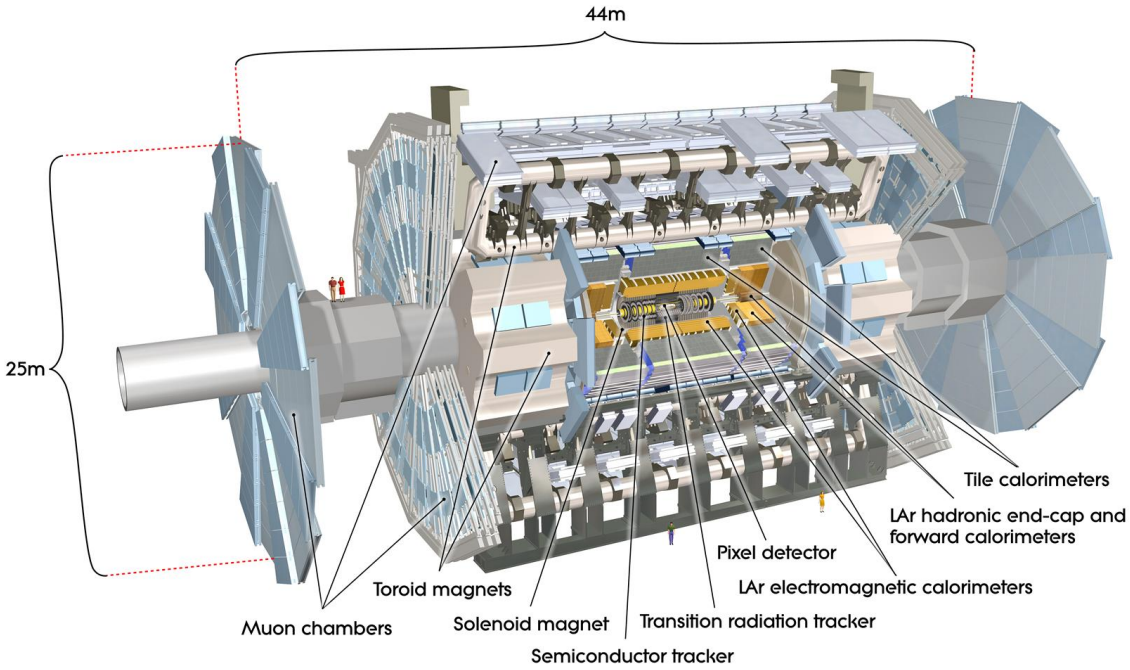


Figure 2.2: The ATLAS detector. Source: CERN [13].

tracked by pixel and strip detectors, and the particles are detected and identified according to their energy and their lifetime inside the detector, that is, to which layer of the detector they penetrate. Figure 2.3 illustrates this in a section of the CMS detector. Photons and electrons deposit their energy in the electromagnetic calorimeter and hadrons in the hadron calorimeter. Instead, muons go through every layer of the detector until they are noticed by the muon system.

2.2 The Analysis chain

Frequently, experimental physicists express their analysis in terms of events (collisions) recorded at a certain point in time. These events are processed in several steps by the analysis chain.

The analysis chain is defined as the necessary actions to observe and analyze the interesting physics processes resulting from the collisions in the detector. It covers the work of going from the signals produced by the collision in the detector to the final analysis visualization and is divided in three different phases or steps: data acquisition, data reconstruction and simulation, and physics analysis.

Data acquisition (Section 2.2) refers to the process of recording a sustainable

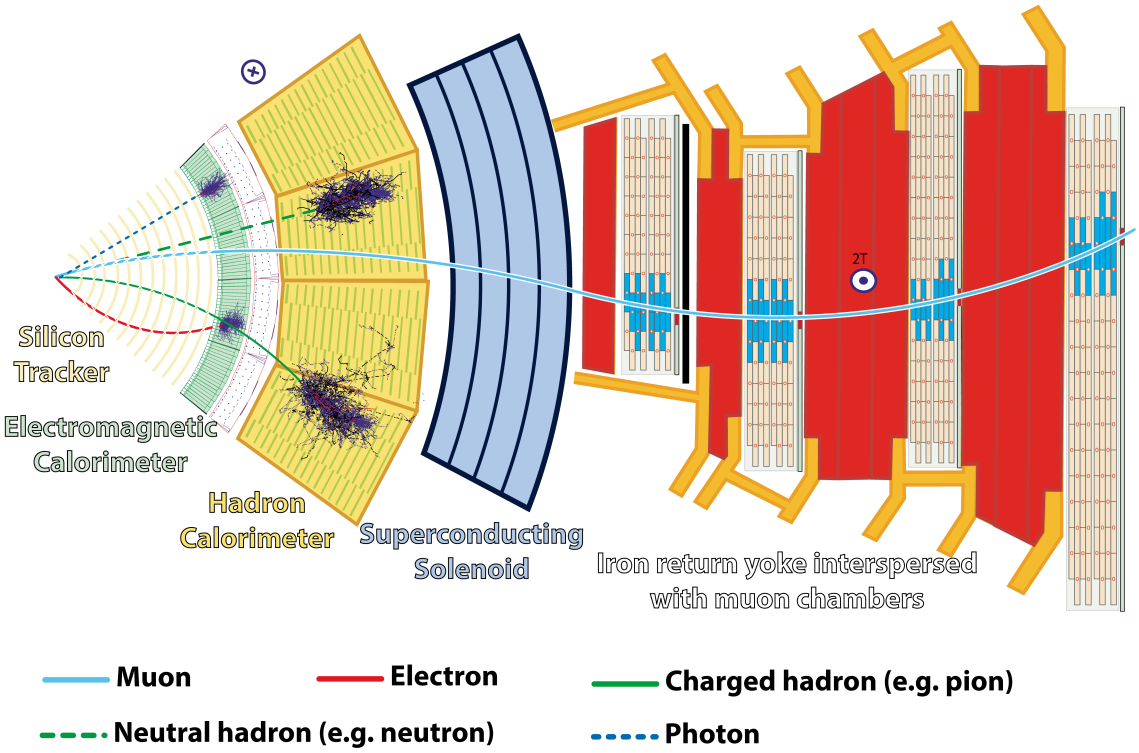


Figure 2.3: Slice of the CMS detector, showing the tracks of different particles (CC by 4.0, taken from [15]).

number of the physical events happening in the detector. Data reconstruction (Section 2.2) and simulation (Section 2.2) involves reconstructing the particles' trajectories after the collision and classifying these particles by type (tagging) from either the raw data acquired from the detector, or from simulations based on a theoretical model. Finally, physics analysis (Section 2.2) focuses on filtering the physics processes of interest from the reconstructed data and comparing them against a theoretical model, in search for its experimental validation or indications of new physics.

Data acquisition

CMS and ATLAS perform approximately $O(10^9)$ collisions per second, generating around 1 megabyte of data each, which results in about 1 petabyte of data per second to record.

Unfortunately, the current state of data acquisition technology is several orders of magnitude away from being capable of handling such a massive amount of data. For this reason, we are forced to record data at a lower rate.

However, collecting data at a slower pace might result in omitting a large number of interesting events from the physics analysis. To avoid this situation, detectors implement the triggers, hybrid software-hardware systems capable of determining the interest of the events and rejecting those that don't show signs of interesting physics processes.

A trigger system is designed with three levels of filters that events need to pass through before being recorded for offline analysis:

- Level-1: hardware-based trigger. Selects events that produce large energy deposits in the calorimeters or hits in the muon chambers.
- Level-2: software-based trigger. Selects events based on a preliminary analysis of the regions of interest identified in level-1.
- Level-3: software trigger. Rudimentarily reconstructs the entire event.

Only the events passing all three filters are stored for further processing, with the next step being the transformation of this data into higher level objects used in physics analysis.

Reconstruction of physics objects from data

The reconstruction step transforms raw detector information into higher level physics objects starting from either the events triggered by the detector's data acquisition system, or from the events obtained from the simulation of the behaviour of the detectors. The latter occurs in programs such as GEANT4 [2]. The output format of this simulation needs to be exactly like the data generated by the detector so it is compatible with the same analysis chain.

The reconstruction process requires the performance of (at least) three operations: *tracking*, reconstruction of the particle trajectories into tracks, determining the parameters of the particles at their point of production and their momentum; *vertexing*, grouping particles into vertices, estimating the location of their production point; and *particle identification*, classifying particles based on their tracks (e.g. photons, muons, etc.).

The particle tracking process involves applying pattern recognition, mapping hits in the detector with specific tracks, and approximating the track with an equation (fitting), usually a Kalman filter [61]. An extra track refinement step is often added, tuning the pattern recognition in order to avoid false positives.

Vertexing involves clustering tracks from a specific event that originated from the same origin point, finding the vertex candidates through cluster analysis and then performing a fitting step, obtaining an estimated vertex position as well as the set of particle tracks associated with that vertex.

The last step, particle identification, aims to associate each identified track with a type of particle. This is a fundamental step, for example, to reduce the volume of data stored for offline analysis to the interesting events, that is, removing background signals which are not necessary for the data analysis process.

In addition, in this step of the analysis chains we perform other operations such as calorimeter reconstruction, to measure the energy of the electromagnetic and hadronic particles, or jet reconstruction, to combine particles in jets using the tracking and calorimeter information.

Monte Carlo generation

The simulation and reconstruction of the detector chain is frequently the most time-consuming step in the analysis chain.

Often, physicists choose to work with very simplified simulations of the detector's observable events . These simulations are produced by event generators, instances of simulation software used to model and simulate physics processes described by a theoretical model. They do so with the aid of Monte Carlo techniques and algorithms, relying on random sampling to produce events with the same average behaviour and fluctuations as real data [52].

Monte Carlo event generators aim to simulate the experimental characteristics of the physics processes of interest, and they are a fundamental tool for HEP, with a wide variety of applications such as optimizing the design of new detectors for specific physics events, exploring analysis strategies to be used in real data or interpreting observed results in terms of the underlying theory.

A large number of Monte Carlo event generators are available for the simulation and analysis of HEP theory, e.g. Pythia [1], Herwig [18] or Alpgen [36].

Physics analysis

The analysis process starts after reconstruction or simulation of the collisions. Once we obtain the reconstructed data, stored in ROOT format (Chapter 3), we apply to it successive campaigns of data reduction and refinement. These campaigns consist of filtering and selecting the events relevant for the analysis at hand, resulting in a reduced dataset. The data reduction process is driven by the physics processes of interest, e.g. selecting all the events that involve a Higgs gamma-gamma decay, and aims to produce an amount of data manageable from the local computing infrastructures of the experiment or the physicist.

This reduced data is then processed by the analysis software to produce data frames (tables) and histogram visualizations, to which statistical inference is ap-

[1] Pythia

plied. This is done by applying sophisticated algorithms, such as those used for the classification of the signal versus the background data of an analysis, for regression analysis, or for the fitting of probabilistic distributions (Chapter 6) for the estimation of physical quantities and observables, for instance estimating the Higgs mass.

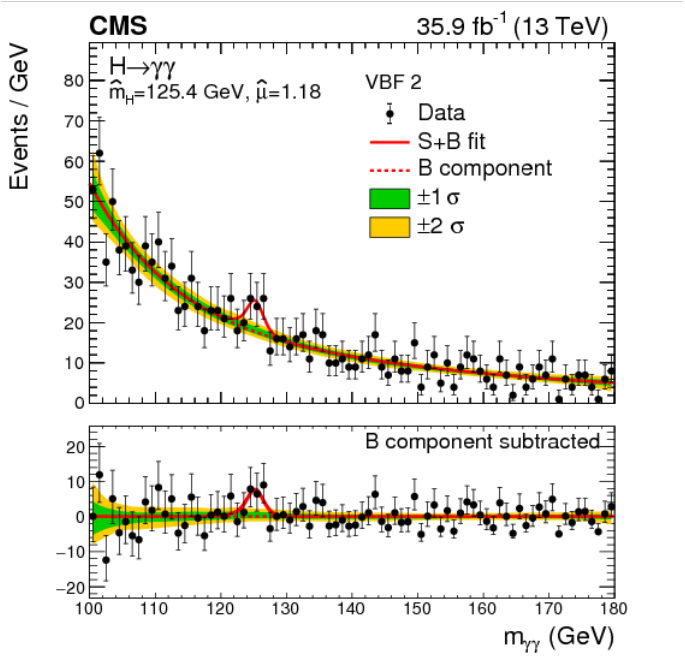
Figure 2.4 displays two visualizations generated from the analysis of measured Higgs boson properties using the diphoton [17] and four-lepton [16] decays channels, based on data collected by the CMS detector. These figures, produced by ROOT, showcase two of the most common procedures to be applied to reconstructed data for analysis. In Figure 2.4a, we aim to fit the data to a theoretical model. Through fitting, we estimate physical quantities values (parameters of the fit) to obtain the parameter values that describe best the distribution of the reconstructed data. Figure 2.4b plots the reconstructed data (points with error bars) against several stacked histograms obtained from simulated data based on theoretical models, representing expected signal and background distributions.

In addition to traditional statistical inference methods, recently there has been a rise in popularity of multivariate analysis methods based in machine learning. In the scope of HEP, we benefit from the (increasingly) intensive use of techniques such as Deep Neural Networks (DNN) or Boosted Decision Trees (BDT). These methods are mainly used for signal-background classification or regression analysis, e.g. for estimating the particle energy from the calorimeter data. These methods have proven very effective, dramatically reducing the time necessary for these processes and improving accuracy.

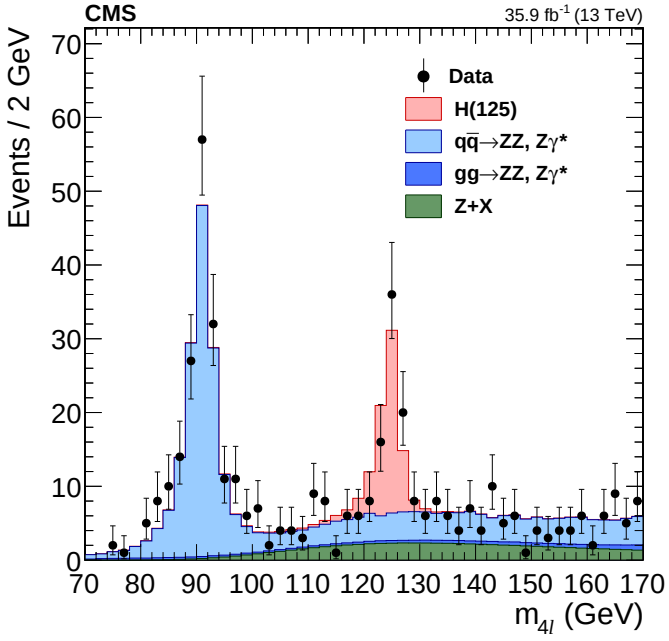
HEP analyses are typically performed by ROOT, the official LHC data analysis toolkit. ROOT provides most of the numerical and statistical methods necessary for the analysis of HEP data. In addition, it offers several attractive features for HEP analysis, such as a C++ interactive interpreter for prototyping and interactive exploration of the analysis, implicit parallelization, new analysis-centric programming paradigms such as TDataFrame, and, most importantly, a highly efficient input/output (I/O) subsystem. ROOT I/O provides an efficient columnar data format, data compression, serialization of C++ objects, and data structures optimized for dealing with the extreme necessities of today's HEP experiments.

2.3 Computing infrastructure at CERN

The LHC experiments generate new data at increasing rates, currently around 40 petabytes annually. This makes it essential to adopt a data-centric computing model, where the data, our most important asset, is at the core of the choices made in the deployment, distribution, storage and processing of the data, the design of computer centers or cluster infrastructures, etc. For instance, the storage, networking



(a) Data and signal-plus-background model fits for the vector-boson fusion Higgs production and top-Higgs production categories in the diphoton decay channel. The one (green) and two (yellow) standard deviation bands include the uncertainties in the background component of the fit. The lower panel shows the residuals after the background subtraction. From [17].



(b) Distribution of the reconstructed four-lepton invariant mass $m_{4\ell}$ in the low-mass range. Points with error bars represent the data and stacked histograms represent expected signal and background distributions. From [16].

Figure 2.4: Mass spectra obtained in the CMS Higgs search using different decay channels, with a significance $> 5\sigma$. Data collected in 2017 at a center-of-mass energy of 13 TeV and an integrated luminosity of $35.9 fb^{-1}$.

and processing resources that would be necessary to perform local analysis on such large remote data sets would exceed anything offered by the commercial solutions currently available. Thus, we need to think of potentially ad-hoc solutions, such as performing the analysis remotely, as close as possible to the data, to reduce the load on the network and facilitate low-latency, high-rate access to the data, or creating a hierarchical federation of interconnected computing centers to improve data distribution, locality and processing time.

For instance, processing this enormous amount of data requires, at least, 500000 typical PC processor cores. A computing center as big as CERN's still cannot provide enough resources to store and process such a large amount of data, both in terms of processing and electrical power, and is estimated to be able to cover only around 20-30% of the storage and CPU power needed.

This task of unprecedented magnitude and complexity sparked the creation of a global computing infrastructure, the Worldwide LHC Computing Grid (WLCG) [51]. The WLCG is an international collaboration that involves several national and international grid infrastructures and many institutions, communities and projects from a wide variety of fields, such as HEP, astrophysics, earth sciences or biological and medical research.

The WLCG is a multi-tiered, geographically distributed, federation of computing centers of various sizes orchestrated from central points. Depending on which tier a computer center is part of, it will provide different services. In the scope of HEP and, specifically, the LHC data analysis chain:

- Tier-0: A single computing center tier, built at CERN, providing 20% of the computing power of the WLCG. Stores all the raw data obtained from the detectors' data acquisition system, runs the first pass of reconstruction and orchestrates the distribution of the raw data to the following tiers.

- Tier-1: Thirteen large computing centers used for simulation and reconstruction of the data received from the Tier-0. They keep a secondary copy of the raw data and distribute the reconstructed data to the Tier-2 centers.

- Tier-2: Around 160 smaller computing centers, typically universities or research centers, with adequate computing power for users to perform analysis processes, calibration measurements, and the generation of Monte Carlo simulations. They provide redundancy for the reconstructed data and also store the results of users' data analyses.

- Tier-3: Local computing clusters managed by local analysis groups or even individual computers. Although we often refer to them as Tier-3, the WLCG does not provide any specification or formal engagement with them.

Although the roles of each tier still hold, recent years have seen an improvement in and reduction of the cost of network links, and now it is possible, for instance, to transfer data between centers of the same tier or to build a set of fast network hubs connecting many Tier-2 to many Tier-1 and Tier-0.

The WLCG is a fundamental infrastructure for the HEP community and has been a core concept for the processing of the analysis chain during the last decade. Proof of that is that the computing model of the LHC experiments is still especially designed for the exploitation of grid resources.

Chapter 3

ROOT

ROOT [11] is an open-source, cross-platform, object-oriented C++ framework for data analysis, conceived for the practice of HEP. Starting as a private project by Rene Brun and Fons Rademakers, ROOT has been at the heart of most HEP analyses during more than two decades, in continuous development and improvement, being in constant communication with the community, obtaining invaluable feedback, providing great toolkit support, and evolving the product according to the needs of its userbase.

ROOT experienced an early fast adoption and it was soon adopted as the official LHC analysis framework. Today, ROOT is a fundamental tool in the field of HEP, and continues to be developed as a collaboration between research institutions and particular users around the world. Every day thousands of physicists use ROOT-based applications to analyze and visualize their data. Designed for storing and analyzing exabytes of data efficiently, ROOT offers a coherent set of features oriented to HEP data analysis. One of ROOT's most important features is the streaming of C++ classes to a compressed machine-independent file format. These C++ classes are often the ones ROOT optimizes and offers for the statistical analysis of large datasets in columnar (stored and accessed column-wise) format.

In order to perform statistical analysis, physicists have at their disposition in ROOT a considerable number of functions, methods and libraries (e.g. RooFit, RooStats) especially oriented to the most common operations in HEP, such as fitting or multivariate analysis. The results of the analysis can be visualized directly from ROOT, without the need for external libraries, and stored in high quality graphic plots in any of the most common and important formats.

The complete analysis can be developed incrementally with Cling, ROOT's interactive interpreter, benefiting from fast prototyping and exploration of ideas. Furthermore, it can be programmed in languages other than C++ thanks to ROOT's language bindings for R and Python. The library implementing Python bindings, PyROOT, is increasing in popularity due to the possibility to interface with widely

used Python scientific libraries, such as NumPy or TensorFlow. ROOT acknowledges this by designing its new features with it in mind, e.g. RDataFrame [23].

This chapter describes some of ROOT's most important components. ROOT's long-lived success and early fast adoption has been traditionally based on its highly useful features for HEP analysis, such as its interactive C++ interpreter (Section 3.1); its high-performant I/O (Section 3.2); its mathematical and statistical libraries (Section 3.4); and the graphic representation of the analysis (Section 3.6). Furthermore, ROOT now offers new exciting developments for the physicists of tomorrow, such as the new paradigm for declarative analysis, RDataFrame (Section 3.5); the adoption of implicit multithreaded parallelism by default (Section 3.3); the increasingly ubiquity of SIMD operations in Math libraries (Section 3.3); the possibility to perform web-based analysis with the Service for Web based ANalisys (SWAN) [43]; and the new redesigned interfaces for the next version of the software, ROOT 7, that are being deployed gradually into the main codebase.

This updated description of ROOT details the framework and introduces ROOT's components referred to later in this thesis. The parallelization tools and libraries we introduce in Chapters 4 and 5 are utilities deployed throughout ROOT's codebase, for instance, for the parallel processing of TTrees, for the vectorization and parallelization of the fit, and for the generation of vectorized code with Cling.

3.1 Cling

The ROOT userbase is mainly composed of non-expert users looking for a user-friendly and fast way to program their data analyses. Compiled languages, like C++, require investing a considerable amount of time in the compilation and linking stages, even more so in large codebases such as that of ROOT. This inconvenience makes interpreted languages (e.g. Python) more attractive when performance is not paramount for the nature of the problem at hand, e.g. for exploration of interfaces or for validating the execution of a simple instruction.

One of the main benefits offered by an interpreter is faster prototyping for Rapid Application Development (RAD) [62]. Prototyping allows the developer to explore ideas cheaply (reduced cost), and it is an efficient way to identify and address problems early on, to identify requirements and to estimate development costs, timescales and potential resource requirements.

In addition to reducing the time the user spends in compilation and linking, interactive interpreters allow users to explore ideas even more efficiently, using the fewest resources possible, in order to validate the ideas' effectiveness and detect, early on, any problems in their design (such as wrong assumptions). For improved usability and user-friendliness, these interpreters adopt the read-eval-print-loop (REPL) [63] environment model, providing users with an interactive environment that executes

single inputs and returns the result after evaluation. In addition, a REPL interpreter usually extends the programming language by making it possible to run expressions at the global level and by implementing special commands that provide information about the current state of the environment (Reflection).

Conveniently, current compiler technology allows the emulation of interpreter behaviour in compiled languages by means of incremental compilation and hybrid compilers, that is, compilers offering hybrid compilation-interpretation and Just-In-Time (JIT) compilation. To emulate interactive interpretation, the hybrid compiler translates the source code into a machine-independent, observable, intermediate representation that is compiled and executed by the JIT compiler at runtime.

In order to accelerate development speed and quality, and to help overcome the complexity of the framework and the programming language, ROOT is distributed with Cling [55], a user-friendly and interactive C++ interpreter built on top of Clang and the LLVM compiler infrastructure.

The Cling interpreter is a fundamental component of ROOT that is used in four principal ways:

- **JIT compiling:** ROOT makes extensive use of Just-In-Time compilation throughout its code, most commonly to simplify its programming model and its user interface. It is used, for example, for the compilation at runtime of mathematical expressions provided as a string or to compile the simplified interfaces of RDataFrame after adding its template parameters at runtime (Section 3.5).

- **Reflection and data-type information:** Reflection, the ability of a computer program to inspect and modify its own data, is necessary for the serialization and deserialization of data classes in I/O operations. Thanks to the interpreter, ROOT is capable of retrieving information from the data types at runtime, interfacing clang through cling, to provide metadata to the write-out of the objects on disk (serialization). When reading this data from disk, the interpreter will utilize the metadata to instantiate at runtime, by *jitting*, the objects read from the file.

- **The ROOT prompt:** The prompt, a REPL environment offered by the interpreter, provides the user with immediacy and interactivity one line of code at a time, making easier and faster the exploration and modification of ideas. Figure 3.1 displays two of its advantages: the value printing of C++ objects and built-in compiler diagnostics, improved from the clang ones. Figure 3.2 exemplifies how we can use the interactive interpreter to approach complex programs, such as the customization of a visualization, incrementally. By obtaining visibility into the program's state, thanks to Cling, we benefit from immediate feedback on simple mistakes early on.

- **Support for python bindings:** The interpreter provides automatic dynamic bindings between the python library (PyROOT) and the implementation of the classes in C++. It automatically generates a python wrapper over the ROOT classes allowing Python to execute C++ code without explicit development of the C++-Python bindings.

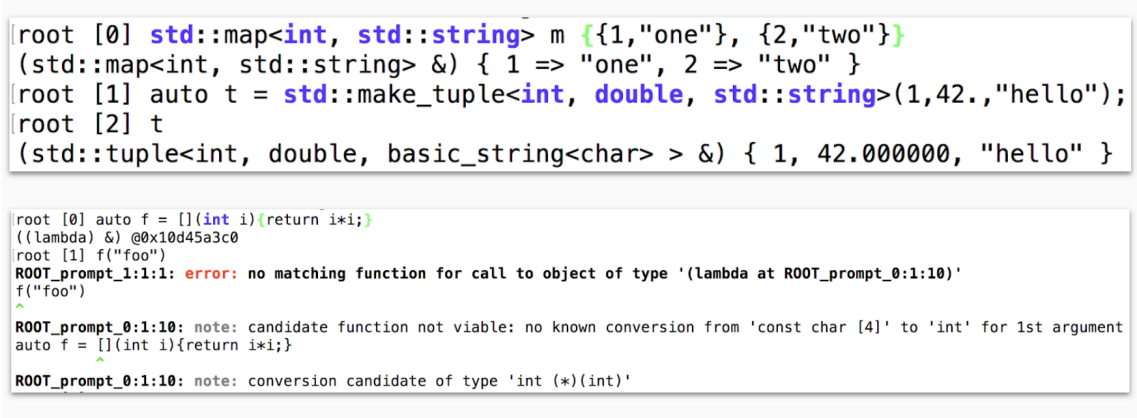


Figure 3.1: The ROOT interpreter, Cling, allows the inspection of the C++ objects and offers immediate validation of the implementations.

3.2 Input/Output

Each year, ROOT is used to process around 50 petabytes of data. Handling such a large amount of data annually would be unreasonable without highly optimized procedures to address the limitations of the Input/Output (I/O) and data distribution systems.

For this purpose, ROOT provides a columnar file format, the ROOT file, and data structures for optimizing the reading from and writing to disk, such as the TTree, which provide the user with multiple improvements that make processing data with ROOT the most performant option available for HEP analysis. Some of these remarkable features are automatic compression and decompression, selective sparse scanning of data, directory-like storage, automatic reading of data partitioned into several files, data consistency over the lifetime of an experiment or taking advantage of the reflection capabilities of Cling to seamlessly read and write C++ objects on disk.

This section introduces the most important features in ROOT's I/O subsystem. Section 3.2 describes the physical and logical formats of ROOT objects and Section 3.2 describes the main data structure for data representation in ROOT, the TTree.

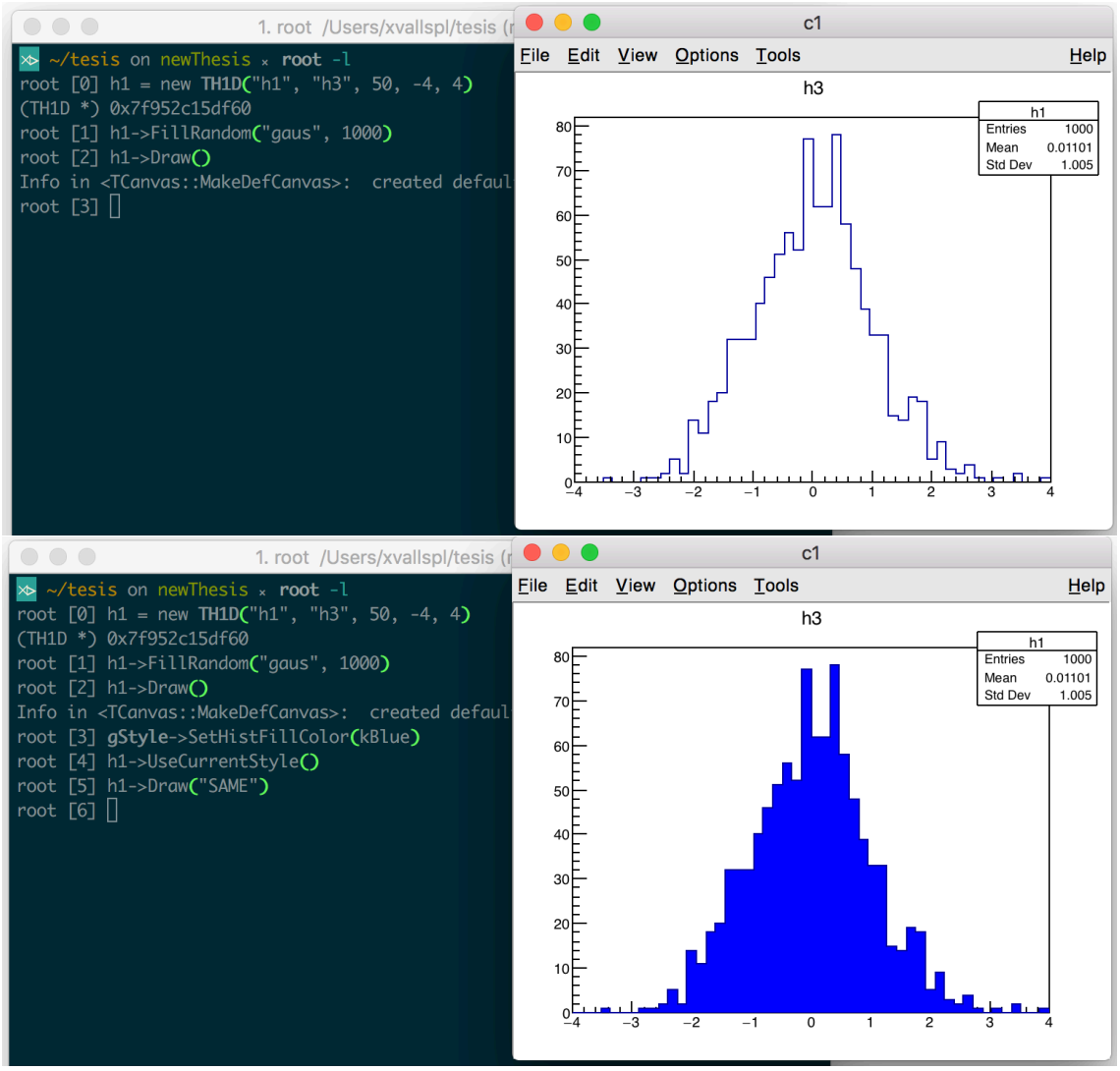


Figure 3.2: Cling used to approach a problem incrementally, in this case, to explore the customization of the visualization. We can modify the aspect of the default graphical representation of a histogram at runtime, without needing to compile and run again our program.

TFile and the ROOT file format

As introduced in Section 3.1, a very interesting feature that ROOT offers is the possibility to automatically store and read any C++ object from a file. These processes, known as serialization and deserialization respectively, are possible thanks to the reflection capabilities offered by Cling and a file format with a physical and logical structure that contribute to implementing these operations efficiently: the

ROOT file.

The TFile object is the class used within ROOT to read and write ROOT files and it is, therefore, the class in charge of the automatic serialization and deserialization of C++ objects, in addition to handling data compression. Based on the information of the data provided by the interpreter, ROOT can locate where the object's data members are in memory, and know their size and how to store them.

Nevertheless, when describing the access to files in ROOT, it is necessary to consider the physical format of the ROOT files and the logic implemented on top in order to provide enhancements such as faster lookups, compression or integrity checks. That is to say, TFile depends on precisely defined physical and logical file formats to provide efficient storage, parsing and reading of the data.

Figure 3.3 depicts the the ROOT file format physical structure. A ROOT file contains three kinds of data: the file header, including metadata about the file/directory and redundancy to check its integrity; a list of data object descriptors (Logical record header/TKey); and the data of the objects we store in the file. In order to limit the bandwidth usage and to reduce the amount of space a ROOT file takes on disk, all this data, except the header, is compressed by default, with LZ4 as the default compressing algorithm of choice [10]. However, working with compressed ROOT files has a cost, as both time and CPU power are required to uncompress when reading the contents of a file, and to compress when writing the data into the ROOT file.

A ROOT file can contain unlimited levels of directories and objects, but the data of the ROOT objects is written in the file in consecutive order. In order to allow lookup on the object identifiers, the file header stores a pointer to the first element of the linked list of data objects' logical records. Logically, this translates to keeping a hash list of entries (keys) for each directory and subdirectory (Figure 3.4). All this allows efficient sequential, random and hierarchical access to the data stored in the file.

Finally, ROOT also considers data integrity when reading ROOT objects from a file written with a previous version of the framework. Because the description of all relevant classes is stored with the data, we need to support the evolution of the stored classes over the lifetime of a collaboration. For this purpose, ROOT will compare the versions of the persistified object and the object in memory, and automatically translate to the new format if possible. This feature is known as *schema evolution*.

All these features make the ROOT file format the most performant solution, both when reading and writing, compared to the most popular commercially available column-wise data storage formats [33] [9]. This is in large part thanks to the TTree class.

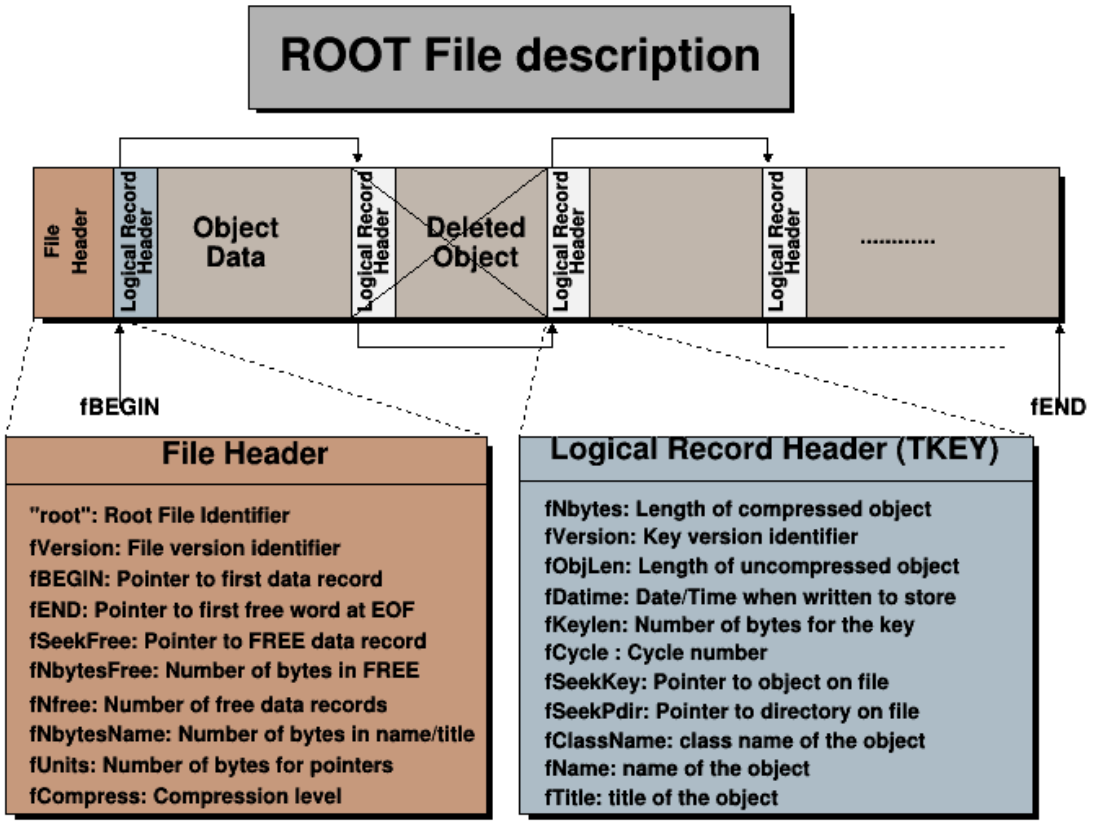


Figure 3.3: Description of the ROOT file physical format. Taken from [44].

The ROOT Trees

TTree is the ROOT class optimized for I/O and memory usage. ROOT's TTree represents a forest of Trees, each representing an entry, such as a specific event data, in the dataset. A ROOT Tree consists of branches and sub-branches, ultimately described by their leaves (data object). The leaves may be either complete objects of a class or individual data members of a given class, split as sub-branches of the same parent branch. This splitting process can be applied recursively, allowing for optimizations such as the grouping in branches of the data members of the different elements of a container. The optimal level of splitting depends on the tree's usual access patterns, providing a great benefit in analyses where we only access few data members of the tree stored objects.

Splitting is especially useful for the column-wise storage (CWS) ROOT implements. By selectively reading the data needed at each moment, CWS reduces the volume of transferred data and I/O operations, improving dramatically the speed when gathering data. ROOT takes advantage of this by grouping in branches and

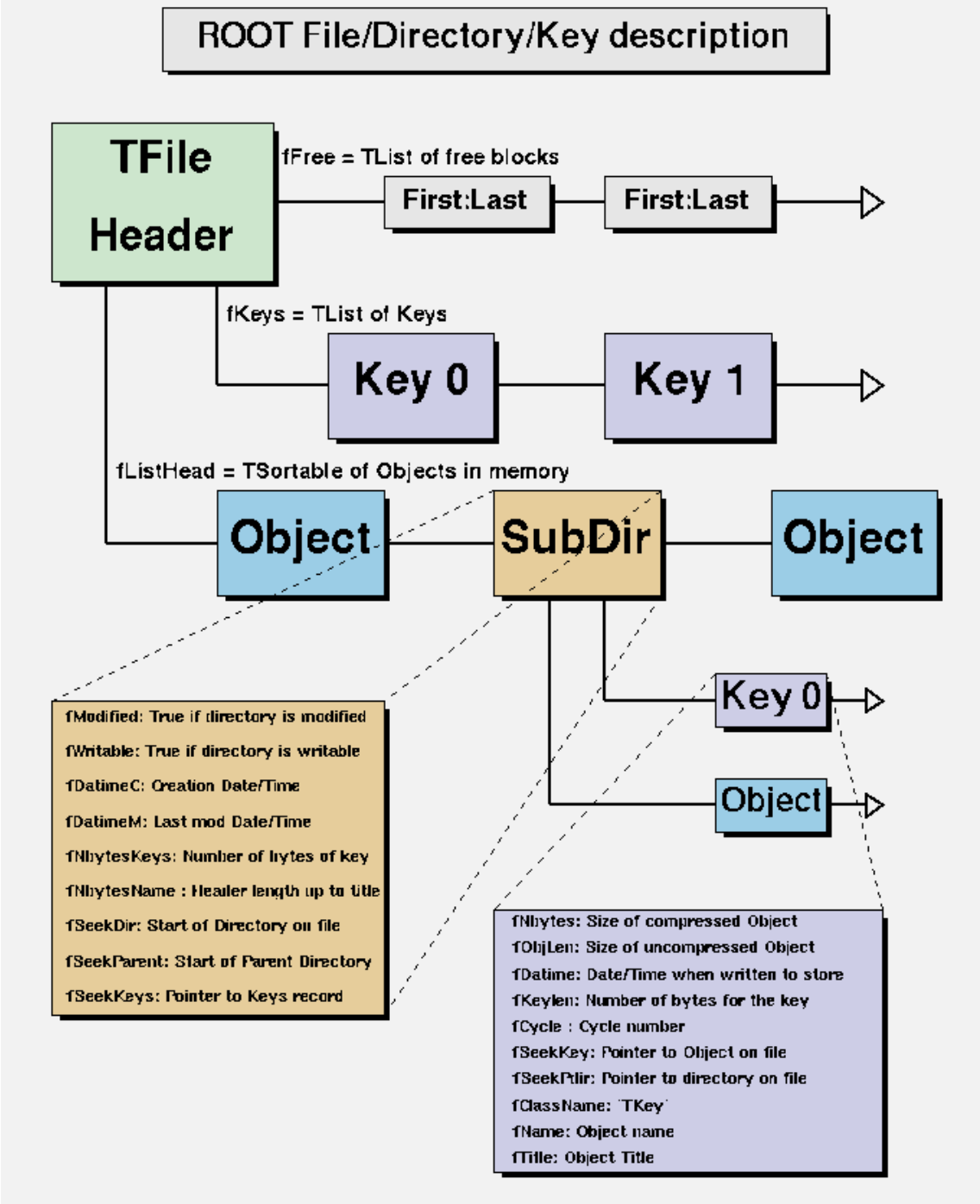


Figure 3.4: Description of the ROOT file logical format. Taken from [44].

storing contiguously the data elements frequently accessed together, optimizing random lookups to the data. This grouping, defined by the level of splitting, provides even higher benefits by applying further optimizations such as the parallel reading of the tree branches.

ROOT also implements several optimizations to improve writing accesses to memory, e.g. buffer writing; continuing with the analogy, when we gather the fruits (data) of a given branch, we collect them into baskets (buffers). When these baskets are full, we take them to the storage system (the file). This helps greatly reduce the number of (costly) accesses to disk.

However, given the nature of the data commonly processed by ROOT's typical analyses, TTrees are usually too large to be practically stored in a single file, in terms of space, distribution or processing time. ROOT implements the class TChain for the reading of TTrees split in several TFiles, which abstracts the splitting from the user perspective. The storage of partitioned TTrees over several file facilitates the distribution of data in remote systems and the parallel and distributed processing of data analyses.

In addition, analyses usually access the same subset of branches for each one of the entries of a TTree. To dramatically improve this access, ROOT implements adaptative prefetching mechanisms, reading the following entry of the one being processed. This improves greatly the performance when processing remote ROOT files, making the bandwidth of the network, previously an important limiting factor, no longer a consideration.

3.3 Parallelism

First developed in 1994—with segments of code dating up to a decade earlier—ROOT was already conceived with parallelism in mind. However, although considerable work was put into it by offering classes (TThread) and mechanisms for synchronization and mutual exclusion (e.g., TMutex or TCondition), this was not enough to meet the challenges posed by the ambitious LHC Physics program.

ROOT, a tool adopted by virtually all HEP experiments, needed to adapt to emerging hardware architectures but, while multiprocessing efforts were a success with the emergence of facilities such as The Parallel ROOT Facility (PROOF), the multithreading efforts were hindered by early software architecture designs as they grew in ambition. These early designs were dependent on inherently non-thread-safe components, such as global variables or global bookkeeping and automatic garbage collection. In recent years, a greater allocation of manpower and other resources has enabled the deployment of multithreaded parallelism in a significant portion of the codebase, covering ROOT's most performance-critical operations.

In this section we present some of the most relevant efforts in concurrent computing that ROOT has been focused on: PROOF, the parallel ROOT facility for distributed analysis; the executors, a parallel implementation of the MapReduce pattern in ROOT (3.3); and the push in the user-oriented programming model, exemplified by the two modes of multithreaded execution of ROOT (Section 3.3): implicit and explicit execution modes, and vectorization (Section 3.3).

PROOF

Multiprocessing in ROOT is particularly attractive because of its platform-independent streaming of C++ objects to disk, a feature that cannot be offered by any other multiprocessing framework. PROOF was a response to an increased demand for processing power at a time when the standard ROOT approach could not take advantage of the full processing capability of multi-core computers.

PROOF is an extension of ROOT that provides users with transparent, interactive multiprocess analysis over large remote datasets. PROOF is explicitly designed to work with the ROOT data stores and objects—such as the class used in ROOT for persistent event data, the TTree—relying on optimized initialization and traversing techniques, via the TSelector class.

The PROOF system is characterized by three properties: transparency, not requiring changes in the user programming model after set up; scalability, not limiting the number of processing units able to be used in parallel; and adaptability, being able to adapt itself to the changes in the remote environment. It presents a multi-tier architecture, its main components being the ROOT client sessions, the master node, and the slave nodes. In addition, it offers the possibility of executing physics analysis on a cluster of computers.

During the typical setup workflow of PROOF for a cluster, the user connects to the master node in the cluster, and the master node creates a slave server on each one of the nodes. In addition, we create a user space for each user of the system, where PROOF caches the packages and code used for the analysis and a temporary session log file. Each slave node, or worker, executes independently its own ROOT session, connecting the input and output streams of the command line interpreter to the network sockets. Once set up, the system is ready to process the physics analysis in a distributed, parallel, multi-processed fashion.

PROOF automatically takes care of synchronization, monitoring and scheduling, abstracting these complexities from the user. For better performance and load balancing, the analysis is split into small packages which will be assigned to the nodes on demand. The size of each package (the amount of work to process) depends on the characteristics (frequency, memory) of the client (slave) node.

Users program their analysis for PROOF as they would do for ROOT. PROOF was designed to support the programming models ROOT offered at the time for the

expression of the analysis, namely by interacting with the GUI, programmatically via TSelectors and with TTDraw().

Although PROOF is currently kept in maintenance mode, it is still used in a considerable number of HEP experiments, and remains the only distributed analysis tool able to fully exploit ROOT capabilities. For example, at CHEP 2015 [49], PROOF still featured in 7 of the presented proceedings for the conference.

Although PROOF provides a good solution for multi-node systems, due to the overhead that arises from the initialization and usage of its many components (which might not be necessary), it can exhibit suboptimal performance for analyses executed on local, single-node multicore hosts. For these analyses, PROOF might be replaced with PROOF-Lite [3], a lighter version of PROOF sharing the same programming model but offloading many components of PROOF which are not needed for single node execution (such as network connectivity). In addition, depending on the nature of the problem, users might benefit from using ROOT's MapReduce facility for multi-process analysis in single node, TProcessExecutor [21] (Section 4.3), the multiprocess implementation of the executors' programming model.

Executors: A new set of tools for expressing parallelism in ROOT

It is often possible to identify repetitive tasks and frequently applied patterns or workflows in HEP analyses. Some of these patterns describe operations that are easy to express concurrently. ROOT exploits this idea by offering a parallel implementation of one of the most frequent patterns: MapReduce.

For this purpose, ROOT gathers inspiration from the Python concept of the executors, facilities that implement the MapReduce pattern in parallel. ROOT offers two executors, both of them implementing the interface defined by the parent class TExecutor:

- TProcessExecutor: ROOT's Multiprocess executor, a lighter multiprocessing framework than PROOF or PROOF-Lite, but limited to MapReduce operations. In addition to the executor interface, ROOT also extends TProcessExecutor functionalities with parallel processing and reading of ROOT trees.

- TThreadExecutor: ROOT's Multithreaded executor, built on top of the Intel TBB library. This executor offers chunking capabilities, NUMA awareness, parallel reduction and very low task overhead.

The executor model has proved to be successful and has been applied throughout ROOT as well as in standalone utilities such as hadd (Section 7.1). It yields large speed up gains, especially in the most performance-critical operations, such as

parallel reading of TTree branches (intra-event or inter-event) or parallel merging of buffers before writing. These executors are described in more detail in Chapter 4.

Implicit and explicit multithreading

Parallel computing is usually considered challenging and complex, but it is unavoidable in today's computer architectures when aiming for high performance.

One of the ways to overcome this complexity is to simplify the programming model. However, there is a tradeoff between giving users full control over the parallelization of the program, which requires at least basic knowledge of concepts such as threads, locks, or critical sections; and hiding the complexity from the user, which limits the parallelization to a certain set of predefined operations. Since version 6.12, ROOT offers both ways of expressing parallelism in a multithreaded way:

- Explicit parallelism: users express parallelism explicitly, by working directly with the threading facilities of the C++ standard library; by using the libraries of their choice, such as TBB, Boost or OpenMP; or by abstracting the complexity by using the set of utilities for expressing parallelism ROOT provides.
- Implicit parallelism: ROOT implicitly manages the concurrency in the user code, parallelizing expensive operations without requiring actions from the user.

Although ROOT also offers tools for the explicit expression of multithreaded parallelism, such as the executors, a great effort has been invested to adapt ROOT's libraries to leverage implicit multithreading, and users are encouraged to prefer it over explicit parallelization. Nevertheless, the two options are not interface-exclusive, and, in some of the cases, for example when performing a fit (Section 3.4), for user convenience, flexibility or interface coherency, both implicit and explicit parallelism are offered.

Vectorization

ROOT implements parallelism at data-level by exploiting single instruction, multiple data (SIMD) vectorization. For this purpose, ROOT relies on three external libraries:

- **VDT** is a mathematical library of fast implementations of transcendental functions automatically vectorizable by modern compilers. This includes the logarithm function, trigonometric functions and the inverse function.

- **Vc** is a library designed to simplify the explicit expression of SIMD operations, offering a much more user-friendly programming model than the compiler intrinsics it is built on.

- **VecCore** is a library that enables efficient vectorization by providing an abstraction layer on top of other vectorization libraries, such as Vc or UME::SIMD. This extra layer of abstraction facilitates the expression of SIMD operations in a user-friendly way. These libraries are supported as interchangeable backends along with a fallback scalar one, allowing the selection of the appropriate backend (e.g., the most efficient, the one that offers better support for the set of instructions available, etc.) for each specific architecture.

Although the three libraries can be built and enabled during the ROOT compilation process, VecCore is the only library explicitly used in ROOT, and enabling it allows users to benefit from features such as the new ROOT types relying on VecCore to perform SIMD operations; the support for functions built with these ROOT SIMD types in TF1 (Section 5.3); the vectorized evaluation of the fitting given a vectorized TF1 (Chapter 6); or the compilation at runtime of vectorized functions thanks to TFormula (Section 7.3).

3.4 Mathematical libraries

Analyzing a large amount of complex data, such as that gathered by the LHC experiments, requires advanced mathematical and statistical computational methods. ROOT's Math libraries [39] are responsible for providing and supporting a coherent set of mathematical and statistical libraries required for simulation, reconstruction and analysis of High Energy Physics data.

These libraries are divided into several parts:

- **MathCore** implements basic mathematical functions and algorithms commonly used in HEP data analysis, as well as a set of interfaces for its use and extension. They allow the user to plug-in at runtime different implementations of the numerical algorithms (e.g. different minimization algorithms when fitting), which are not required to co-exist in the same library (e.g., picking algorithms from ROOT's MathMore library). MathCore includes utilities such as commonly used free functions, classes for random number generation, basic numerical algorithms (e.g., integration, derivation, minimization), and interfaces for defining function evaluation in one or more dimensions.

- **MathMore** provides a more advanced collection of functions and interfaces for numerical computing. This library, an extension of MathCore that might

be needed only for specific applications, wraps the GNU Scientific Library (GSL) and includes special mathematical functions, statistical and probabilistic functions present in GSL but not in MathCore, numerical algorithms from GSL, numerical integration and derivation classes, interpolation, function approximation using Chebysev polynomials, polynomial evaluation, and ROOT solvers.

- The **Linear Algebra** libraries implement a complete environment for ROOT to perform calculations such as equation solving and eigenvalue decomposition, including the vector and matrix classes and linear algebra functions. ROOT implements its linear algebra operations in two libraries: a general matrix package (TMatrix in Figure 3.5) and an optimized one (SMatrix) for small and fixed-size matrices.

- The **fitting and minimization libraries** provide classes and libraries that implement various fitting algorithms and function minimization methods. This includes both the classes and interfaces under the namespace Fit in ROOT and RooFit, used for fitting of probabilistic distributions (Sections 3.4 and 3.4 respectively), and the various minimization libraries (minimizers) supported by ROOT for different minimization algorithms. These minimizers are distributed, depending on their usage, throughout ROOT math libraries, i.e. in MathCore, such as Minuit 2 or Fumili, or in MathMore, such as the minimizers from the GSL.

- The **Statistical libraries** are mostly composed of libraries that implement machine learning and multivariate analysis methods (TMVA) and advanced statistical tools for computing confidence levels and discovery significances (RooStats, Section 3.4).

- The **Histogram library** is used for displaying and analyzing one-dimensional or multidimensional binned data. Histograms are the most common way of representing the large amounts of data gathered in HEP experiments, as they reduce the complexity of the computations several fold without necessarily reducing their accuracy or significance in a considerable way. This library also contains the bounded function class in ROOT, TF1, which is fundamental for most of the mathematical operations in the libraries mentioned above.

Fitting

ROOT provides several techniques for fitting [40] that are applied depending on the user's choice or type of data (differentiating between binned and unbinned data).

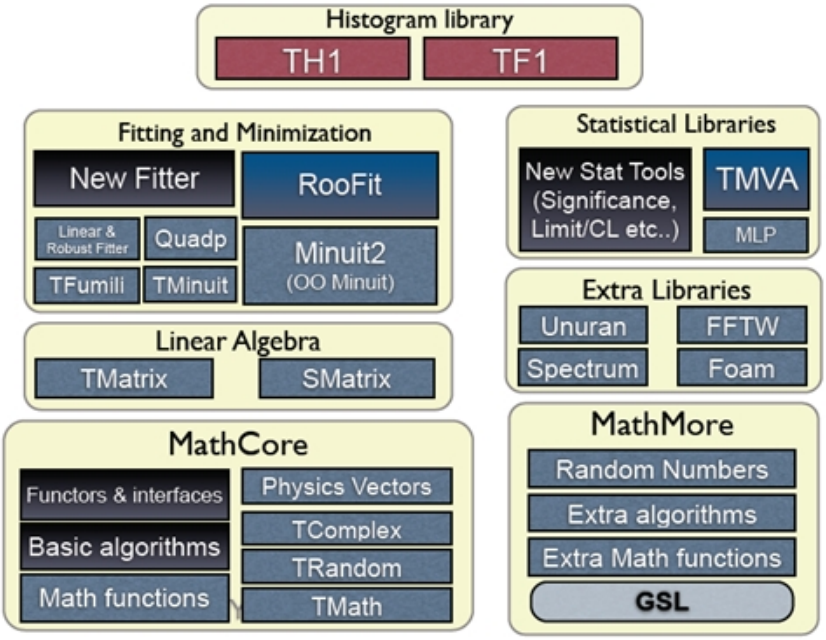


Figure 3.5: Math libraries and packages in ROOT [46].

These techniques, most of which are described in Section 6.1, include methods such as the maximum likelihood method, applied for unbinned datasets; the least squares method, applied for binned datasets; or the Poisson likelihood method, applied for Poisson-described binned datasets.

The implementation of ROOT's fitting process is highly modular, and it is composed of five main components: the user function model, the fitting data, the fitting method, the fitter and the minimizer.

The signature of the model function provided by the user should be compatible with the parametric function interface callable operator in ROOT, and it is provided via a function class in ROOT, e.g. `TF1` (Section 5.3 provides an example on how to adapt a function to the TF1 interface). To allow that the fitting method function classes interface with it, the ROOT class representing the model function needs to be wrapped in a class that implements the parametric function interface. The type of the data to fit, also specified by the user, is identified and described either with the unbinned data or the binned data class representations that ROOT implements. The appropriate objective function class is then built with the wrapped parametric function class object and the fit data representations.

The fitter is the class that manages the fitting process and interfaces the objective function with the minimizer class. It is also responsible for setting all the control parameters and options of the fit, such as allowing the selection of the appropriate minimization algorithm at runtime depending on the complexity of the problem.

ROOT implements interfaces to several minimization algorithms and classes: numerical minimizers, such as Minuit, Minuit2 and Fumili; stochastic minimizers, such as the GSL minimizer based on simulated annealing; and even minimizers based on genetic algorithms.

This modular design allows the reusability of the model, e.g. with different types of data sets or functions, but increases the overall complexity of the process. ROOT provides a way to hide this complexity from the user: calling the fitting process via the Fit method implemented on the most important ROOT object classes, such as histograms, graphs and trees (Listing 1, Figure 3.6). More information about the fitting, as well as the modifications implemented to parallelize the fitting process, both at task-level and at data-level, are provided in Chapter 6.

```
1 TF1 * f1 = new TF1("f1", "gaus");
2 f1->SetParameters(1, 0, 1);
3
4 h1->Fit(f1);
```

Listing 1: Fitting of the histogram `h1` with a Gaussian function in ROOT by calling its member function `Fit`.

RooFit and RooStats

ROOT mathematical libraries provide a set of methods, functions and libraries that respond to the needs of the majority of traditional HEP experiments and analyses. However, the LHC experiments demand more powerful advanced fitting and advanced statistical functionality for some of their analyses. For these cases, ROOT distributes with it the RooFit and RooStats packages.

RooFit [56] is a library that allows building more complex models than ROOT's fitting, e.g. combining probability density functions; performing fits, e.g. maximum likelihood, and generating its respective plots; and generating Monte Carlo samples. It is also oriented to large-scale projects. RooFit was designed to be used with ROOT, and it delegates important parts of the underlying functionality to ROOT when possible.

RooStats [41] is a library that extends ROOT functionality with advanced statistical tools, with emphasis on discovery significance, hypothesis testing, confidence intervals, and combined measurements. These tools, based on the RooFit classes for describing probability density functions or likelihood functions, aim to satisfy the requirements of the LHC experiments and, therefore, include the major statistical techniques approved by the Experiment Statistical Committee to perform the aforementioned measurements.

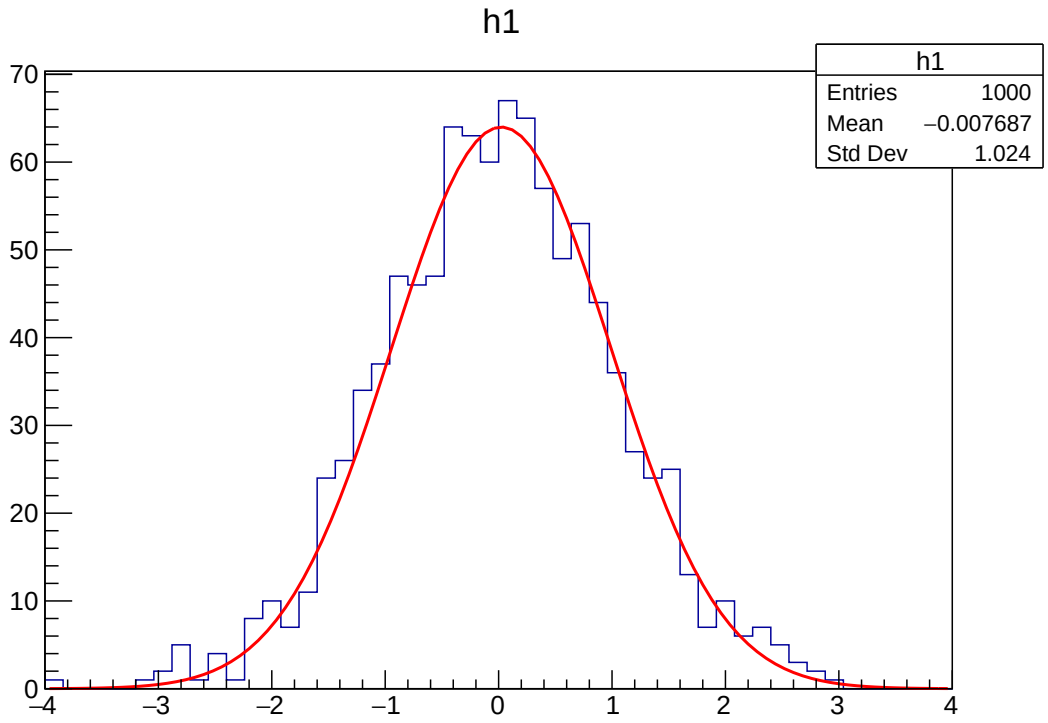


Figure 3.6: Plot result automatically obtained when fitting a histogram with a Gaussian function in ROOT as shown in Listing 1.

3.5 ROOT Data Frame (RDataFrame)

RDataFrame is a programming paradigm available in ROOT that allows its users to express their analyses declaratively in a functional way.

RDataFrame entails a paradigm shift in the way HEP analyses are expressed. By renouncing full control of the event loop, an iterative process over the data recorded from collisions, the user obtains something potentially more powerful: complete control over the expression of the analysis; that is to say, instead of investing time and effort in designing and implementing the details of the analysis computation, the user specifies the different operations that compose the analysis while the framework manages the underlying complexity and optimizations.

RDataFrame hides the complexity of the TTree and the branch-based columnar storage by reasoning in terms of entries and columns. A RDataFrame analysis is composed of actions, such as creating and filling a histogram or computing the mean of the values of a column (a branch in the tree structure of Section 3.2), and of transformations, e.g. filtering data or creating new columns. These operations

are combined and pipelined by connecting them in *functional chains*. Listing 2 compares the traditional way (lines 2-7) of filling a histogram with the entries that pass the IsGoodEntry filter against the same operation expressed with RDataFrame ((lines 11-12), showcasing the chaining of actions and transformations.

```
1  // Traditional programming model
2  TTreeReader reader(data);
3  TTreeReaderValue<A> x(reader,"x");
4  TTreeReaderValue<B> y(reader,"y");
5  TTreeReaderValue<C> z(reader,"z");
6  while (reader.Next()) {
7    if (IsGoodEntry(*x, *y, *z)) h->Fill(*x);
8  }
9
10 // TDataFrame
11 TDataFrame d(data);
12 auto h = d.Filter(IsGoodEntry, {"x","y","z"}).Histo1D("x");
```

Listing 2: Comparison of the traditional programming model against RDataFrame.

While the functional chains were the original idea that sparked the implementation of RDataFrame, it soon became clear that it was more beneficial to express the analysis as a computational graph. Expressing the analysis by means of functional chains requires us to execute all actions and transformations in a single loop over all entries. In addition, expressing it in terms of a functional graph allows us to potentially increase the benefits obtained from a lazy execution of the loop: the loop is executed when accessing an end node of the graph or chain, another feature of RDataFrame. That is to say, given several end nodes of the graph, the loop will be executed only once for their shared actions and transformations, rather than evaluating the whole loop for each functional chain. Section 7.2 details this lazy evaluation, providing an example of a functional graph.

These optimizations of the functional graph require the user to delegate control of the event loop to RDataFrame. Traditional approaches offer more detail and customization possibilities for the computation of the loop, at the cost of increased complexity, unavoidable boilerplate code, and non-trivial parallelization. Instead, RDataFrame emphasizes the programming model, by offering high-level features such as better expressivity, language economy and, most importantly, the abstraction of complex operations, e.g. developing parallel code.

Thanks to Cling, we can further simplify the programming model, improving the expressivity and economy of the language by JIT-compiling, translating simplified user interfaces into fully templated code at runtime. Listing 3 showcases two formulations of the same expression. While both are available, it is usually more convenient for the user to leverage the compact formulation (lines 11-12), at the cost of including minimal overhead by the runtime compilation.

```
// Traditional programming model
TTreeReader reader(data);
TTreeReaderValue<A> x(reader,"x");
TTreeReaderValue<B> y(reader,"y");
TTreeReaderValue<C> z(reader,"z");
while (reader.Next()) {
  if (IsGoodEntry(*x, *y, *z)) h->Fill(*x);
}

// TDataFrame
TDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"}).Histo1D("x");
```

Listing 3: Comparison of the compact and fully expressive formulation of a RDataFrame analysis.

Parallelization is built in at the core of RDataFrame, handled through the implicit multithreading (Section 3.3) mechanisms of ROOT, and will be employed when possible, from the parallelization of the event loop at entry level to the parallel writing of the data frame on disk.

RDataFrame works not only with the ROOT format but with others as popular and important as CSV and Apache Arrow [6], providing an adapter to read and convert these formats into the ROOT format.

RDataFrame's implementation respects modern C++ best practices and aggressively adopts template metaprogramming, which allows it to benefit from improvements such as the absence of virtual calls or type-safe access to the datasets. In addition, ROOT's Python library, PyROOT, has been considered during each stage of the design of RDataFrame, in order to provide seamless integration that paves the way for the interaction with some of the (currently) most important data science libraries, such as NumPy [54], Pandas [38] or TensorFlow [1].

All the benefits obtained from RDataFrame's declarative programming model, along with the exceptional performance exhibited, triggers its fast adoption by HEP's most important experiments.

Section 7.2 describes how by porting a highly optimized, ad-hoc, parallel code to RDataFrame we can not only match the speed up gains obtained by the original analysis but also improve scaling both in commodity laptops and massive parallel architectures.

3.6 User Interfaces and visualization

One of the most important features of ROOT, together with the interpreter and the highly performant I/O, the one responsible for ROOT's fast adoption by the community, is the development of a comprehensive set of visualization tools for HEP data analysis results and objects.

For simplicity and fast exploration of the results, any class inheriting from TObject has a Draw method, which will generate a graphical representation of the caller object in a ROOT canvas, the area mapped to a system graphical window. This automatically-generated graphical representation is implemented as well in most of the ROOT methods generating end results, such as the `Fit` method of a histogram.

In addition, users may request full control over the graphical representation of their results, either for customization or to improve the default visualization. For this purpose, ROOT allows the user to define and generate the components of a plot programmatically, exposing all the different components of the visualization in a set of classes and, therefore, providing high customization and configuration of the graphic results (e.g. Figure 3.7). This is especially useful in combination with the interpreter, whose interactivity and visibility into the state of a program is ideal for the exploration and development of incremental improvements.

Figure 3.8 showcases some of the graphical representations, both in 2D and 3D, available in ROOT.

The Graphical User Interface

ROOT is distributed with a Graphical User Interface (GUI), a user-friendly way to interact with ROOT objects that is very rich in features, offering real-time interactivity and modification of the components of a canvas (possible thanks to the interactive interpreter); automatic generation of the C++ source code for the modified objects; and exploration of the contents of a ROOT file. Figure 3.9 showcases the different context menus obtained when selecting different components on a ROOT canvas, and the actions allowed on them directly accessible through the GUI, without writing code.

ROOT also offers a development toolkit, the ROOT GUI builder, a set of ROOT classes used to develop internal components of the GUI. This includes context menus, the fit panel (Figure 3.10) or the TBrowser, a class for browsing ROOT files. The ROOT GUI builder is also used to develop external applications, such as event displays (Figure 3.11) or experiment-specific visualization tools, e.g. AliEVE [53].

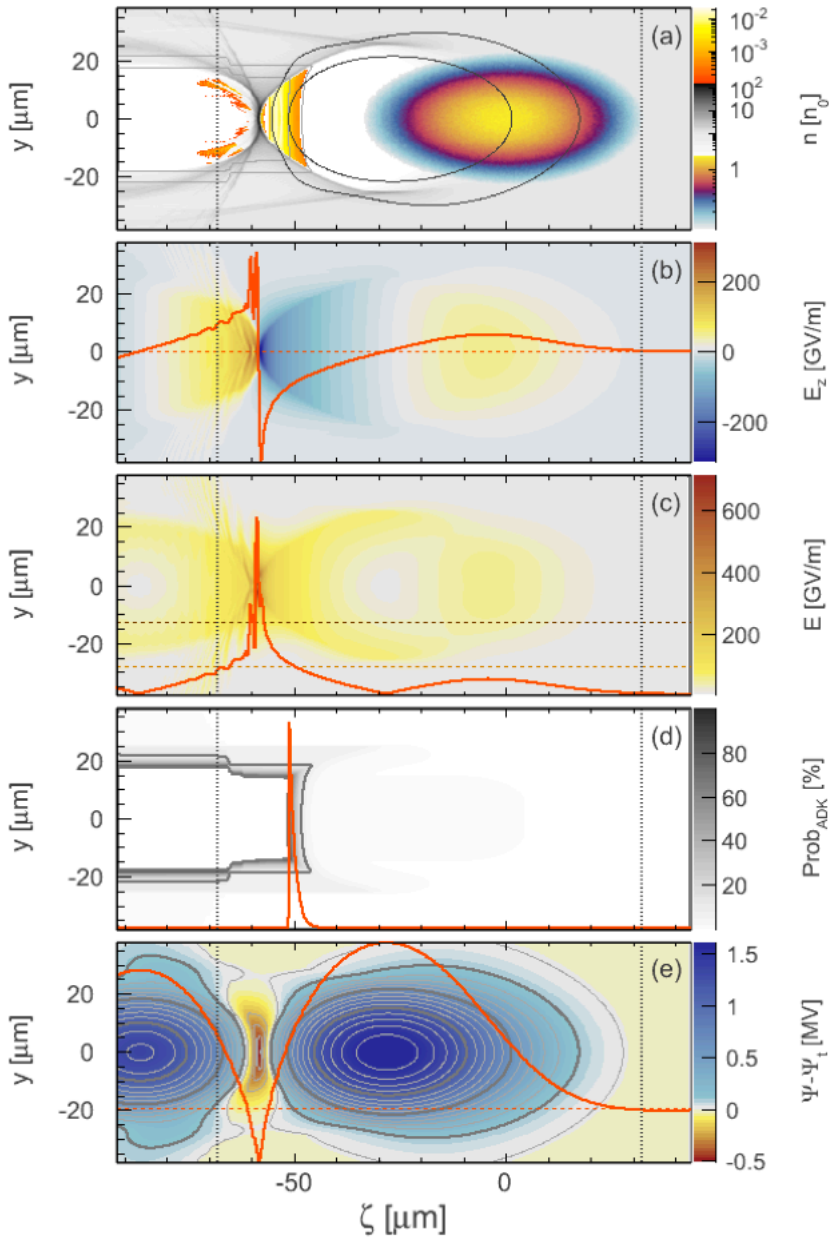
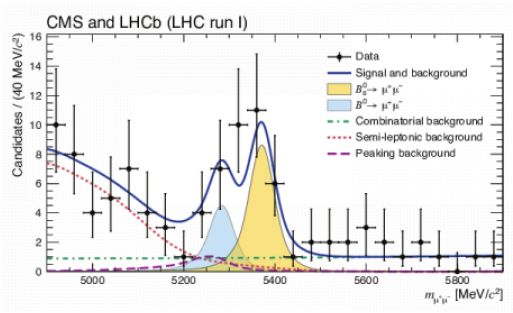
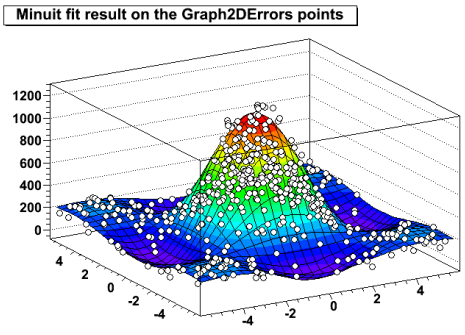


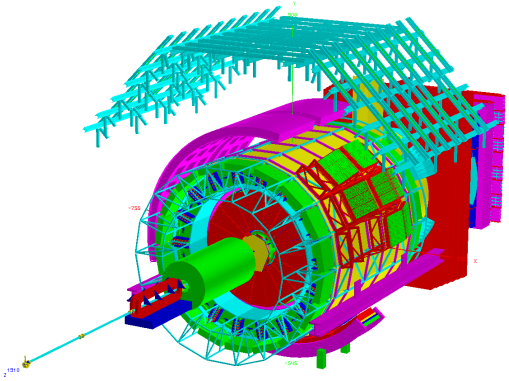
Figure 3.7: Example of how color palettes and transparency can be combined in ROOT.



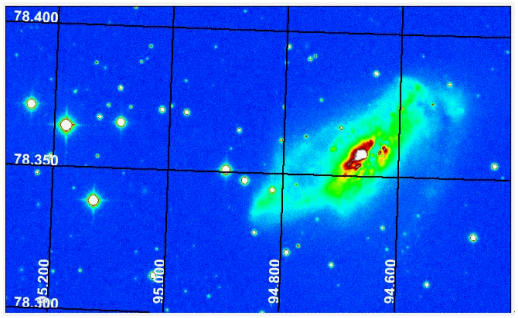
(a) Observation of the rare $B_s^0 \to \mu^+\mu^-$ decay from the combined analysis of CMS and LHCb data



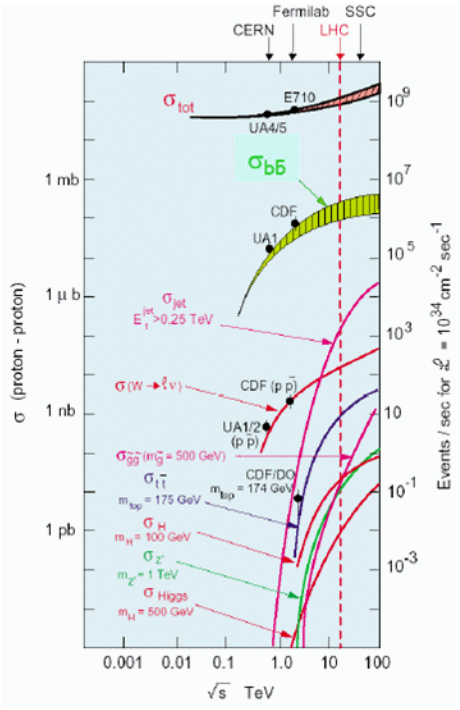
(b) Minuit fit result on the Graph2DErrors points.



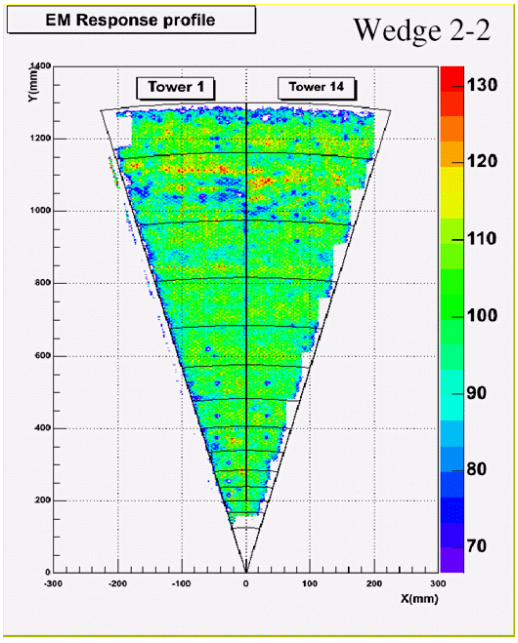
(c) 3D plot of the ALICE detector.



(d) Astrophysics example.



(e) Cross-sections vs. centre-of-mass energy for proton-proton interactions.



(f) Electromagnetic response.

Figure 3.8: Showcase of some of the plots available in the ROOT graphics package. Examples extracted from the ROOT gallery [45].

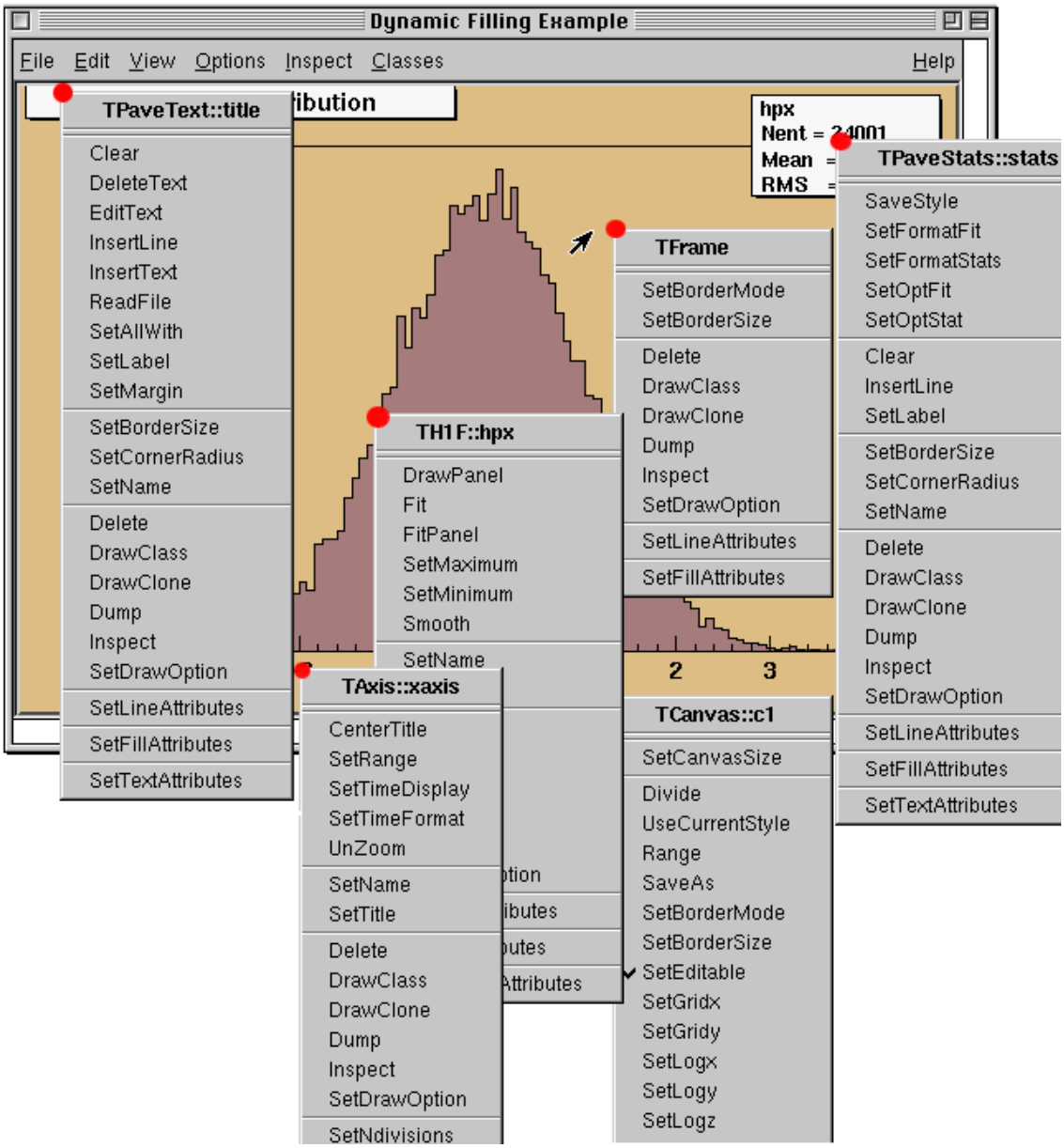


Figure 3.9: Context menus for the objects drawn on the ROOT canvas. Figure from the ROOT User's guide [47].

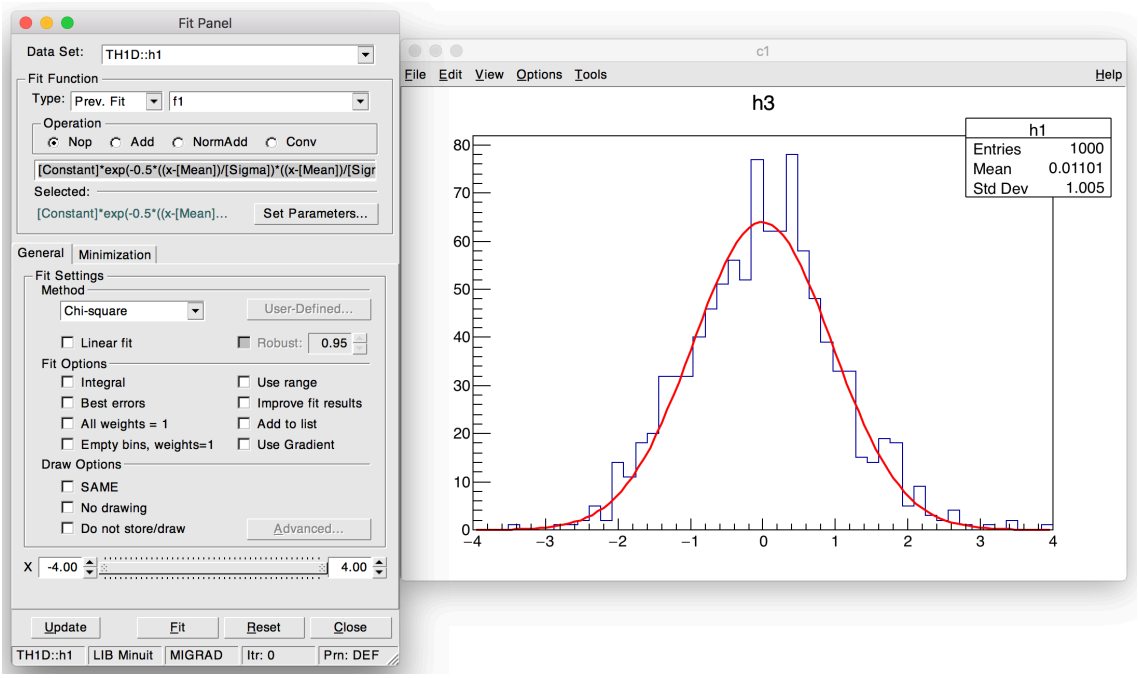


Figure 3.10: The Fit panel.

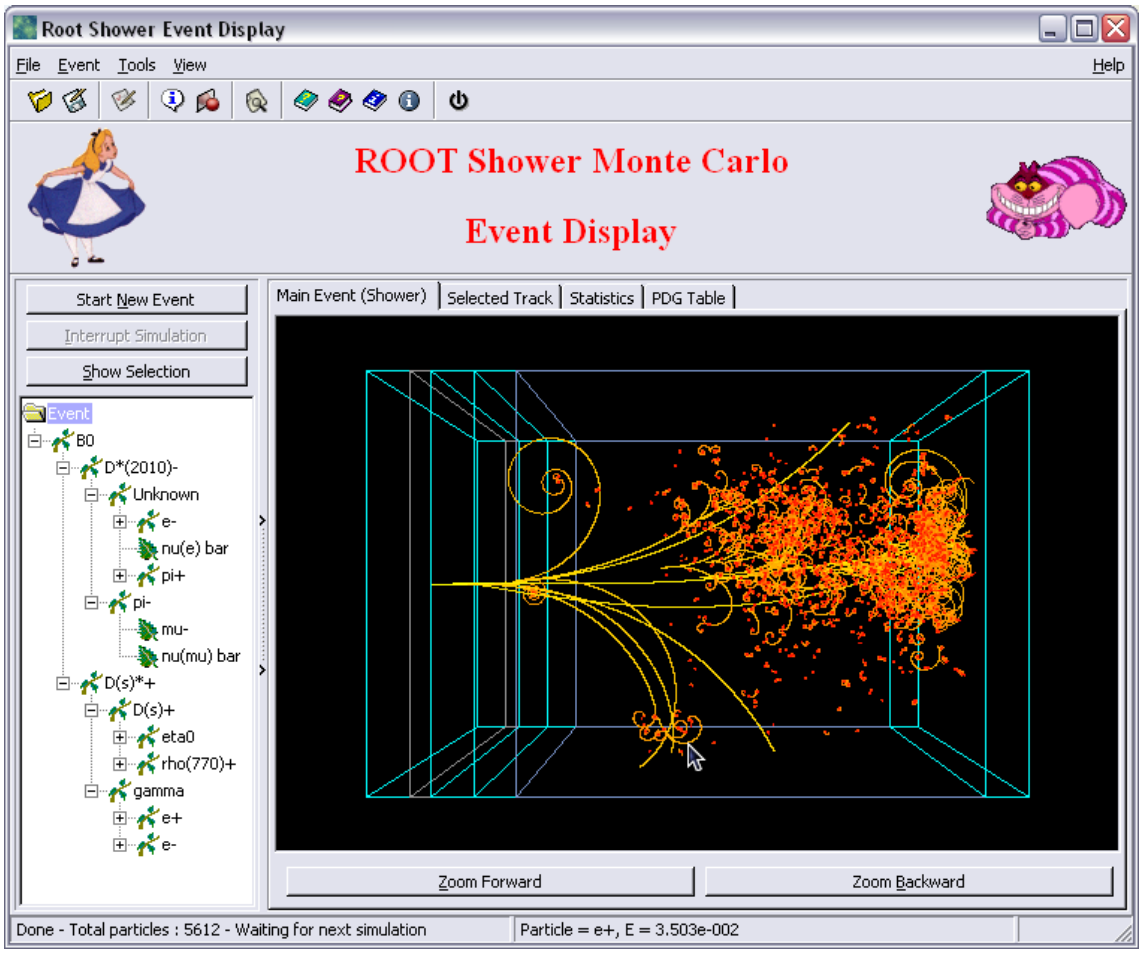


Figure 3.11: Shower Monte Carlo event display built with the ROOT GUI Builder. From the ROOT gallery [45].

Chapter 4

Task-level parallelism

Contemporary HEP analysis requires the processing of large quantities of data. With the breakdown of Dennard scaling in the early 2000s [58] [37], and the subsequent decrease in CPU clock speed, alternative solutions were required to satisfy the demands of modern physics analyses.

Historically, this limitation has been addressed by incrementing the number of processing units, or cores, in a processor. This evolution of hardware towards increasing levels of parallelism drives the adoption of parallel workflows in the data analysis software, across many levels of parallelism[1], to exploit the possibilities offered by modern architectures.

At the same time, many HEP processes can be classified in generic patterns, or algorithms, reproduced over several analyses, adapted to the specific problems at hand. Our goal is to identify these patterns in the Physics analysis processes, and implement in the ROOT software framework a set of tools to parallelize them across distinct levels, from the most common patterns, such as MapReduce, to the ones that provide more potential speed up, such as the integration of tools for vectorization and the adoption of SIMD types. As part of this effort, new generic classes supporting a task-based parallelization and support for different SIMD intrinsics abstraction libraries have been introduced in ROOT.

[1] We define several levels of parallelism depending on the work distributed across several processing units for parallel execution: instruction-level parallelism, parallelizing instructions of the same stream; task-level parallelism, parallelizing tasks of the same application; data-level parallelism, parallelizing multiple data on single instructions; and transaction-level parallelism, parallelizing several transactions.

This Chapter focuses on task-level parallelism. Section 4.1 familiarizes the reader with the event-processing paradigm, a core concept of HEP analyses, and lays out the ideas behind the implementation of task-level parallelism in both multiprocess and multithreaded execution policies.

Section 4.2 introduces TExecutor as the executors' common interface, with a multiprocess implementation in TProcessExecutor (Section 4.3) and a multithreaded implementation in TThreadExecutor (Section 4.4). These executors' interfaces allow the explicit specification of the task's size to potentially improve performance, as analyzed in Section 4.6. Section 4.7 proposes a new executor that combines multiprocessing and multithreading to mitigate the negative performance impact when dealing with non-uniform memory access (NUMA) architectures.

Finally, Section 4.5 justifies the creation of a centralized manager for the pool of workers in the multithreaded case, to avoid undesirable interactions between the parallelization modes in ROOT and the different classes built on top of the TBB scheduler.

4.1 The event-processing paradigm and task-level parallelism

In a typical HEP analysis workflow, the event— that is the data representation of single or multiple piled-up collisions detected and stored for offline analysis—is the minimum unit of data to process. The necessary operations required by the analysis at hand are applied independently and consecutively to each event in a discrete collection, an iterative process that is referred to as the event loop.

Task-level parallelism refers to the partition of the workload into several tasks, or work packages, that are evaluated concurrently. In terms of the event-processing paradigm described above, this results in a division of the event loop into partitions, or blocks of events. These work units typically contain the same number of events, which are then distributed between the different concurrent processing units in the system. The event loop is a natural fit for the task-level approach, as the events recorded by the detectors are many and independent from each other. This makes it easy to encapsulate, in independent tasks, the evaluation of the event loop over subsets of the total recorded events.

Task-level parallelism is implemented in ROOT through the executors, which are ROOT classes implementing the MapReduce pattern [19]—a flexible pattern that appears rather often when processing data composed of independent elements.

Traditionally, task-level parallelism is extracted by dividing the workload in as many blocks as processing units are available in the system. This scheme though is not ideal in terms of workload balancing between the workers when the tasks to

process contain early termination instructions or aggressive branching, which applies to many real use cases.

Instead, we divide the work into smaller tasks and delegate the load balancing to the schedulers of the chosen executor implementation so that we benefit from the mechanisms (e.g. work stealing, greedy reduction) they integrate. By default, we produce as many tasks as data elements are passed as inputs. However, this can create a significant task creation overhead that can impair performance. Therefore we also provide the option for users to chunk themselves the data as appropriate, or to rely on the built-in chunking mechanisms in our interfaces. These three options—the default or the two user-driven approaches—improve over the traditional approach in unbalanced cases and, in addition, accommodate more flexibility in the program design phase. Section 4.6 analyzes the impact that task granularity has on the performance obtained when parallelizing differently balanced examples.

While the approach we adopt in order to exploit task-level parallelism potentially improves performance, it requires a change in how multithreading problems have traditionally been addressed. For example, thread local variables become useless if we allow task interleaving, or adding synchronization directives between tasks would defeat the purpose of creating additional tasks for better load balancing. If these cases cannot be avoided, the traditional approach of dividing the workload is the best option.

4.2 TExecutor: a MapReduce for ROOT

The *MapReduce* programming model expresses problems in parallel by partitioning the input data and distributing it to several workers for processing. During the first step, each one of these workers will then apply transformations on its data partition and return an intermediate result (*Map*). During the second step, all the intermediate results will be combined into a single output (*Reduce*).

TExecutor is the generic interface for MapReduce operations in ROOT. In addition to applying Map and Reduce over input data, `TExecutor` contemplates the case where no data is received as an input but a method must be applied N times and then its outputs have to be reduced, as well as the case where the Map operation does not generate intermediate results (`ForEach`).

In more detail, `TExecutor` offers the following interfaces or methods:

- `Map`: Function that receives as an input the processing method to apply and either input data or the number of times to apply the method. It returns the results in a standard vector.

- `Reduce`: Combination or reduction of the results returned by map into a final result. This operation is usually sequential, but an overload has been provided for parallelizing the case where the reduction function is a binary operator.
- `MapReduce`: Concatenation of the previous two interfaces.
- `Foreach`: A Map that does not generate partial results.

TExecutor has specialized instances of this pattern for multiprocessing (in TProcessExecutor) and thread-based multitasking (in TThreadExecutor).

4.3 TProcessExecutor

Traditionally, multiprocessing has been the common approach to parallelize ROOT due to the thread-unsafe implementation of most interfaces. While there existed several multiprocess solutions such as PROOF [8] and PROOF-Lite, used to parallelize the work with the data model in multicore machines, there was a need for a lighter framework capable of parallelizing more general operations in a transparent way for the user.

TProcessExecutor [22] provides a lighter multiprocessing framework in ROOT, oriented to single nodes, whose interface implements the MapReduce model defined by TExecutor.

Listing 4 shows a basic example of the usage of TProcessExecutor. First we define our map function, which implements the task to be computed in parallel (lines 1-3). Then we propose our reduction function (lines 5-7), accumulating the intermediate result from the map. We initialize TProcessExecutor and the pool of workers (line 9). Finally, by calling `TProcessExecutor::MapReduce` (line 10), we perform the operation in parallel.

```
1  auto mapFunc = [](const UInt_t &i){
2      return i+1;
3  };
4
5  auto reduceFunc = [](const std::vector<UInt_t> &mapV){
6      return std::accumulate(mapV.begin(), mapV.end());
7  };
8
9  ROOT::TProcessExecutor pool;
10 auto r = pool.MapReduce( mapFunction, ROOT::TSeq<int>(100), reductionFunction);
11
12 std::cout<<"Results of the MapReduce operation"<<r<<std::endl;
```

Listing 4: Simple working example of a multiprocessed call to the MapReduce framework.

As a framework, TProcessExecutor improves on the previous approaches to single-machine multiprocessing (PROOF, PROOF-Lite) by forking a ROOT session. Before TProcessExecutor was available, ROOT's multiprocessing performance on a single machine was undermined by the overhead due to the creation of the child processes, with constraints such as the creation of ad-hoc binaries for each of the subprocesses, or the setting of the process environment by passing all the information on scripts and libraries loaded in the client session. Instead, the system call `fork` provides the workers with access to the client's memory, which greatly reduces the amount of information passed; and copy-on-write, which avoids duplicating the memory of the original process until the subprocesses write it.

Thanks to these advantages, creating workers becomes cheaper and faster. Another benefit is flexibility. While PROOF and PROOF-Lite are designed for dealing with the data model, in particular the TTree class, TProcessExecutor can tackle more generic problems thanks to its MapReduce model.

The combination of these new features allows a new kind of workflow in which we are not tied to multiprocessing from start to end. Instead, we can run our program sequentially until it is switched to a multiprocessing mode to perform a MapReduce operation on the data, and then gather the results and continue executing the code on a single core.

As a performance enhancement over the requirements set by TExecutor, the TProcessExecutor's `MapReduce` call implements the greedy worker paradigm. In this operation mode, instead of processing a task and returning the processed result, workers continue drawing tasks from the queue and processing them until all work has been consumed. Then, they perform a partial reduction and send the result to the parent for a final reduction of all the partial results, using ROOT I/O for communication.

4.4 TThreadExecutor

`TThreadExecutor` [12] is a task-oriented, multithreaded MapReduce for ROOT that provides a simple programming model for multithreaded MapReduce operations on decoupled-data based loops. It implements the programming model defined by TExecutor, which means that, as Listing 5 exemplifies, it shares TProcessExecutor's public interfaces.

TThreadExecutor is built on top of Intel's Threading Building Blocks (TBB) library, and in particular, its `parallel_for` and `parallel_reduce` instructions. TBB provides several useful features, such as efficient load balancing, parallel algorithms, and a powerful and configurable interface for task-based multithreading. Unlike alternatives such as OpenMp or POSIX threads, TBB is a cross platform library that

```
auto mapFunc = [](const UInt_t &i){
    return i+1;
};

auto reduceFunc = [](const std::vector<UInt_t> &mapV){
    return std::accumulate(mapV.begin(), mapV.end());
};

ROOT::TThreadExecutor pool;
pool.MapReduce( mapFunction, ROOT::TSeq<int>(100), reductionFunction);
```

Listing 5: TProcessExecutor and TThreadExecutor interfaces are mostly interchangeable.

doesn't require compiler suport and offers support for complex parallel patterns and concurrent datastructures.

TBB implements load balancing by relying on work stealing [28] instead of distributing the tasks evenly among the threads. In this scheme, each thread has a limited task queue that is processed sequentially. The remaining tasks after the initial distribution are kept in a common pool of tasks. If a thread completes the processing of all tasks in its queue, but there still exist tasks in the task pool, it will withdraw them from there and add them to its queue. If the task pool is empty, it will try to steal from the queue of other threads. Finally, if there are no tasks available for stealing, it will remain idle until the other threads complete their work. This mechanism minimizes the time a thread spends idle.

On initialization, TBB creates a pool containing the maximum number of threads supported by the system, which is governed from `tbb::task_scheduler` [29]. This task scheduler creates an implicit task arena [30] where threads may share and execute tasks, allowing the specification of the maximum number of threads to be executed simultaneously. Two tasks schedulers can only co-exist if they have been instantiated from different user threads. This is important, for example, for resource-allocated asynchronous execution and manual load balancing. By default, ROOT treats TBB's task scheduler as a singleton, controlled by TPoolManager (Section 4.5). TThreadExecutor allows for another degree of customization, handling a subset of the threads initialized by the scheduler through a task arena. This allows for concurrent execution of multiple TThreadExecutors by assigning a subset of the total resources to each one.

TThreadExecutor extends the TExecutor interfaces with two useful features:

- **Chunking**: By default, `Map` evaluates the mapping function once per element in the input data. If these data is composed of a large number of elements, and the user does not partition it, this becomes inefficient both spatially (we need to generate a large return `std::vector`) and temporally (e.g, if the mapping

function evaluation is short we have too much task overhead). To address this problem, TThreadExecutor introduces the concept of chunking. The input data is divided into N chunks, specified by the user, and the tasks are evaluated over each one of this chunks—i.e. several data points at a time—applying reduction after each data chunk is processed.

- **Parallel Reduction**: If the user provides a binary function, the reduction step is parallelized by spawning a binary tree with the elements to process, and parallelizing the evaluation starting from the leaves. This interface is only available if the reduction function evaluates on `double` or `float` types in order to reduce complexity adjusting to the ROOT user needs.

Currently, TThreadExecutor is used in ROOT fitting to compute the maximum likelihood and the least squares functions in parallel. In TMVA [24], it is utilized to train BDTs, by parallelizing the loss function; and DNNs, by parallelizing the internal matrix operations. It is also applied for performing implicitly multithreaded operations in ROOT I/O (reading, parallel writing, de-serialization and decompression of tree branches in parallel) and in the parallel execution of functional chains in RDataFrame [23].

4.5 TPoolManager

ROOT provides two modes for expressing thread-based parallelism: implicit and explicit. In implicit multithreading (IMT), users write sequential code using ROOT interfaces and ROOT parallelizes its execution. Many common operations in ROOT adhere to IMT and, while for the moment IMT is not enabled by default, activating this functionality only requires a call to `ROOT::EnableImplicitMT()` (Listing 6).

With explicit multithreading, the user explicitly states which operations should be parallelized and how. ROOT provides several classes for the expression of parallelism, for example, by using locks (e.g. `TMutex`, `TReadWriteLock`, `TCondition`), by employing thread locality mechanisms (`TThreadedObject`), or by applying multithreaded patterns (`TThreadExecutor`).

The user may need a mixed solution for those cases requiring additional flexibility, such as parallelizing interfaces still not available in IMT; or employing user-generated multithreaded code, e.g. via TThreadExecutor, that makes use of the same common TBB scheduler. For reliability, we need to solve the interoperability problems that may arise in these scenarios.

To tackle these problems, we developed the new internal class TPoolManager to handle the TBB library scheduler. It acts as a common interface to TBB's task scheduler for all thread-based parallelization in ROOT, providing shared ownership,

```
// First enable implicit multi-threading globally,
// so that the implicit parallelisation is on.
ROOT::EnableImplicitMT();

// Open the file containing the tree
TFile *file = TFile::Open("tree.root");

// Get the tree
TTree *tree = nullptr;
file->GetObject<TTree>("tree_0", tree);

// Read the branches in parallel.
// Note that the interface does not change, the parallelisation is internal
for (Long64_t i = 0; i < tree->GetEntries(); ++i) {
   tree->GetEntry(i);  // parallel read
}

// IMT can be also disabled globally.
// As a result, no tree will run GetEntry in parallel from now on
ROOT::DisableImplicitMT();

// This is sequential: the global flag is disabled
for (Long64_t i = 0; i < tree->GetEntries(); ++i) {
   tree->GetEntry(i);  // sequential read
}
```

Listing 6: Simplified example from the multicore tutorials in ROOT showcasing implicit multithreading.

reference counting, and automatic destruction and termination when the scheduler is idle.

Due to TBB restrictions, once the TBB scheduler is active, the number of active threads cannot be changed. Because of that, all instances pointing to the TPoolManager scheduler will need to be destroyed before the pool of threads can be initialized again.

TPoolManager grants the following advantages:

- Provides a unique interface to the TBB scheduler: TThreadExecutor, IMT and any class relying on TBB will get a pointer to the scheduler through `TPoolManager::GetScheduler()`, which will return a reference to the only pointer to the TBB scheduler that will be active in any ROOT Process.

- Solves multiple undefined behaviors. Guaranteeing that all classes use the same multithreading scheduler avoids interferences and undefined behavior by providing a single instance of the scheduler and automated bookkeeping, initialization and destruction.

4.6 Analysis of work partitioning granularity in the executors

Dividing the work at thread-level in the traditional way, that is, in as many partitions as processing units there exist, has a drawback: if, for instance, some of the partitions complete their part of the work earlier than others, the corresponding processing units will remain idle, waiting for the remaining ones to finish, and leading to a suboptimal exploitation of the hardware resources. However, we can mitigate the time workers spend in idle state due to an unbalanced distribution of the workload by reducing the granularity of the data partitions.

In this section, we analyze the impact that the number of partitions has on the performance when parallelizing a job with ROOT's executors. Our analysis is executed over different sizes of data and increasing number of processing units, for both TProcessExecutor and TThreadExecutor. In all cases, we increase the number of tasks to execute exploiting the built-in chunking mechanism of these classes, moving progressively from coarse-grained partitions, when the workload is divided in as many partitions as processing units available, to the most fine-grained division, with one task per data element

For the following experiment, we designed a benchmark consisting of the conditional evaluation of two different implementations of the Vavilov probability distribution function—fast and accurate—which provides with a simple way to introduce imbalance in the execution time for the chunks. The fast implementation of the Vavilov probability density function [48] is about 5 times faster for the calculation of the distribution function, while its accurate implementation [50] sacrifices speed for correctness. We perform the evaluation of the Vavilov fast function on the positive values of a given data set, and the Vavilov accurate evaluation on the negative values of the same data set. Listing 7 displays the code to perform our benchmark with an explicitly chunked MapReduce operation with TThreadExecutor, given a container of data, `data`, and a number of chunks, `nChunks`.

This benchmark was executed on a Intel Core i7-4790 desktop server with 4 physical cores (8 logical threads) at 3.6 GHz and 8 GB of RAM [26], each core supporting hyperthreading, over two different data sets, both of them featuring unbalanced executions. The first data set is generated randomly following a Gaussian distribution with $\sigma = 1$, $\mu = -0.25$ for the first third of data elements; $\sigma = 1$, $\mu = 0.25$, for the second third; and $\sigma = 1$, $\mu = -0.655$ for the remaining third. The measurements were timed with the C++ standard high resolution clock, in several consistent executions with low variation, and then averaged and used to compute the speedup with respect to the sequential execution time measured.

Figure 4.1 displays the speed ups obtained for different sizes of the data set when using a different number of threads (including hyperthreading) in TThreadExecutor,

```
// (... variable initialization: data, nChunks)

ROOT::Math::VavilovAccurate vavilovAc(0.3, 0.5);
ROOT::Math::VavilovFast vavilovFast(0.3, 0.5);

auto mapFunction = [&data, &vavilovAc, &vavilovFast](const unsigned i){
   return data[i]>0? vAccurate.Pdf(data[i]) : vFast.Pdf(data[i]);
};

auto redFunction = [](const std::vector<double> vec){
   return std::accumulate(vec.begin(), vec.end(), .0);
};

ROOT::TThreadExecutor  threadPool(nThreads);
auto res = threadPool.Map(mapFunction, ROOT::TSeqU(data.size()), redFunction, nChunks);
```

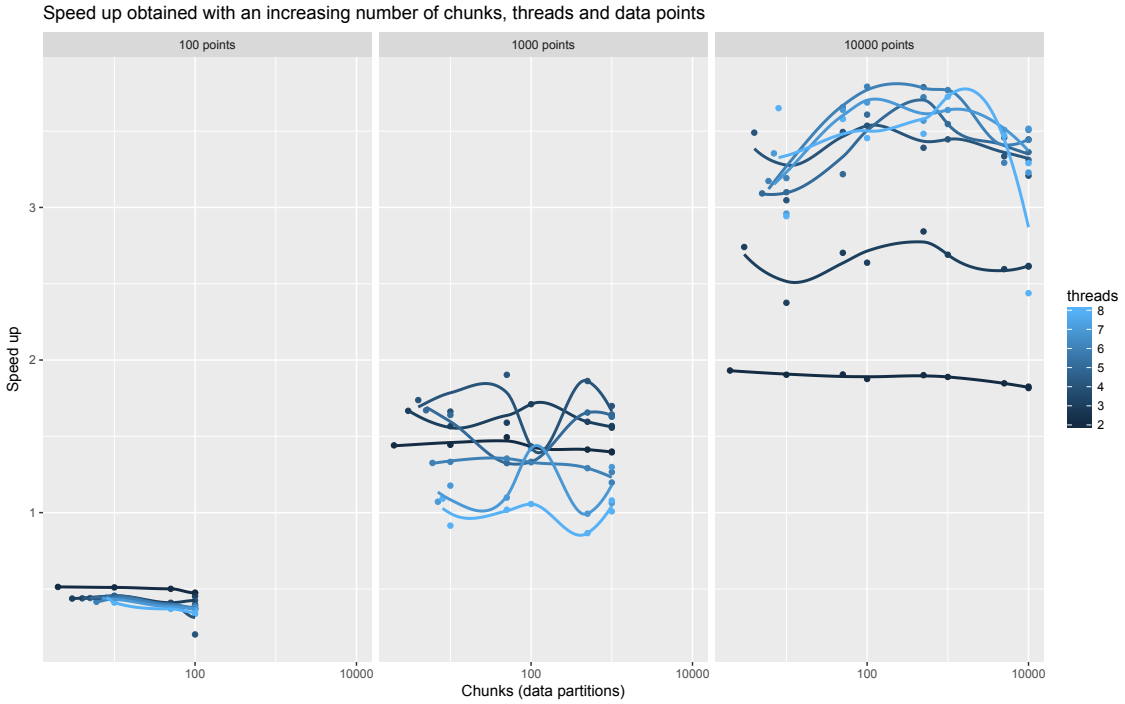
Listing 7: Example code of our benchmark computing the Vavilov probability density function in its fast implementation for negative values of the data, and its accurate implementation for positive values of the same data.

increasing the number of data partitions in each case. In Figure 4.1a we can observe that, for 100 points, the execution time is dominated by the task creation overhead, causing a slowdown in the total execution time, independently of the number of data partitions. The performance improves as we increase the dataset size until it stabilizes, as shown in Figure 4.1b. There we observe significant speed ups, which improve slightly until, again, the task overhead negatively affects the execution time when dealing with smaller tasks.

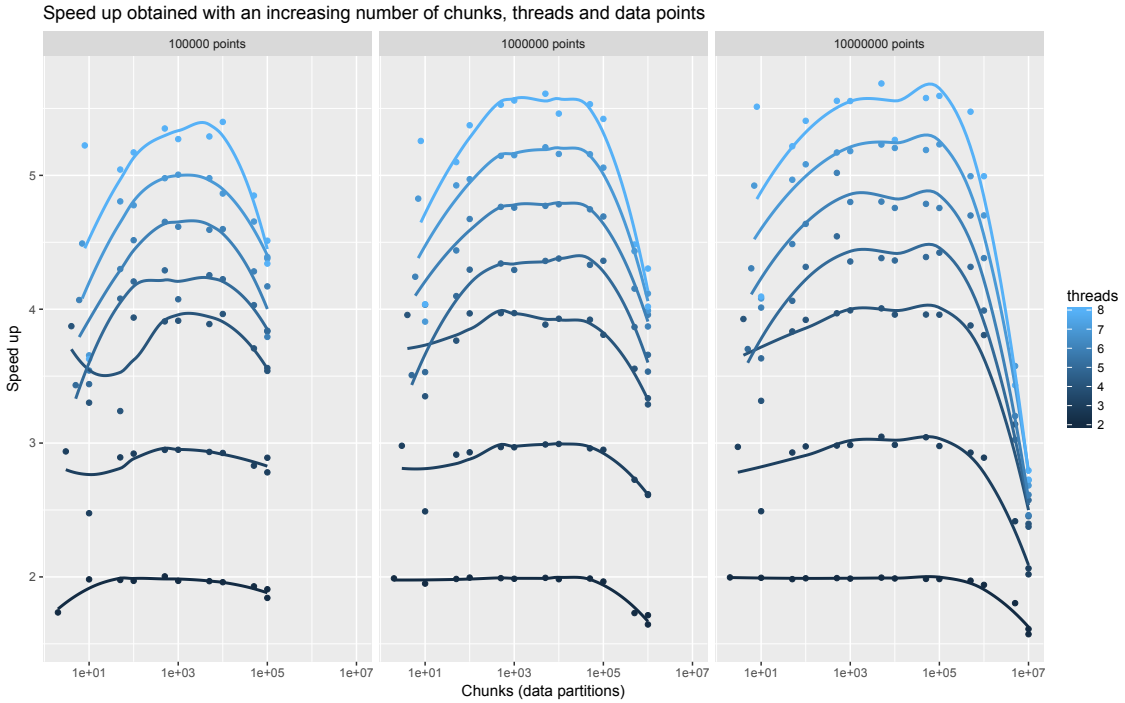
This example exhibits higher performances when exploiting hyperthreading. With this option enabled we observe a gradual, more pronounced improvement in performance, reaching up to a remarkable 45% higher speed up than its non-hyperthreaded counterpart. However it exhibits a more sharp decline in speed up when the task creation overhead starts to be noticeable.

Figure 4.2 shows the results obtained for a multiprocessing execution with TProcessExecutor of the same example. While a multiprocessing execution is more useful in several contexts (e.g. when performing non thread-safe operations), and the results display a fair speed up for big data sets, the task overhead introduced by TProcessExecutor is several orders of magnitude larger than that of TThreadExecutor, requiring a greater amount of work for the acceleration to be noticeable. For this same reason, the plots in Figure 4.2 exhibit an earlier decline of performance with a higher number of tasks.

The near-optimal speed up we obtain in the theoretically most unbalanced case (dividing the workload into as many tasks as processing units), seems to indicate that the data set may not be unbalanced enough to observe dramatic improvements when tuning task granularity. We observe this situation in Figure 4.3, produced using Intel's Vtune Amplifier 2018, where we visualize the performance trace of an



(a) Speed up for $8 * 10^2$, $8 * 10^3$ and $8 * 10^4$ data points.



(b) Speed up for $8 * 10^5$, $8 * 10^6$ and $8 * 10^7$ data points.

Figure 4.1: Speed up obtained with TThreadExecutor for different number of data points generated for the first benchmarking data set, divided into an increasing number of chunks. The measures have been taken with different number of threads.

execution of our benchmark with 800 million points, 4 processing units, and 4 tasks for both TProcessExecutor and TThreadExecutor. The green color in the timeline indicates process running time, the brown graphs represent the CPU time, and the red spikes correspond to spin and overhead time, indicating synchronization areas. Figure 4.4 shows the trace of an execution with the same parameters but with an increase in the number of tasks to 8000, demonstrating an improvement in workload balancing.

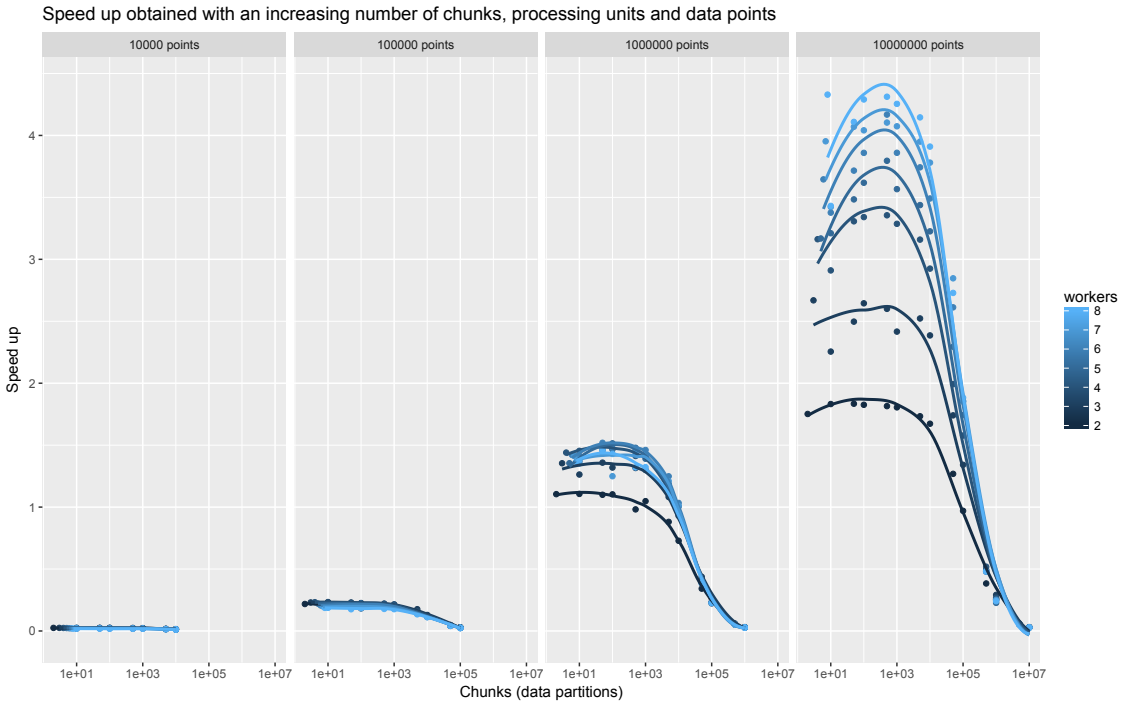
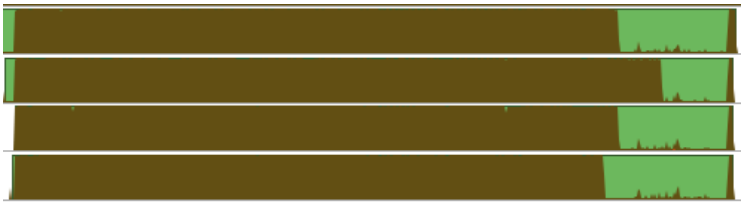


Figure 4.2: Speed up obtained with TProcessExecutor for different sizes of the data and different number of chunks when benchmarking the first dataset. The measures have been taken with different number of processing units.

The second data set, which is more extremely unbalanced, is composed of two distinct parts: the first two thirds of the data set evaluate the fast version of the Vavilov distribution, and the remaining third performs the accurate evaluation. Figure 4.5 demonstrates that this case exhibits a more accentuated imbalance among the processing units. The orange trace in Figure 4.5a represents the spin (time spent in locked state) and overhead time of the main thread, after completing the execution of its assigned tasks and while it waits for the remaining threads to complete their work. Again, in Figure 4.6 we provide a better balanced example of the same execution, achieved by increasing the number of tasks to 8000.

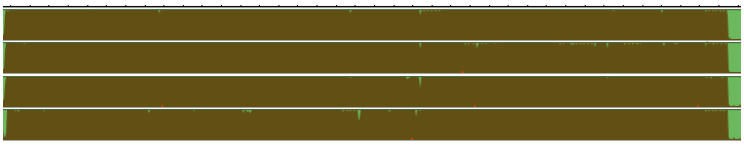
(a) Multithread execution with TThreadExecutor.



(b) Multiprocess execution with TProcessExecutor.

Figure 4.3: Thread monitoring for sample size of $8 \cdot 10^7$ points generated for the first data set, divided into 4 partitions ($2 \cdot 10^7$ points per chunk) and evaluated by 4 threads.

(a) Multithread execution with TThreadExecutor.



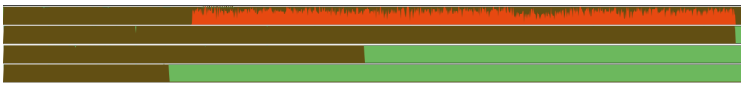
(b) Multiprocess execution with TProcessExecutor.

Figure 4.4: Thread monitoring for a sample size of $8 \cdot 10^7$ points generated for the first data set, divided into 8000 partitions (10^4 points per chunk) and evaluated by 4 threads.

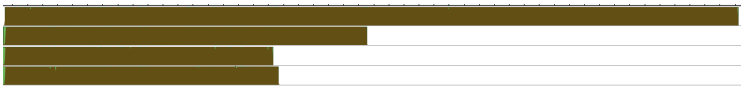
Figures 4.7 and 4.8 present the results of executing our benchmark with this second data set. For the multithreaded and the multiprocessed cases, they respectively display the increase in speed up when varying task granularity in different configuration pairs of number of threads and size of the workload. In this case, we observe an earlier steeper increase in speed up for the higher values of the chunk size, indicating the convenience of splitting tasks into smaller partitions. In the last configuration of Figure 4.7b, we perceive a gradual decrease in speed up caused by excessive fine-graining of the tasks, yielding an execution time that is dominated by

the task overhead. Figure 4.8 exhibits, again, an earlier decay of the speed up due to the higher task overhead inherent to a multiprocess solution.

These results support the importance of an adequate task size and showcase the impact it has on parallel performance for unbalanced workloads. Moreover, they indicate that granularity has to be treated differently for the multithreaded and the multiprocessed cases, keeping task creation overhead in mind.



(a) Multithread execution with TThreadExecutor .



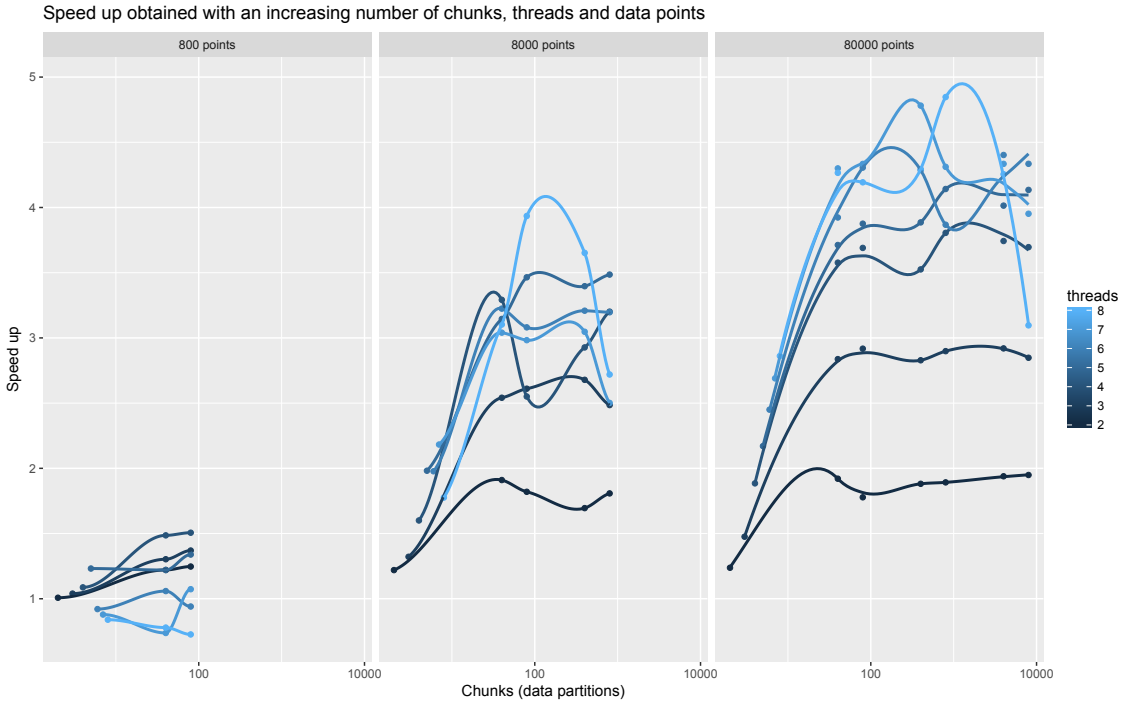
(b) Multiprocess execution with TProcessExecutor.

Figure 4.5: Thread monitoring for sample size of $8 \cdot 10^7$ points generated for the second data set, divided into 4 partitions ($2 \cdot 10^7$ points per chunk) and evaluated by 4 threads.

(a) Multithread execution with TThreadExecutor.

(b) Multiprocess execution with TProcessExecutor.

Figure 4.6: Thread monitoring for a sample size of $8 \cdot 10^7$ points generated for the second data set, divided into 8000 partitions (10^4 points per chunk) and evaluated by 4 threads.



(a) Speed up for 10^2, 10^3 and 10^4 data points.

(b) Speed up for 10^5, 10^6 and 10^7 data points.

Figure 4.7: Speed up obtained with TThreadExecutor for different number of data points of the second benchmarking dataset and an increasing number of chunks. The measures have been taken with different number of threads.

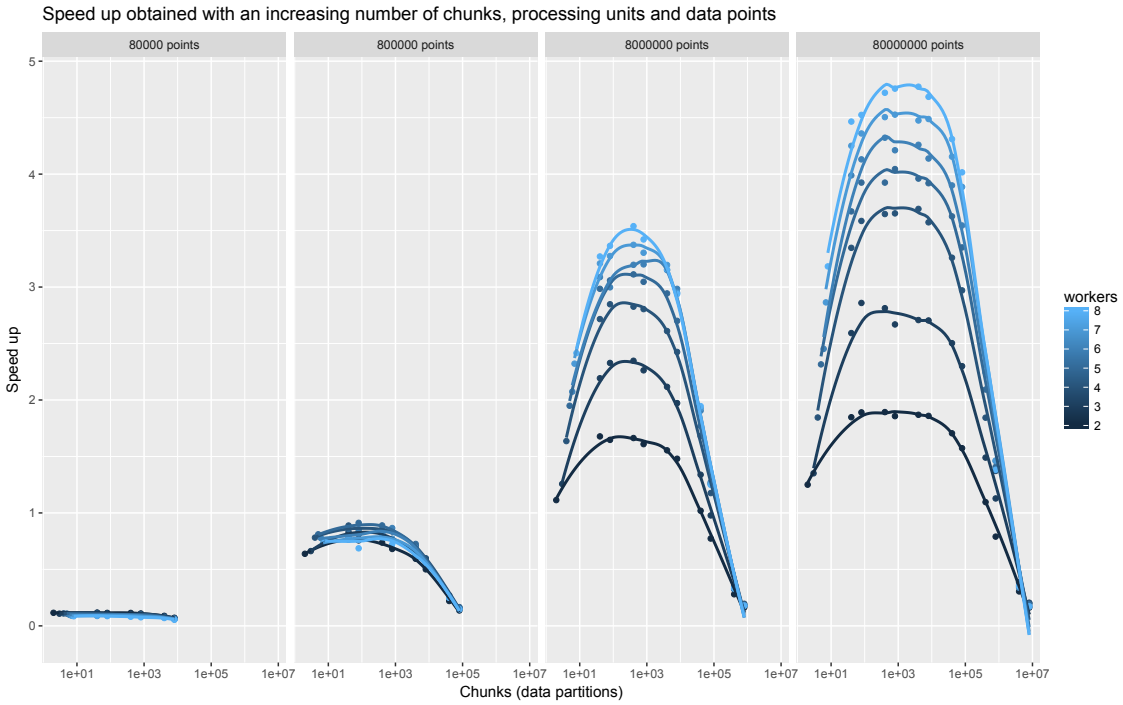


Figure 4.8: Speed up obtained with TProcessExecutor for different sizes of the data and different number of chunks when benchmarking with the second dataset. The measures have been taken with different number of processing units.

4.7 A NUMA-aware executor

For convenience, modern high-end, many-core processors often adopt multi-socket designs, in which each socket is a processor package composed of multiple cores with dedicated physical RAM and integrated memory controllers. Despite the clear advantages of this approach in terms of modularity and scalability, these architectures may exhibit undesirable effects resulting from non-uniform memory accesses (NUMA).

We refer to these undesirable consequences as NUMA effects: negative impacts on performance (e.g. increased latency, bandwidth overload) resulting from remote memory accesses in a heterogeneous memory hierarchy. These negative effects are easily observable in multithreaded operations where there coexist several tasks accessing the same block of memory concurrently from different sockets or NUMA domains.

In ROOT, TThreadExecutor shares a common pool of tasks, where the different threads of the system draw the next item to process once their task queue is depleted. These tasks are most frequently tied to a partition of the data, where the mapping function will be applied. The data may be stored in one of the NUMA domains or shared between them, depending on the size and the locality of the writing or reading policy. Frequently, threads are responsible for the creation of tasks which require accesses to memory of a remote domain. This memory is brought to the shared cache of the socket and fetched by the higher-level caches for faster accesses to data. If several misses on the last-level cache, shared between threads, occur, this will cause frequent accesses to remote memory and, therefore, may overload the interconnection.

In addition, we cannot currently pin tasks to specific threads. Even if we could, it is not possible to assert that this would be the optimal solution. Threads can execute in any of the processors (in the case of TBB, they may even switch execution between processors dynamically), and it might happen that data is moved from one NUMA node to another in order to increase the locality access and, as a consequence, decrease aggregate bandwidth usage, latency, and NUMA effects overall.

To address the common problem of NUMA effects, we have developed a new NUMA-aware hybrid multiprocess-multithread executor for handling multithreaded operations in NUMA architectures. First, we split the MapReduce processing, by means of TProcessExecutor, into as many partitions as NUMA domains exist in the system, and we distribute the work among them. The execution of each partition is restricted to a set of cores belonging to the same NUMA node; that is, each processing unit execution is confined to a NUMA domain. Each task will then be split following the typical multithreaded strategy with TThreadExecutor. Each TThreadExecutor internal to the TProcessExecutors' tasks generates its own independent pool of threads that are only able to execute tasks in the NUMA node to

which they are confined.

Libnuma [5] is the library that manages the restricted execution of processes in a NUMA node. In our case, we employ its `numa_max_node()` directive to obtain the number of NUMA nodes in the system, and the `numa_run_on_node(int i)` and `numa_run_on_node_mask(numa_all_nodes_ptr)` directives to modify the scope of the execution.

Listing 8 displays a code snippet extracted from the TNUMAExecutor class, implementing the process described above. Lines 15 and 16 present the familiar instantiation and execution of TProcessExecutor, with `fNDomains = numa_max_node()+1` worker units and `fNDomains` tasks (described by the sequence `TSeq(fNDomains)`) that execute the C++ lambda `runOnNode` implemented in lines 5-13. This lambda receives the task index (the same as the domain index in our case) and commences by restricting its execution to the corresponding domain (line 6). This restriction is lifted in the last instruction of the lambda before returning (line 11). Lines 7-10 instantiate and execute a NUMA domain specific TThreadExecutor, allowing the specification of the chunking granularity.

```
1  template<class F, class R>
2  auto TNUMAExecutor::MapReduce(F func, unsigned nTimes, R redfunc,
3                                unsigned nChunks) -> typename std::result_of<F()>::type
4  {
5     auto runOnNode = [&](unsigned int i) {
6        numa_run_on_node(i);
7        ROOT::TThreadExecutorImpl pool{fDomainNThreads};
8        auto res = nChunks?
9                      pool.MapReduce(func, nTimes, redfunc, nChunks/fNDomains) :
10                     pool.MapReduce(func, nTimes, redfunc);
11       numa_run_on_node_mask(numa_all_nodes_ptr);
12       return res;
13    };
14
15    ROOT::TProcessExecutor proc(fNDomains);
16    return proc.MapReduce(runOnNode, ROOT::TSeqU(fNDomains), redfunc);
17 }
```

Listing 8: Example of the interaction of libnuma, TProcessExecutor and TThreadExecutor, corresponding to TNUMAExecutor's N-execution MapReduce overload.

To benchmark TNUMAExecutor, we performed an unbinned fit (Listing 9) of the diphoton invariant mass distribution resulting from a Higgs boson, with an analytically computed integral. This parallelization analysis is performed in Section 6.5, but here we focus on the monitoring and potential decrease of the NUMA effects. To augment the probability of observing uncore events [57] (i.e. events that occur outside of the processor such as remote memory accesses), we fit an amount of data that doubles the size of the last-level cache of a NUMA domain. This way we are

more likely to observe cache misses at the last-level cache of a NUMA node, as data must be fetched from memory which is potentially stored in another domain.

This fit is benchmarked on a Intel Xeon E5-2683 v3 [32] server composed of two NUMA domains with 14 physical cores and 28 logical threads each at 2.00 GHz with 32 GB of RAM per NUMA node. Each NUMA domain of the system has a last-level cache of 35 MB, which implies that fitting over 9.5 million (double precision) data points should be sufficient to provoke a considerable number of NUMA effects.

```
1  double func(const double *data, const double *params)
2  {
3      //... Model function
4  }
5
6  // Enable implicit multithreading
7  ROOT::EnableImplicitMT();
8
9  TF1 f("model", func, 100, 200, 4);
10 f.SetParameters(1, 1000, 7.5, 1.5);
11 ROOT::Math::WrappedMultiTF1 wF(f, f.GetNdim());
12 ROOT::Fit::UnBinData dataSB(nPoints);
13 for (unsigned i = 0; i < nPoints; ++i) dataSB.Add(f.GetRandom());
14
15 ROOT::Fit::Fitter fitter;
16 fitter.SetFunction(wF, false);
17 fitter.Fit(data, 0);
```

Listing 9: Implicitly parallel unbinned fit of a model function.

In an attempt to observe and quantify NUMA effects, from all the uncore events available, we choose to monitor and record, with 1 second samples, the DIMM [59] page misses in the DDR memory controller; that is, the number of requests to remote RAM memory. Running the benchmark with 14 threads to fit 9.5 million data points, we notice that TNUMAExecutor exhibits a considerable improvement with respect to fitting via the plain TThreadExecutor. Figure 4.9 compares the number of remote memory accesses monitored with and without the NUMA executor, by displaying the distribution density of these accesses. We can infer that our choices of benchmark and uncore events were appropriate as, by using TNUMAExecutor, we observe a 28.09% reduction in the average number of page misses in the memory controller, and we obtain a 1.89 times improvement in the execution time (Table 4.1).

Unfortunately, due to the implicit use of multiprocessing in TNUMAExecutor, we cannot simply replace TThreadExecutor by TNUMAExecutor. If so, these promising results would be hindered by the overhead introduced by TProcessExecutor. In Figure 4.10 we analyze the convenience of using TNUMAExecutor for an increasing number of threads of the system, and an increasing amount of data points. By reducing the problem and performing single evaluations of the objective

	TThreadExecutor	TNUMAExecutor	**Improvement**
Execution time (s)	465.16	246.43	**1.89**
Accesses to remote memory/s	27.66	19.89	**1.39**

Table 4.1: Comparison of the execution times before and after substituting TThreadExecutor by TNUMAExecutor and speed up in the later case.

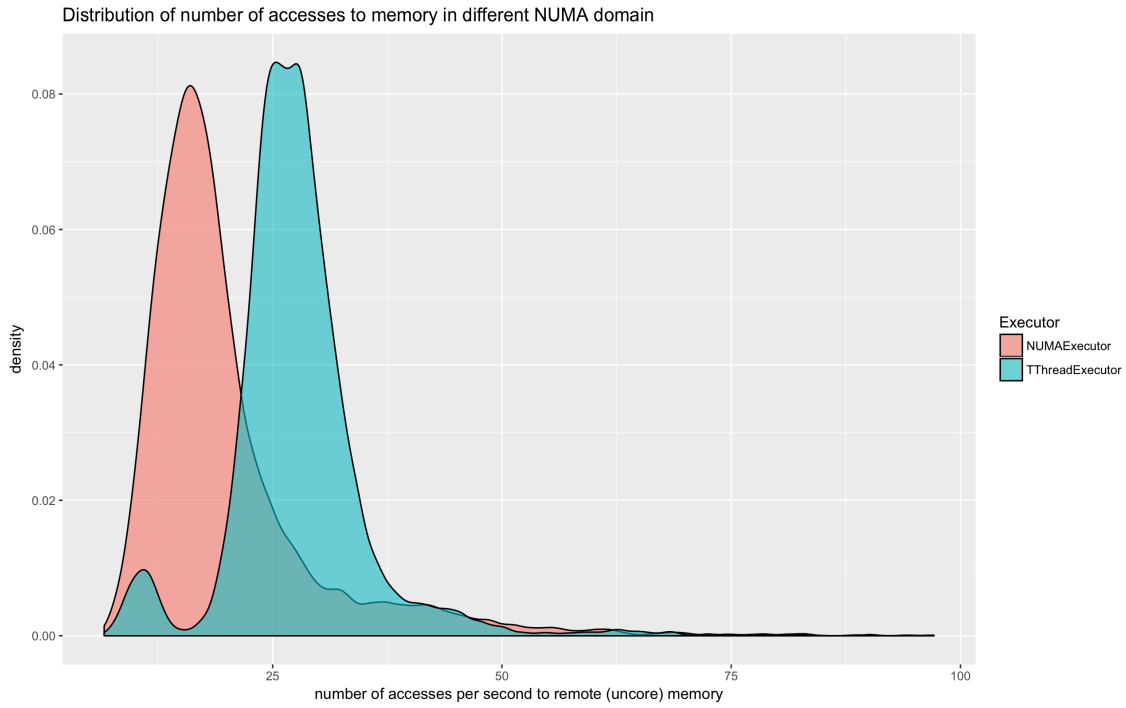


Figure 4.9: Distribution density of the number of accesses to memory controllers to retrieve data from a different NUMA domain.

function, in this problem the maximum likelihood (Equation (6.5)), instead of a complete fitting minimization process, we conclude that it is only convenient to rely on TNUMAExecutor when the size of our data is of the same magnitude of the last level cache, 35 MB, in this case $O(10^6)$. Finally, for large datasets, we notice a large gap in speed up with respect to TThreadExecutor between the hyperthreaded and the non hyperthreaded case. This indicates that employing hyperthreading helps to mitigate the observed NUMA effects in this case when employing only TThreadExecutor.

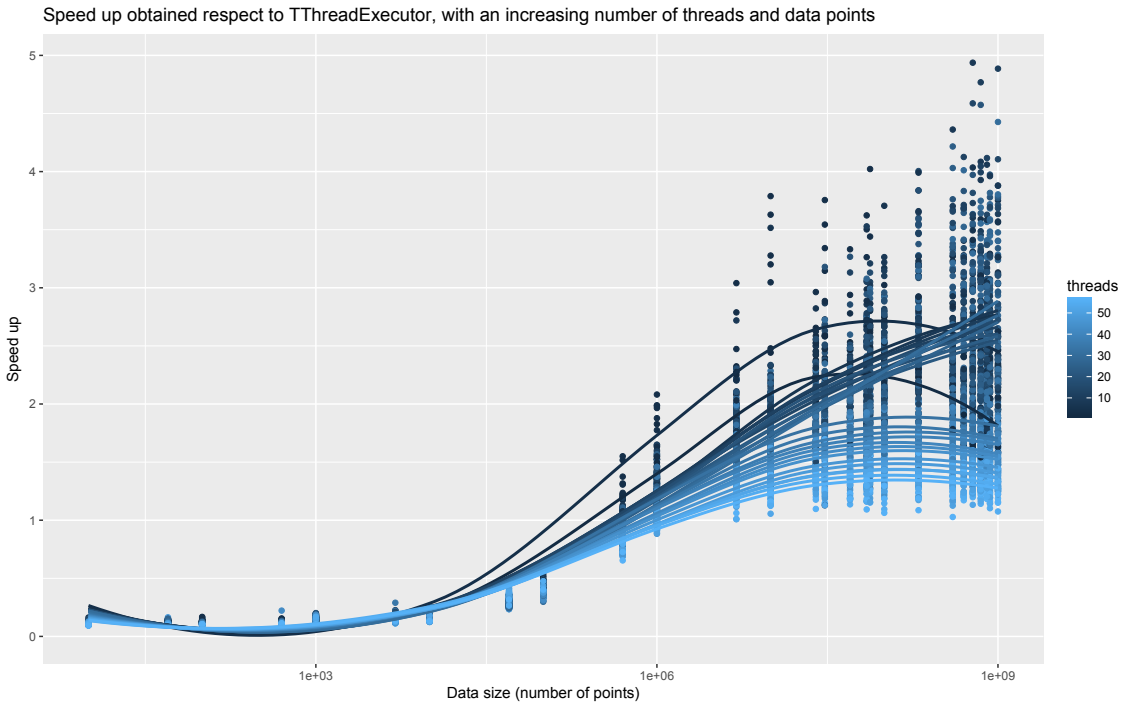


Figure 4.10: Speed up with respect to the TThreadExecutor execution, obtained by employing TNUMAExecutor for different number of threads and an increasing size of data points to evaluate the maximum likelihood on.

4.8 Conclusions

This chapter introduces the executors, a performant set of tools to exploit task-level parallelism in ROOT-based analyses. The executors are ROOT classes implementing the parallel MapReduce, identified as one of the most frequent patterns in ROOT analyses.

These easily-deployable classes provide a set of common interfaces that hide the complexity of concurrent programming from the user. We factored out the MapReduce functionality of TPool, the precursor of the executors, into TProcessExecutor, ROOT's multiprocess executor. We then defined and implemented TThreadExecutor, a multithreaded executor leveraging TBB as a backend, and abstracted the executors' common interfaces into TExecutor. The Executors provide good performance and an improved programming model. On the other hand, they require adapting the parallel analysis to avoid synchronization and, in the multithreaded case, thread-local storage and thread race conditions.

We benchmarked and analyzed the executors' performance and demonstrated the benefits derived from tuning task granularity with the chunking interfaces of

TProcessExecutor and TThreadExecutor when dealing with unbalanced workloads. TThreadExecutor exhibits close to ideal performance, higher than TProcessExecutor, which experiences a larger task overhead inherent to a multiprocessed implementation, of $O(10^{-3}$ s) compared to $O(10^{-7}$ s) for TBB tasks in TThreadExecutor. Thanks to its lower task overhead and the work stealing mechanisms provided by TBB, TThreadExecutor exhibits an earlier balancing of the workload when increasing the number of tasks and a later decay in performance. TProcessExecutor showcases good speed up as well, but it requires a larger amount of data to do so.

In addition, with TPoolManager, a single point of management for the TBB pool of threads, we successfully introduced a solution for undefined behaviours arising from interactions between the implicit and explicit multithreaded modes of ROOT.

Finally, we provided an executor for NUMA architectures. By combining TProcessExecutor with TThreadExecutor and concealing the execution of the latter in each NUMA domain via libnuma, we reduce the impact of NUMA effects in performance. This new executor, TNUMAExecutor, diminishes the number of uncore events monitored when dealing with sizeable data, up to a remarkable 30% less when performing a fit in a NUMA machine with two NUMA domains of 14 cores (+14 hyperthreaded). As a result, this example displays a speed up to 1.8x against the baseline TThreadExecutor execution.

Chapter 5

Data-level parallelism

In addition to task-level parallelism, where we distribute the workload between the multiple cores or processors of a system, we can further parallelize the computations performed in each core by taking advantage of data-level parallelism. Modern architectures support different SIMD instruction sets that enable us to capitalize on the larger CPU registers existing in current computers to perform simultaneous operations on multiple data elements.

This chapter describes the work done to introduce data-level parallelism in the mathematical libraries of ROOT. Section 5.1 discusses the concept of vectorization and describes some of its common pitfalls. Section 5.2 introduces VecCore, the vectorization library we introduce in ROOT to implement SIMD operations, and benchmarks its performance by exploiting SIMD operations to generate Julia sets. We use this same benchmark in Section 5.3, to analyze the performance of the support for VecCore types evaluation in TF1, the bounded function class in ROOT.

5.1 Vectorization

Data-level parallelism is achieved in ROOT by exploiting SIMD operations on arrays of data.

Vectorization allows the loading of several values of a specific type in an SIMD array and the activation of the same operation on them simultaneously. The number of elements that we can operate with at the same time will depend on the set of instructions supported by the architecture and the data type of the elements. Table 5.1 provides a relation of sizes for the most common data types and sets of instructions on Intel x86 architectures.

For all these benefits, vectorization does not come without pitfalls. Among the problems that may arise when vectorizing, such as wrong type conversions or non-aligned memory, one of the most serious is branching. In the traditional scalar

Instructions	Array size (bit)	`int` (16 bit)	`float` (32 bit)	`double` (64 bit)
SSE	128	8	4	2
AVX2	256	16	8	4
AVX512	512	32	16	8

Table 5.1: Relationship between the set of SIMD instructions, the size of the SIMD array supported and the number of elements of each type that fit in it.

case, a control flow instruction chooses which branch of the code to select based on a condition, and then executes only the set of instructions corresponding to that branch. In SIMD programming, these control flow operations require performing the computations on all branches and, afterwards, assigning their results per element position, depending on a mask filter created when evaluating the condition on an SIMD array. This is a common operation that can have a strong negative impact on performance. Figure 5.1 illustrates these flows for conditional branching.

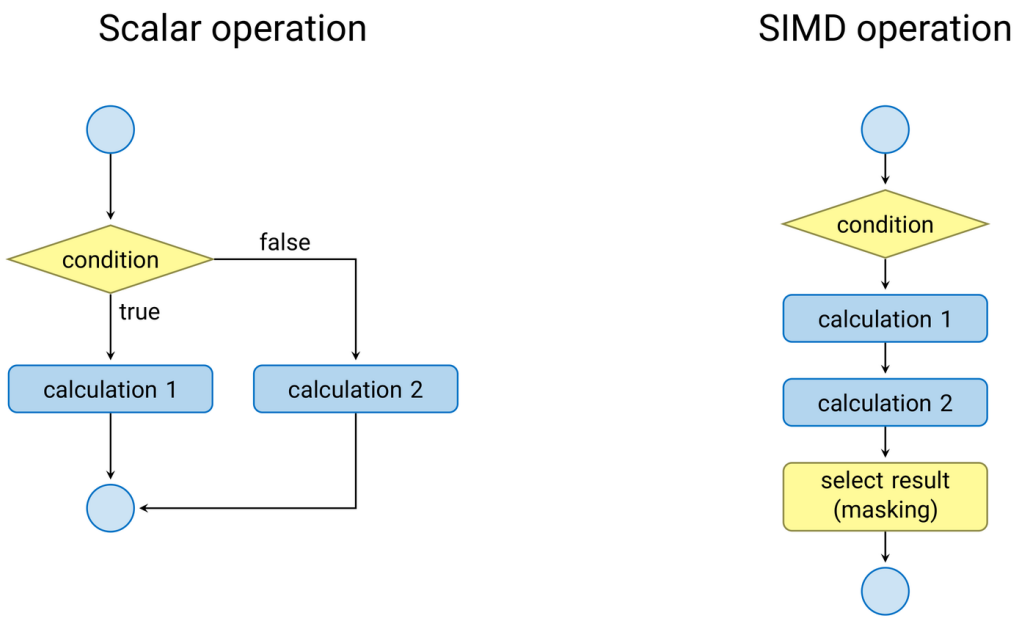


Figure 5.1: Comparison of the conditional branching flow between traditional scalar instructions and SIMD instructions.

In addition to these problems inherent to vectorization, there is the question of usability. Vectorization is usually expressed with compiler intrinsics, which do not expose the most user-friendly interfaces. Libraries such as Vc [35] or UME::SIMD [34], while different in concept, improve the readability and usability of the code

by abstracting these instructions and providing a more familiar and user-friendly interface for the developer. The vectorization library used in ROOT, VecCore, goes one step further and provides an abstraction over these two libraries.

Despite the disadvantages, vectorization offers enough performance benefits to prefer it over scalar operations.

5.2 VecCore

VecCore [4] provides efficient vectorization on a variety of platforms by offering an abstraction layer on top of the libraries Vc, UME::SIMD and the language extension CUDA [42].

These three libraries are supported as optional backends as well as a fallback scalar one in case SIMD operations are not available. The users will make their choices depending, for example, on system availability, performance or simple preference.

VecCore allows the development of architecture-oblivious code that maps to the appropriate backend specific types, methods and instructions. Figure 5.2 displays graphically the most important and recurrent of these instructions, while Listing 10 presents its code API.

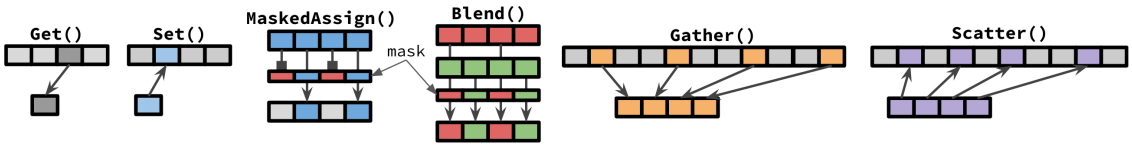


Figure 5.2: Graphical depiction of the VecCore API.

`Get` and `Set` allow safe read/write access to the individual elements of the SIMD array. Given an iterable container of indices, `Scatter` will distribute the elements of the SIMD arrays pointed by those indices, while `Gather` will access the elements referenced by the given indices of the data array and load them into the SIMD array. `MaskedAssign` and `Blend` depend on the concept of *masking*.

A boolean mask is generated when a condition is applied to an SIMD array. This mask signals which elements of the array fulfill the condition and which do not. `MaskedAssign` takes the generated mask, a value, and the destination SIMD array as input, and stores the value in the positions given by the `true` values in the mask. If the value is an SIMD array, the assignment is done element-wise. `Blend` performs the same operation but takes as an input another value that will be assigned to the elements that do not meet the condition.

Listing 11 shows how the branching operations described in Figure 5.1 are applied using VecCore.

```
namespace vecCore {

template <typename T> struct TypeTraits;
template <typename T> using Mask   = typename TypeTraits<T>::MaskType;
template <typename T> using Index  = typename TypeTraits<T>::IndexType;
template <typename T> using Scalar = typename TypeTraits<T>::ScalarType;

// Vector Size
template <typename T> constexpr size_t VectorSize();

// Get/Set
template <typename T> Scalar<T> Get(const T &v, size_t i);
template <typename T> void Set(T &v, size_t i, Scalar<T> const val);

// Load/Store
template <typename T> void Load(T &v, Scalar<T> const *ptr);
template <typename T> void Store(T const &v, Scalar<T> *ptr);

// Gather/Scatter
template <typename T, typename S = Scalar<T>>
T Gather(S const *ptr, Index<T> const &idx);

template <typename T, typename S = Scalar<T>>
void Scatter(T const &v, S *ptr, Index<T> const &idx);

// Masking/Blending
template <typename M> bool MaskFull(M const &mask);
template <typename M> bool MaskEmpty(M const &mask);

template <typename T> void MaskedAssign(T &dst, const Mask<T> &mask, const T &src);
template <typename T> T Blend(const Mask<T> &mask, const T &src1, const T &src2);

} // namespace vecCore
```

Listing 10: VecCore API.

In addition to the described API, VecCore offers vectorized mathematical functions and some other interesting utilities such as the `Load` method, which deals with problems caused by reading from unaligned memory, and safe type castings.

ROOT defines new VecCore-based SIMD array data types, e.g. `ROOT::Float_v` or `ROOT::Double_v`, representing double and single precision SIMD vectors respectively.

Performance analysis: generating Julia sets

We next leverage the generation of Julia sets [60] (Figure 5.3) as a benchmark to evaluate the performance of vectorization with VecCore.

Julia sets are most popularly represented as complex quadratic polynomials. The Julia set we benchmark is composed of a fixed subset of image points (x, y) on the

```
// ROOT::Double_v is the new VecCore based
// double SIMD type integrated in ROOT
ROOT::Double_v x_v, res_v, var1_v, var2_v;
double x, res, var1, var2;

// (.....)

// Case 1: Assigning rvalues if condition is fullfilled
double res = x<0.? 1.: 2.;
vecCore::Blend(res_v, x_v<0.0, 1., 2.);

// Case 2: Assigning variables element-wise if condition is met
double res = x<0.? var1: var2;
vecCore::Blend(res_v, x_v<0.0, var1_v, var2_v);

// Case 3: Only updating elements of the SIMD array that pass the condition
if(x<0.){
   res+=var1;
}
vecCore::MaskedAssign(res_v, x_v<0., res_v + var1_v);
```

Listing 11: Comparison of the expression of branching conditions between the scalar case and using VecCore.

complex plane

$$z = x + iy, \quad z \in \mathbb{C}, \tag{5.1}$$

for which we compute the series

$$z_{n+1} = {z_n}^2 + c \quad c \in \mathbb{C}, \quad -2 \leq z_n \leq 2, \tag{5.2}$$

where $z_0 = x_j + iy_j$, and x and y are the coordinates of the point j in the complex plane, and the series is evaluated until divergence (values of z_n out of bounds) or until z_N, with N being the maximum number of evaluations set by the user to approximate the progression in the bounded case. We obtain a different Julia set approximation for each value of the complex constant c.

Listing 12 contains the code for generating a fractal image related to the Julia set with any library other than the standard, while Listing 13 implements a version of the same code adapted for vectorization using VecCore. The value of each pixel in the final image is given by a function of the number of steps taken by the progression on each point until termination, for both the set of points within our approximation of the Julia set and the ones for which the series diverged. We refer to these images as a representation of the Julia set, despite including values outside of the set.

Benchmarking the evolution of Julia sets

From Equation (5.2) we infer that the time needed to generate different Julia sets from the same initial subset of points in the complex plane varies significantly with

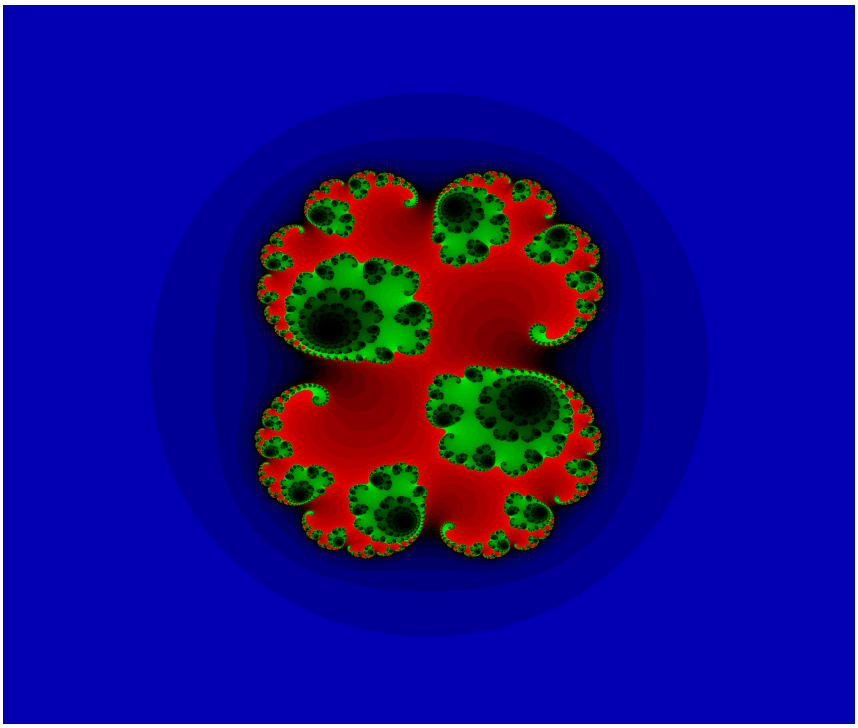


Figure 5.3: Julia set for $c = 0.285 + 0.01i$.

```
template<typename T>
void julia(T xmin, T xmax, size_t nx,
                T ymin, T ymax, size_t ny,
                size_t max_iter, unsigned char *image)
{
    T dx = (xmax - xmin) / T(nx);
    T dy = (ymax - ymin) / T(ny);

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; ++j) {
            size_t k = 0;
            T x = xmin + T(i) * dx, cr = x, zr = x;
            T y = ymin + T(j) * dy, ci = y, zi = y;

            do {
                x  = zr*zr - zi*zi + cr;
                y  = 2.0 * zr*zi + ci;
                zr = x;
                zi = y;
            } while (++k < max_iter && (zr*zr+zi*zi < 4.0));

            image[ny*i+j] = k;
        }
    }
}
```

Listing 12: Traditional implementation of a templated Julia set generation function.

```
template<typename T>
void julia_v(Scalar<T> xmin, Scalar<T> xmax, size_t nx,
                  Scalar<T> ymin, Scalar<T> ymax, size_t ny,
                  Scalar<Index<T>> max_iter,
                  unsigned char *image)
{
    T iota;
    for (size_t i = 0; i < VectorSize<T>(); ++i)
        Set<T>(iota, i, i);

    T dx = T(xmax - xmin) / T(nx);
    T dy = T(ymax - ymin) / T(ny), dyv = iota * dy;

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; j += VectorSize<T>()) {
            Scalar<Index<T>> k{0};
            T x = xmin + T(i) * dx,       cr = x, zr = x;
            T y = ymin + T(j) * dy + dyv, ci = y, zi = y;

            Index<T> kv{0};
            Mask<T> m{true};

            do {
                x = zr*zr - zi*zi + cr;
                y = T(2.0) * zr*zi + ci;
                MaskedAssign<T>(zr, m, x);
                MaskedAssign<T>(zi, m, y);
                MaskedAssign<Index<T>>(kv, m, ++k);
                m = zr*zr + zi*zi < T(4.0);
            } while (k < max_iter && !MaskEmpty(m));

            for (size_t k = 0; k < VectorSize<T>(); ++k)
                image[ny*i+j+k] = (unsigned char) Get(kv, k);
        }
    }
}
```

Listing 13: VecCore based implementation of a Julia set generator. This implementation can be executed on any backend supported by VecCore, from the Scalar backend to Vc or UME::SIMD, and it can target any set of SIMD instructions supported by the architecture it executes on.

the value c. For this reason, we consider interesting to evaluate the performance of vectorization with the evolution of the fractal image, that is, the Julia sets generated with different values for the complex constant c.

To do this, we represent c as a complex number in polar form, i.e. $c = b * e^{i\alpha}$, where $b \in \mathbb{R}$ and α is the angle parameter in the interval $[0, 2\pi)$. If we fix the value of b, e.g. to $b = 0.7885$, and approximate the series by restricting the maximum number of evaluations to $N = 100$, we observe the following evolution of the fractals given by the Julia sets defined by the progression of the series:

$$z_{n+1} = {z_n}^2 + 0.7885e^{i\alpha}, \quad -2 \leq z \leq 2, \, \alpha \in [0, 2\pi), \quad n \leq 100. \tag{5.3}$$

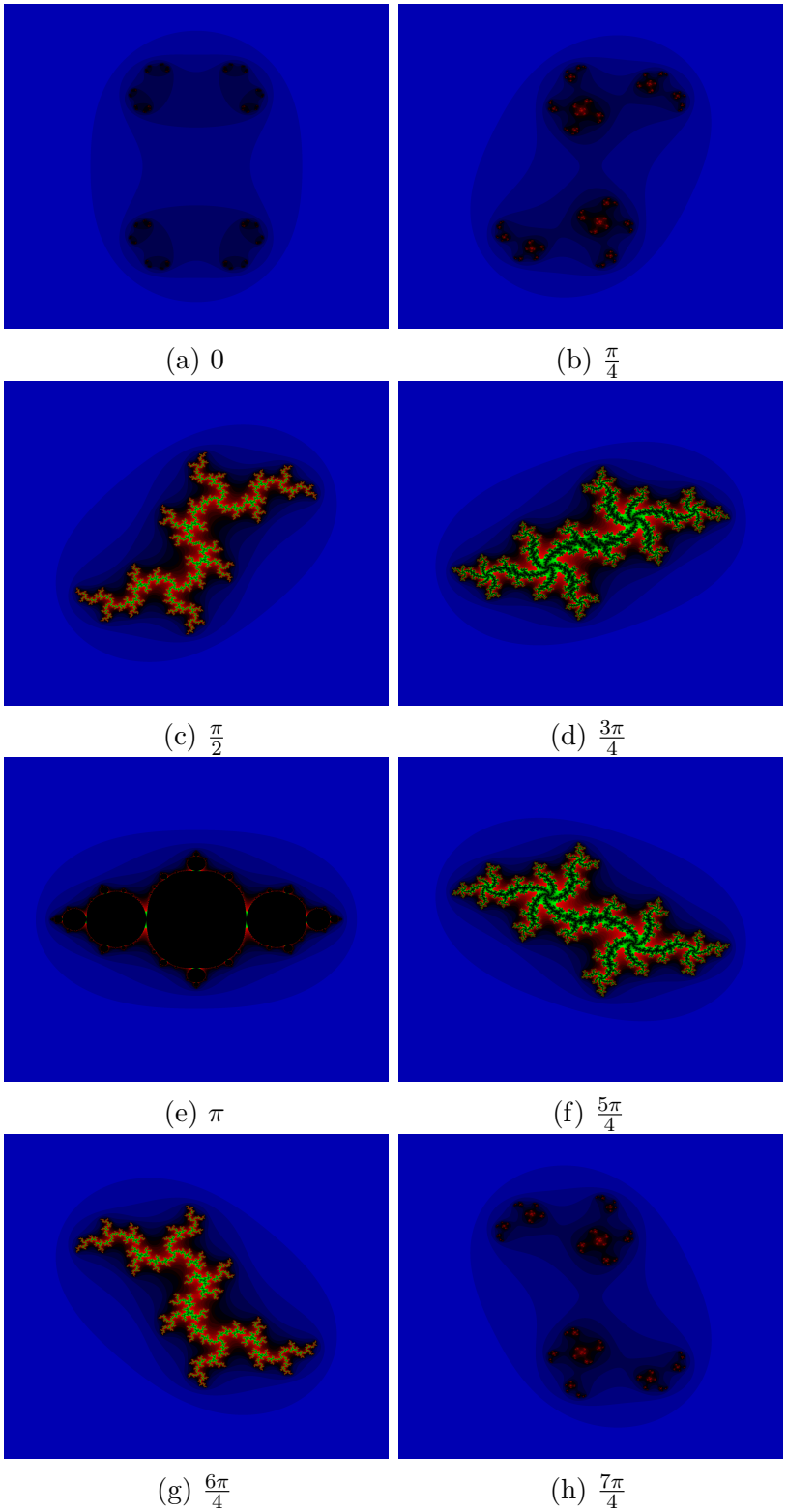
Figure 5.4 depicts the fractals generated by this formulation of the Julia sets in steps of 45 degrees, or $\frac{\pi}{4}$ radians, for α in the interval $[0, 2\pi)$.

The following measurements have been taken in an Intel Core i7-4790 desktop server with 4 physical cores (8 logical threads) at 3.6 GHz and 8 GB of RAM [26], executed with AVX2 as SIMD instruction set and compiled with gcc 7.

In Figure 5.5, we display the execution times measured for a continuously increasing α in the same interval when generating the Julia sets obtained by evaluating the series progression described in Equation (5.3). The time exhibits a high volatility based on the value of α, independently of the backend of choice. These measurements also reveal that Vc outperforms UME::SIMD in any benchmarked type and that, while vectorization always provides better times, the scalar backend displays minimal performance penalty overall and even occasionally improves the time obtained when evaluating the base types. These differences can be observed more clearly in the speed up plot in Figure 5.8, where we compute the speed up with respect to the base types case.

The speed ups stay below the ideal ones and, although they improve occasionally in several spikes that match those of the execution time figure, they are greatly influenced by the backend of choice. In this regard, we notice that the execution on base types (`float`, `double`) outperforms the execution relying on VecCore's scalar backend. We attribute this observation to auto-vectorization.

If we compute the speed up of these results with respect to the time obtained for the base types without allowing auto-vectorization (Figure 5.7), we observe that Vc considerably outperforms UME::SIMD as a backend in both cases (Table 5.2). In addition, VecCore's scalar backend now improves the time obtained with the base types, although it does not match the autovectorization offered by the compiler, except on the spikes. Curiously, it is in the same spikes that we obtain the best vectorization for any of the other backends. This seems to indicate that the vectorization requires a greater amount of work to yield noticeable benefits. However, if we plot the fractal picture, i.e. the Julia set generated, for the value of the angle α corresponding to each of the local maximums observed (Figure 5.9), we observe that



(a) 0 (b) $\frac{\pi}{4}$

(c) $\frac{\pi}{2}$ (d) $\frac{3\pi}{4}$

(e) π (f) $\frac{5\pi}{4}$

(g) $\frac{6\pi}{4}$ (h) $\frac{7\pi}{4}$

Figure 5.4: Evolution of the Julia sets generated for increasing values of the angle α, with a stride of $\frac{\pi}{4}$ radians, in Equation (5.3).

these images have a sizeable black area. We call this the convergence area, that is, an area composed of points that did not diverge before reaching the N_{th} step in the series progression (our approximation to the theoretical notion of a Julia set).

	Base type	`float`		`double`	
	Backend	UME::SIMD	Vc	UME::SIMD	Vc
Speed up	Reference	8	8	4	4
	Max.	6	6.17	1.75	2.84
	Min.	2.75	3.52	1.27	2.02
	Avg.	3.45	4.36	1.51	2.34

Table 5.2: Comparison of the maximum (*Max.*), minimum (*Min.*) and average (*Avg.*) speed ups obtained respect to the ideal ones (*Reference*) for the Vc and UME::SIMD backends of VecCore evaluating in the AVX2 instruction set.

When vectorizing the generation of Julia sets, we attribute the improved speed up to the existence of large uniform areas. In contrast with the N evaluations reached by the points within the convergence area, the points that diverge (i.e. reach out of bounds values, in this case outside the interval $[-2, 2]$) require a heterogeneous number of evaluations of the series progression. In terms of computation, this is related to the conditional branching problem introduced in Section 5.1: let S be an SIMD array fitting 4 integers, and let $\vec{x}$ be an array of integer data of the same size as the SIMD array, in which processing each component takes the following amount of evaluations:

$$\vec{x} = (\underbrace{x_0}_{10\ ev.}, \underbrace{x_1}_{20\ ev.}, \underbrace{x_2}_{80\ ev.}, \underbrace{x_3}_{35\ ev.})$$

Tables 5.3 and 5.4 explain the number of evaluations required in total for the evaluation of the data in scalar and vectorized mode, respectively. In this case, we obtain a speed up of $137/80 = 1.71$, when the ideal one (i.e. each element requires the same number of evaluations) is 4. A homogeneous area in the image indicates that the pixels within its borders take a similar number of evaluations and, as a consequence, avoid the problem of conditional branching. Heterogeneous areas may cause the fluctuations in execution time and speed up displayed in Figure 5.5 and Figure 5.7. However, after analyzing the homogeneity of the images and computing the potential speed up loss caused by branching for several values of α, it becomes clear that branching only affects the speed up in a minimal way in this case, almost negligible with respect to the results obtained.

If we benchmark the code line by line, and analyze the assembly instructions, we can deduce that the vectorization libraries are responsible for the suboptimal speed

$$\begin{array}{r} 10 \\ 12 \\ 80 \\ +35 \\ \hline \mathbf{137} \end{array}$$

Table 5.3: Number of evaluations of the scalar case: 370. We define the total number of evaluations in the scalar case as the sum of the evaluations required for each element.

$$S \quad = \quad \begin{array}{|c|c|c|c|} \hline 10 & 20 & \mathbf{80} & 35 \\ \hline \end{array}$$

Table 5.4: Number of evaluations on the vectorized case: 80. The SIMD array is evaluated as a whole as many times as the element that requires the most evaluations.

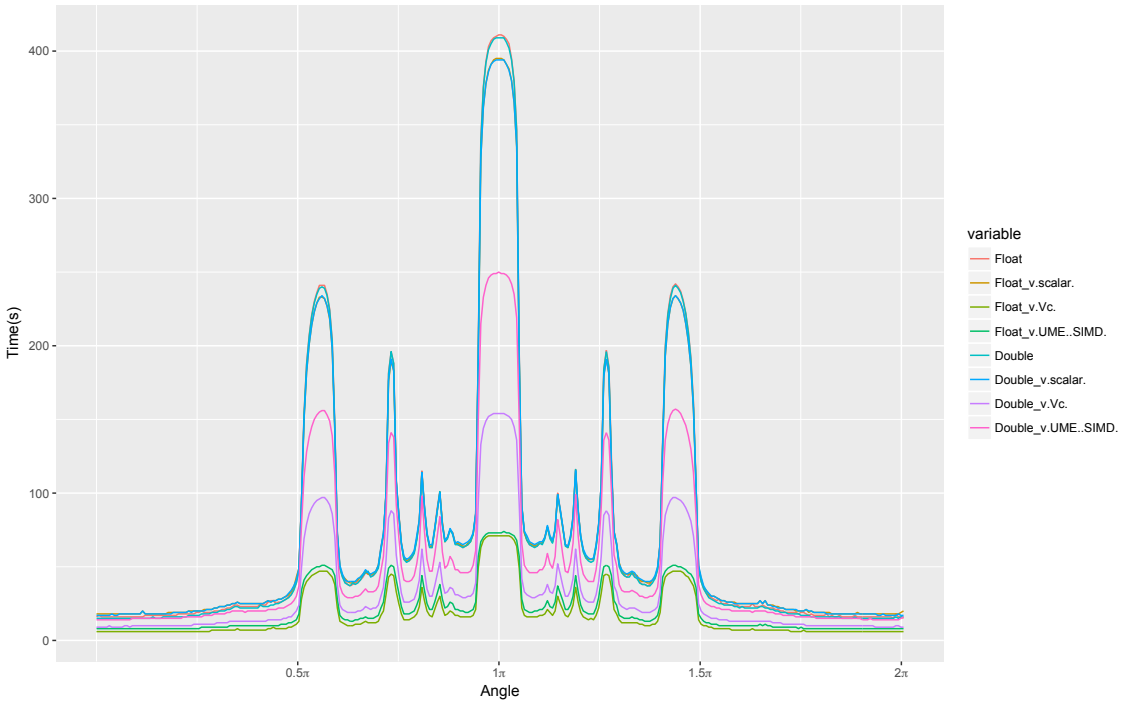


Figure 5.5: Execution times for pairs (data type, backend) of the Julia fractal formulation in Equation (5.3) for different values of the angle α.

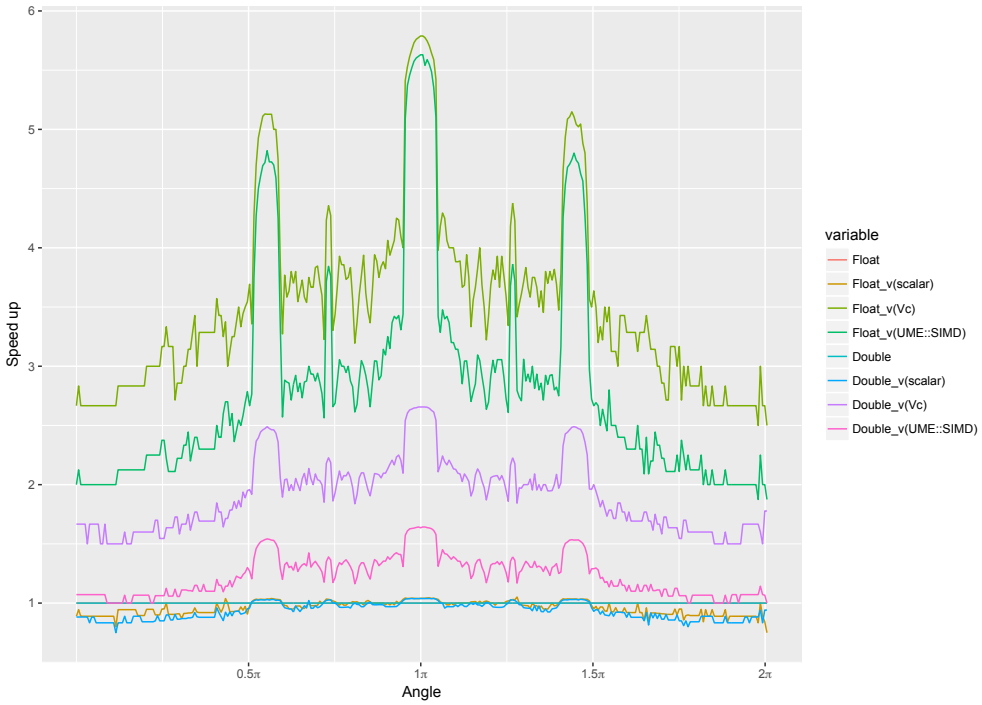


Figure 5.6: Speed up of the Julia sets generated for $z^2 + 0.7885e^{i\alpha}$ with $\alpha \in (0, 2\pi]$.

ups. The implementation of these libraries, and especially the SIMD intrinsics chosen to implement the abstracted operations, might impact performance considerably. For example, in the implementation of the initialization of the mask in Vc, and probably as a compromise to offer a user-friendly interface, the code loops through the SIMD array, assigning the initialization value element by element. These instructions are scalar instead of packed. This occurs in UME::SIMD as well. Due to the particularities of its design, UME::SIMD emulates scalar instructions instead of using vector operations more frequently than Vc. Furthermore, the out-of-order execution of the processor might be able to process and parallelize scalar operations more efficiently at instruction level.

In addition, the architecture we execute on may attenuate the differences in speed up even more, especially with respect to instruction-level parallelism. As an example, Table 5.5 displays the performance of the instruction adding two unsigned integers. As we can observe, Intel's Skylake architecture offers a throughput of 3 instructions per CPU cycle (ICP), improving the ICP rate we obtain on the Broadwell and Haswell architectures.

Figure 5.7: Speed up obtained, without allowing autovectorization, of the Julia sets generated for $z^2 + 0.7885e^{i\alpha}$ with $\alpha \in (0, 2\pi]$.

Architecture	**Latency**	**Throughput**
Skylake	1	0.33
Broadwell	1	0.5
Haswell	1	0.5

Table 5.5: Performance of the AVX2 instruction for the 8-bit integer addition operation. Extracted from the Intel intrinsics guide [27].

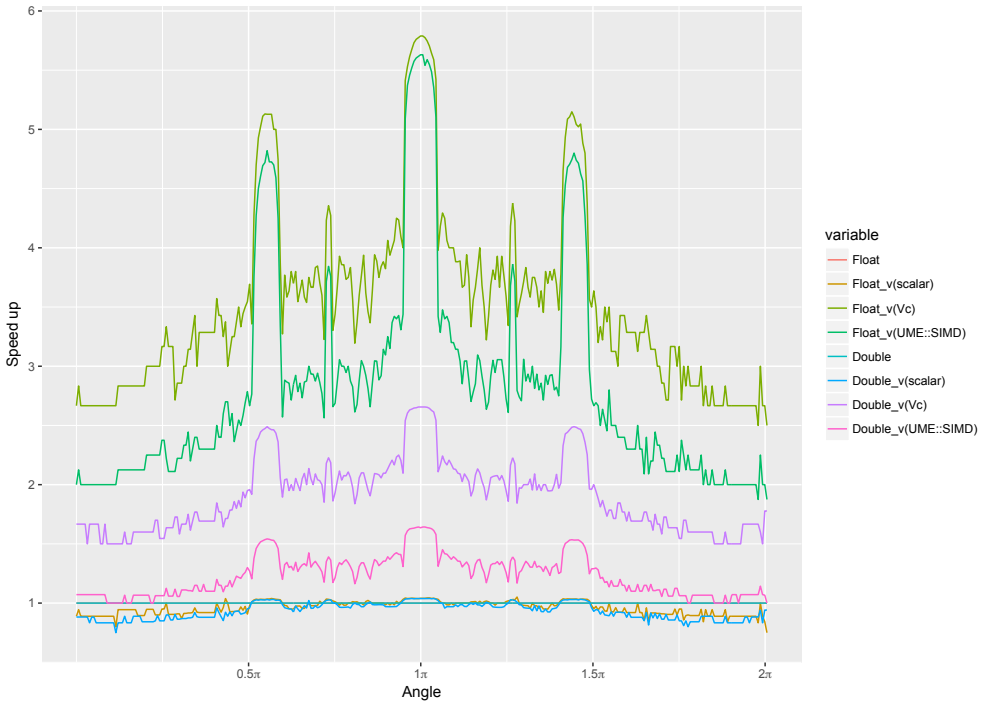


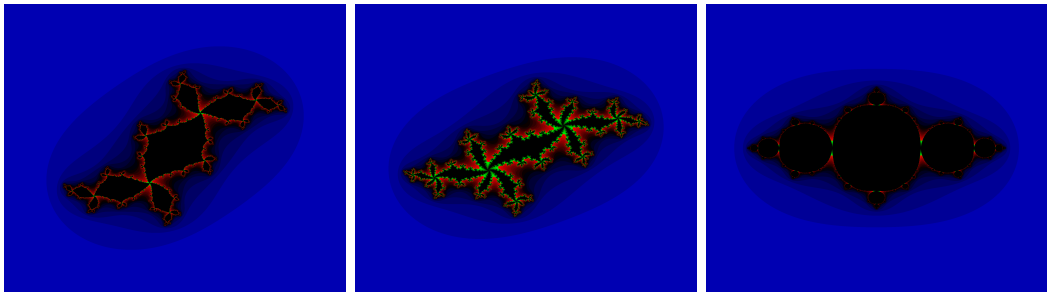
Figure 5.8: Speed up of the Julia sets generated for $z^2 + 0.7885e^{i\alpha}$ with $\alpha \in (0, 2\pi]$.

Compiler comparison

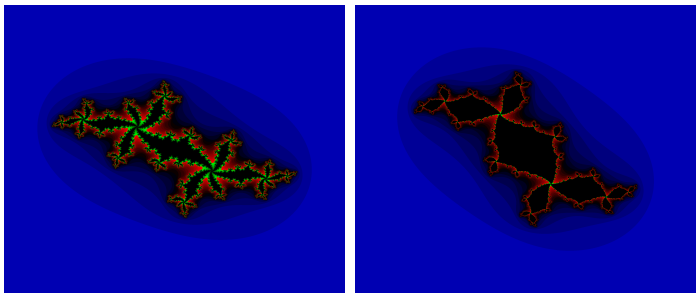
A conspicuous result from the previous benchmarks is the difference in speed up between Vc and UME::SIMD. One of the most plausible reasons for this considerable gap in performance is that UME::SIMD is a library that has been optimized for its use in combination with icc instead of gcc. That is, the implementation of the vectorization directives in each library used as a backend for VecCore may be optimized for a concrete compiler.

Therefore, it is interesting to measure the difference in performance of these libraries when relying on different compilers, in order to make an informed decision about the combination of VecCore backend and compiler that provides the best performance. Figure 5.10 offers the speed up comparison for both `float` and `double` types in AVX2 SIMD arrays for three main compilers (gcc, icc and clang). All of the following measurements have been taken in a Intel Core i7-4790 desktop server with 4 physical cores (8 logical threads) at 3.6 GHz and 8 GB of RAM [26] without task-level parallelism being exploited.

We observe that Vc generates more efficient SIMD instructions when compiling with gcc and clang, while UME::SIMD displays better results with icc than gcc. In particular, Vc shows a considerably higher speed up in the double precision case for



(a) $\sim 0.56\pi$ (b) $\sim 0.74\pi$ (c) π



(d) $\sim 1.26\pi$ (e) $\sim 1.44\pi$

Figure 5.9: Angles (in radians) where the time exhibits a spike. Notice how all maximize the black central area (area of convergence).

both gcc and clang in comparison with any other combination.

If the best option were to be selected in terms of speed up, Vc and gcc would be the combination of choice, closely followed by Vc plus clang. However, this deduction is based on a wrong premise, as the speed ups are computed as a factor of the sequential execution compiled by the same compiler. Therefore, we cannot base our decision on the speed up metric.

Instead, we must consider the absolute execution time. In Figure 5.11 we display the different execution times observed with our benchmark code for an increasing α in the interval $[0, 2\pi)$. From this figure we conclude that the best absolute choice is to compile Vc with gcc, while the worst one is to compile UME::SIMD code with clang. In general, we note that Vc reduces the time obtained by UME::SIMD independently of the compiler.

Efficiency of Julia set generation for different sets of instructions

Finally, we measure the efficiency of our implementation when executing VecCore's vectorized backends on different sets of SIMD instructions. For a more complete picture, we execute our benchmark on a machine that supports SSE2, AVX2 and AVX 512 as sets of instructions, an Intel Xeon Phi 7210 [31] server composed of two

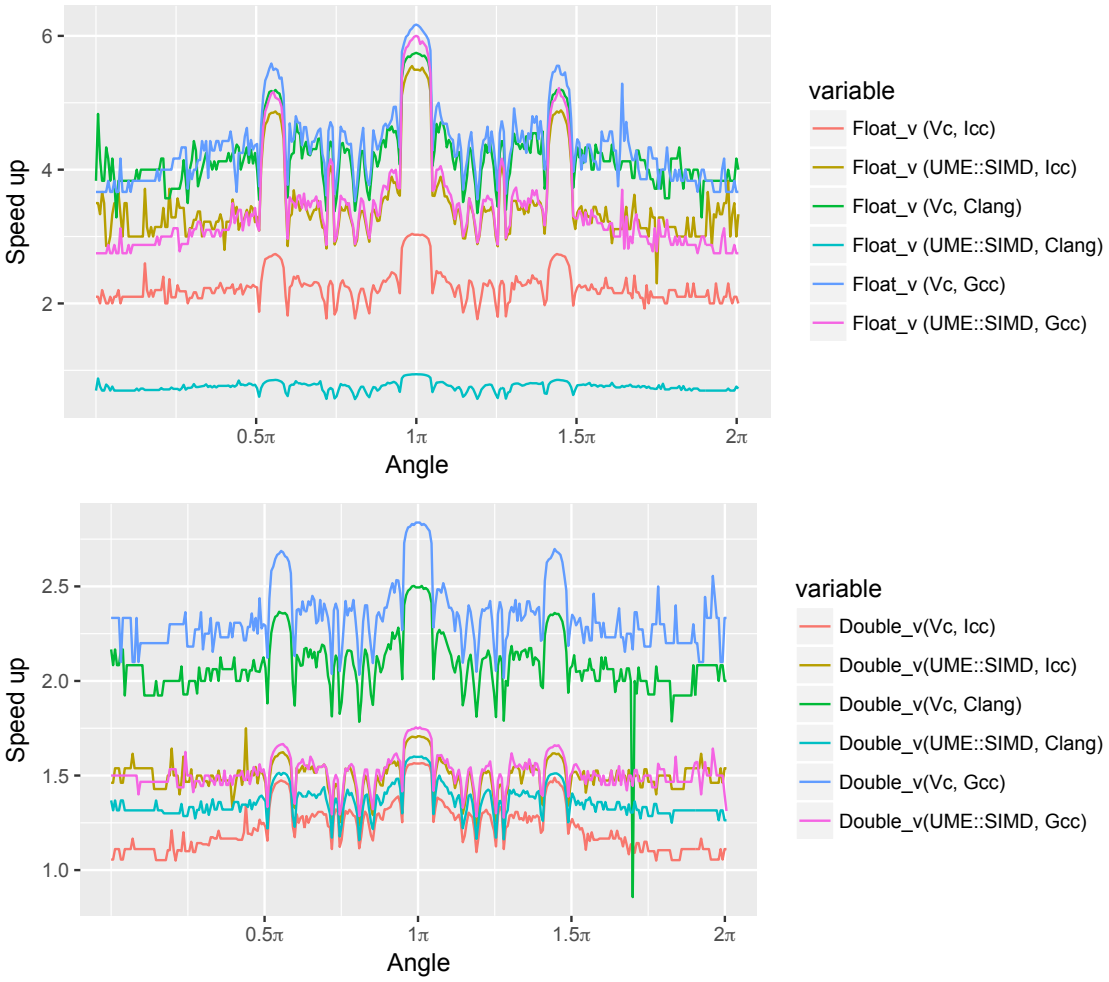


Figure 5.10: Speed up obtained for each pair of compiler (gcc, icc, clang) and vectorization library supported as a backend by VecCore (UME::SIMD, Vc).

sockets with 64 cores each at 1.3 GHz (supporting hyperthreading) and 110 GB of RAM in total. The benchmark is executed in the quadrant execution mode of the Xeon Phi, and it has been compiled with icc 17 (the only compiler yet supporting the AVX 512 set of instructions).

Figure 5.13 displays the execution times recorded for each combination of VecCore's vectorization library backend and SIMD instruction set; and Figure 5.10 reports the speed up. In both figures the value of α varies in the interval $[0, 2\pi)$. In general, the results exhibit less fluctuation and volatility in this platform (Xeon Phi) than in the previous servers.

One of the first things we notice is that Vc does not have a proper AVX512 implementation and falls back to AVX2. Therefore, the times and speed up obtained

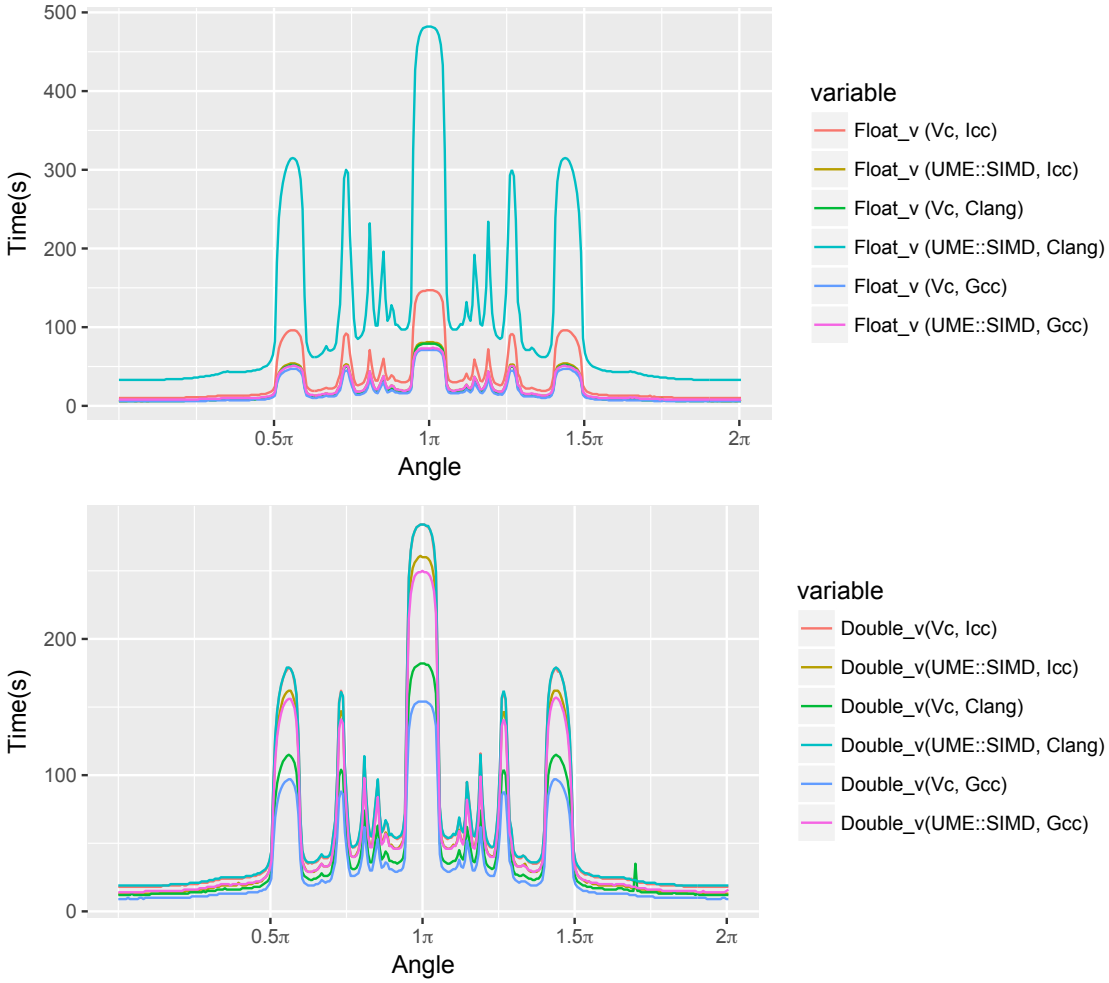


Figure 5.11: Times obtained for the computation of the Julia set for each pair of compiler (gcc, icc, clang) and vectorization library supported as a backend by VecCore (UME::SIMD, Vc).

by Vc for both the AVX2 and the AVX512 sets of instructions are similar. While executing with AVX2, Vc improves the results of UME::SIMD for the `double` case, but underperforms with `float`. The inverse occurs when executing with SSE2, with Vc significantly improving the results attained by UME::SIMD, which shows a slight slowdown with respect to the base type execution in this case. Finally, UME::SIMD displays high scalability between the three instruction sets, and the support of the AVX512 SIMD instruction set renders it the most convenient backend when executing on platforms that support it.

As a final remark, the speed up obtained with the combination of UME::SIMD and AVX512 is the best we have recorded for any benchmark, but looking at the execution time, we can assert that this platform's processor frequency is too slow to make it the preferred option.

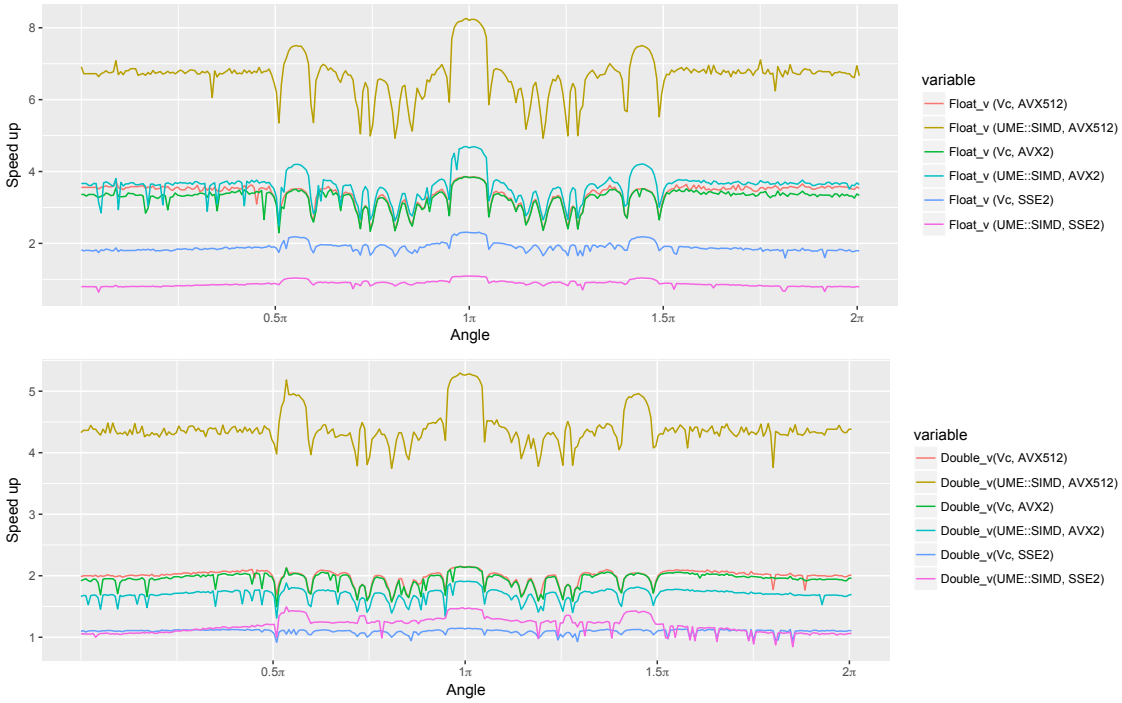


Figure 5.12: Speed up of the computation for each tuple of compiler (gcc, icc), instruction set (SSE2, AVX2, AVX512) and vectorization library supported as a backend by VecCore (UME::SIMD, Vc).

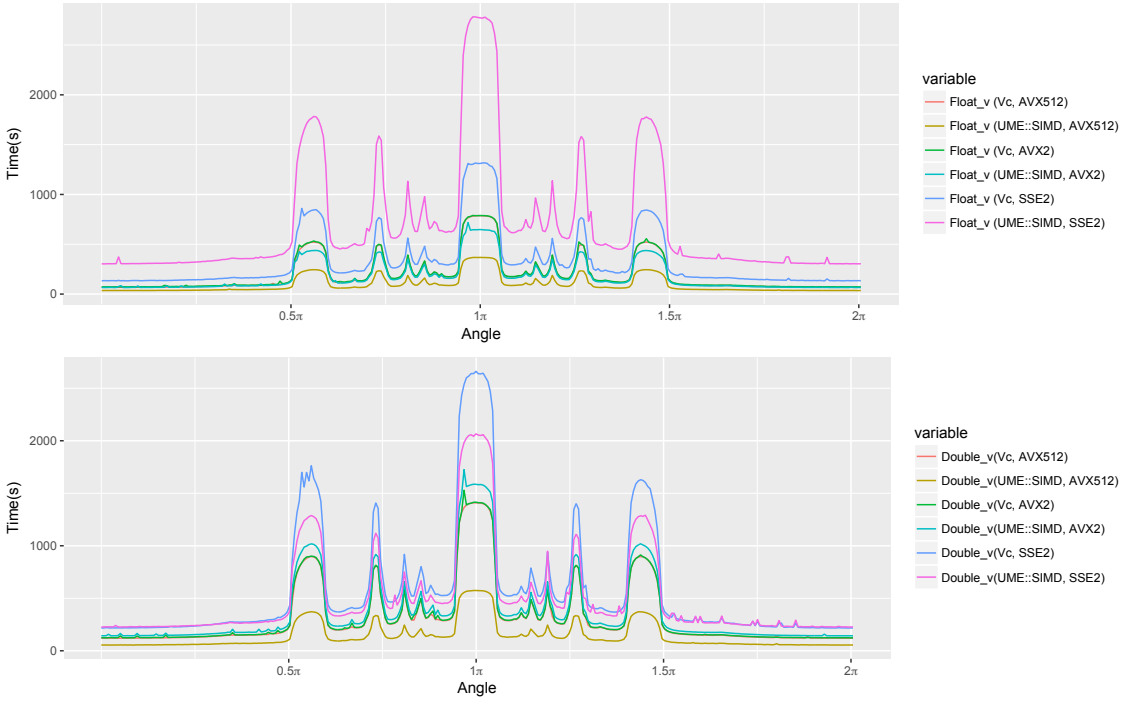


Figure 5.13: Absolute times of the computation of the Julia set for each tuple of compiler (gcc, icc), instruction set (SSE2, AVX2, AVX512) and vectorization library supported as a backend by VecCore (UME::SIMD, Vc).

5.3 Generating Julia fractals with TF1

We benchmark again the generation of Julia sets, this time with a constant $c = 0.285+0.01i$ (Figure 5.3), using the new ROOT SIMD types, concretely `ROOT::Double_v`, and the interfaces we adapted for vectorization. For this, we use TF1, the ROOT class representing a one dimensional function defined between lower and upper limits. TF1 is a wrapper over a user-specified function that extends its functionality to adapt it to ROOT's interfaces and conforms a fundamental class in ROOT. It is also the base class for the multidimensional function classes (TF2, TF3), and it is leveraged with frequency, from fitting (as it can be parameterized) to graphics display.

Unlike other classes, such as those that define the fitting interfaces, TF1 is exposed to the user for direct manipulation. For this reason, and to maintain backwards compatibility, instead of templating the class, we template the necessary class members to process the new vectorized types. This makes existing code available for vectorization by adapting the user function to operate on `ROOT::Double_v`.

In addition, we developed TF1 further with new features such as new construc-

tors for `std::function` types and the possibility to evaluate over scalar data types (`double`) when the function that TF1 has been built on is vectorizable. In this case, we fill the entire SIMD array with the scalar value, operate with it, and return only one of the (identical) values of the resulting SIMD array.

Our benchmark also serves to measure the adaption of a user function with more than two parameters to the TF1 interfaces. TF1 is a parametric function with strict interfaces, allowing as its only parameters a pointer to the data and a pointer to the parameters needed for the evaluation of the user function. This limitation is frequently overcome by wrapping the user function in a C++ functor, packing the parameters in an array of `double` when calling the TF1, and unpacking them after forwarding the call to the user provided functor. This operation, shown in Listing 14, adds a slight overhead in the packing/unpacking process and to the call forwarding.

Figure 5.14 presents the results obtained on an Intel Core i7-4790 desktop server with 4 physical cores (8 logical threads) at 3.6 GHz and 8 GB of RAM [26], executed with AVX2 as set of SIMD instruction and compiled with gcc 6.2. Notice how the functor adaption and TF1 wrapping adds a small overhead in all cases, and VecCore's scalar backend essentially matches the performance attained by the original code including performing several masking operations as a consequence of conditional branching (Section 5.1). Finally, we observe a fair speed up when vectorizing, considering the complexity of the problem at hand.

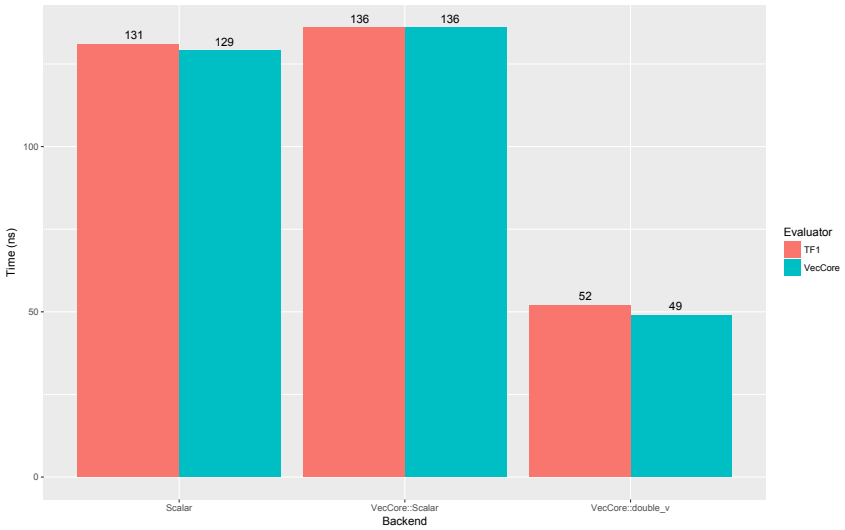


Figure 5.14: Performance comparison of Julia fractal generation with three different backends. VecCore::Scalar offers TF1 wrapping adds negligible overhead.

We further benchmark the performance of TF1 in Section 7.3, where we compare the single call performance of TF1 built with a free function and TFormula, the formula class in ROOT, with their vectorized counterparts.

```
class Julia {
public:
    Julia(size_t xpx, size_t ypx, size_t iterations):
       nx(xpx), ny(ypx), max_iter(iterations), image(new unsigned char[nx*ny]) {}

    double operator()(const double *data, const double *c) const
    {
        double xmax = data[0], xmin = data[1], ymax = data[2], ymin = data[3];
        double dx = (xmax - xmin) / nx;
        double dy = (ymax - ymin) / ny;

        for (size_t i = 0; i < nx; ++i) {
            for (size_t j = 0; j < ny; ++j) {
                size_t k = 0;
                double x = xmin + i * dx, cr = c[0], zr = x;
                double y = ymin + j * dy, ci = c[1], zi = y;

                do {
                    x  = zr*zr - zi*zi + cr;
                    y  = 2.0 * zr*zi + ci;
                    zr = x;
                    zi = y;
                } while (++k < max_iter && (zr*zr + zi*zi < 4.0));

                image[ny*i + j] = k;
            }
        }

        return 0.0;
    }

    unsigned char* getImage(){ return image;}

private:
    size_t nx;
    size_t ny;
    size_t max_iter;
    unsigned char *image;
};
```

Listing 14: C++ functor that computes the Julia set operating on SIMD types. It adapts the function in Listing 13 and wraps it in a class.

5.4 Conclusions

We introduced data-level parallelism in ROOT by capitalizing on the advantages offered by the VecCore library and its backends in terms of programming model and performance. By offering an extra abstraction layer on top of its exchangeable backends, VecCore provides efficient vectorization with an improved yet simpler programming model that, among other benefits, allows for better portability with the development of architecture-oblivious code.

We benchmarked VecCore's performance extensively with a complex example,

the computation of different Julia sets. We evaluated the generation of Julia sets on two different floating point types, `float` and `double`; two different computers, a desktop server and a Xeon Phi; two different VecCore backends, Vc and UME::SIMD; three different compilers, icc 17, gcc 7.2 and clang 5; and three different instruction sets, SSE2 (desktop server and Xeon Phi), AVX2 (desktop server and Xeon Phi) and AVX512 (Xeon Phi).

In all cases, VecCore exhibits suboptimal speed ups due to the nature of the benchmark, with gcc performing better than icc or clang. We demonstrate that Vc outperforms UME::SIMD in almost all scenarios, but the latter supports AVX512, which is not the case for the former. Finally, while the highest speed ups are obtained with AVX512 in the Xeon Phi server, the processor frequency in this machine is not fast enough to consider this combination as the best option.

In addition, we introduced two new SIMD types based on VecCore types in ROOT, `ROOT::Float_v` and `ROOT::Double_v`, for the implementation of SIMD operations in ROOT's classes. Moreover, we provide an example of the integration of VecCore within ROOT by introducing VecCore-based SIMD evaluation in TF1, the bounded function class, which is widely used in ROOT for the performance of mathematical operations. We compare the execution of a vectorized function as a free function with the same function adapted for its execution as a TF1. We demonstrate an efficient implementation of SIMD vectorization in TF1, which even with the required scaffolding, produces almost negligible overhead over the free function evaluation.

Chapter 6

Parallelization of the fit

Fitting of data distributions is one of the most demanding and computing-intensive activities when performing data analysis on the results of the LHC.

In order to take full advantage of new computer architectures, and to maintain or even improve performance with increasing amounts of data to analyze, parallelization and vectorization have been introduced in the ROOT mathematical and statistical libraries.

The tools for parallelism described in Chapters 4 and 5 can be applied together when parallelizing the minimization process for solving fitting problems. As a result of the substantial work done on developing and integrating these parallelization tools in ROOT, vectorization and parallelization have been introduced in ROOT's fitting interfaces, requiring minimal changes in user code. To improve the speed up, ROOT's main fitting methods, introduced in Section 6.1, have been parallelized both at data-level (Section 6.3) and task-level (Section 6.4).

We report on the improvements obtained by adding support for SIMD vectorization and multithreaded parallelization in the function evaluation with a typical example from HEP data analysis. In Section 6.5, we show how the different evaluations of the likelihood and the least square functions used for fitting ROOT histograms and datasets represented by ROOT trees perform for this case.

6.1 Fitting fundamentals

Fitting is one of the most recurrent and fundamental operations in HEP, as it provides valuable information about the distribution of the observed data. In this section we introduce the foundations of likelihood fitting, introducing the concepts of likelihood and likelihood function, and focusing on two methods to estimate the parameters that maximize the likelihood function: maximum likelihood and least squares (Section 6.1).

The likelihood function

Consider the *hypothesis* H as a conjecture over the probability $P(x|H)$ of the observed data x. If $P(x|H)$ were to define the probability density function (p.d.f.) for the data as a function of H, we would then call $L(H) = P(x|H)$ the *likelihood of* H.

If the hypothesis is characterized by the typically unknown parameters θ, then $L(\theta) = P(x|\theta)$ is called *the likelihood function.*

The likelihood function plays a central role in the fitting process, which ultimately consists of inferring θ from x so that we find the probability mass or density function (in this case $L(\theta)$) that better adjusts to the observed data. For this purpose, we need good estimators of the parameters.

For instance, we are frequently in need of good estimators for the median, mean and variance parameters of a distribution. In this case, a good choice of estimators is to set the median $\hat{x}_{med}$ to the sample median and to estimate the mean and the variance with the unbiased estimators given by the equations

$$\begin{aligned} \hat{\mu} &= \frac{1}{n}\sum_{i=1}^{n} x_i \qquad\qquad \text{and} \\ \hat{\sigma}^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2, \end{aligned} \tag{6.1}$$

respectively.

Maximum Likelihood

Given a set of parameters θ, we want to obtain the set of estimators $\hat{\theta}$ that maximize the likelihood function. In other words, the *maximum likelihood estimate* follows

$$\hat{\theta} = \max L(\theta). \tag{6.2}$$

However, it is frequently more convenient to work with the natural logarithm of the likelihood function, $\ln L$, finding the estimators by solving the equation

$$\frac{\delta \ln L}{\delta \theta_i} = 0, \quad i = 1, ..., N. \tag{6.3}$$

The likelihood function formulation depends on the nature of the data. For, example, if the data consists of a fixed number n of independent and identically distributed values following the same p.d.f., the likelihood becomes

$$L(\theta) = \prod_{i=1}^{n} f(x_i; \theta). \tag{6.4}$$

where $f(x, \theta)$ is the p.d.f. of observing the data x given the set of parameters θ. If the data does not consist of a fixed number of events, but the probability to observe n events follows a Poisson distribution with mean μ depending on the parameter θ, then we obtain the *extended likelihood* function formula:

$$L(\theta) = \frac{\mu^n}{n!} e^{-\mu} \prod_{i=1}^{n} f(x_i; \theta). \tag{6.5}$$

Binned maximum likelihood

If the data sample contains a large number of values, it is usually convenient to represent it as a histogram, classifying the values in N bins.

If, again, the n bins created out of n_{tot} events are independent and follow a Poisson distribution, the likelihood function takes the form:

$$f_P(n; \theta) = \frac{\mu_i^n}{n_i!} e^{-\mu_i} \prod_{i=1}^{n} f(x_i; \theta), \tag{6.6}$$

where μ_i are the mean values of the bins described as functions of θ, and $\mu_{tot} = \Sigma_i \mu_i$ is the mean of the n_{tot} original events of the Poisson distribution.

By using the *likelihood ratio* $\lambda(\theta)$, we can obtain, in addition, an indicator of the "goodness of fit" [7]:

$$-2ln\lambda(\theta) = 2 \sum_{i=1}^{N} \left[\mu_i(\theta) - n_i + n_i \ln \frac{n_i}{\mu_i^n} \right], \tag{6.7}$$

where $\lambda(\theta) = f_P(n; \theta) / f(n; \hat{\mu})$.

Least squares

Besides the maximum likelihood, there is an alternative way of inferring the parameters by using the method of the *least squares*.

Given a set of N independent measurements at point y_i, with mean $\mu(x_i; \theta)$ and variance σ_i^2, the least squares estimators $\hat{\Theta}$ for the unknown parameters Θ can be defined by the equation:

$$\chi^2(\theta) = \sum_{i=1}^{N} \frac{(y_i - \mu(x_i; \theta))^2}{\sigma_i^2}, \quad \forall \theta \in \Theta, \tag{6.8}$$

where the parameters minimizing $\chi^2(\theta)$ maximize the likelihood L. If y_i were to be Gaussian distributed, the least squares method would be equivalent to that of the maximum likelihood described in Equation (6.4).

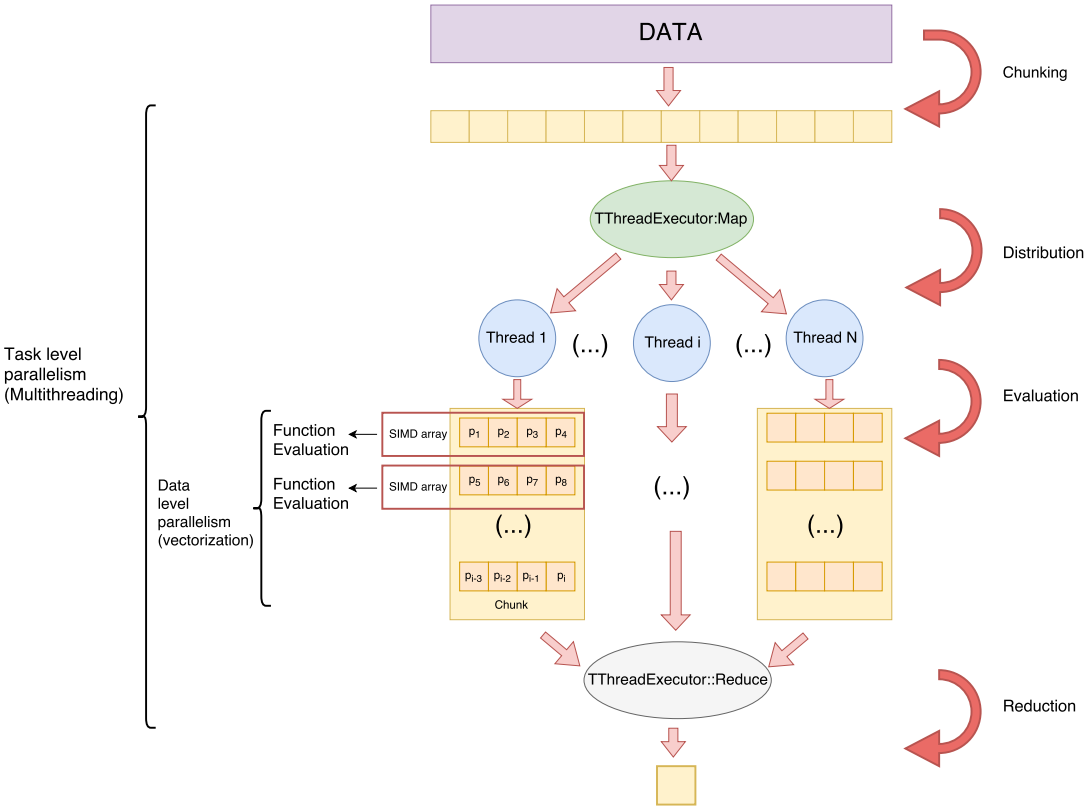


Figure 6.1: Depiction of the parallelization of the fitting. The parallelization at task-level involves chunking, multithreaded mapping (distribution and evaluation) and reduction to a single result. At data-level, we vectorize the model function evaluation.

6.2 Parallelization of the fitting in ROOT

In this Section, we study the parallelization of the maximum likelihood (Equations (6.4) and (6.5)), the binned likelihood (Equation (6.7)) and the least squares (Equation (6.8)) methods.

Figure 6.1 illustrates the parallelization process followed in each one of the algorithms and the way each layer of parallelism is exploited in a multicore computer.

We start by dividing the data into N chunks and distributing them to N tasks that will run in M threads (*Chunking* stage). The Map stage of the MapReduce implementation of choice —TProcessExecutor (Section 4.3) for multiprocessing, TThreadExecutor (Section 4.4) for multithreading—is responsible for the distribution of the tasks to the threads (*Distribution* stage).

Although the clustering of the data can be delegated to the user, we can take advantage of the chunking capabilities of the MapReduce executors as well. However,

since in this case the nature of the data and the evaluation functions are opaque to the fitting interfaces, it is difficult to infer the most efficient number of chunks without any input from the user.

Notwithstanding, we provide a heuristic function to infer an appropriate number of chunks, depending on the size of the datum and within a range given by empirical experimentation (see Section 6.4). This number of chunks should be large enough to improve balancing, but within a certain limit to avoid that the overhead of task creation undermines the balancing benefits.

Every task of the Map stage evaluates the function on C events, where $C = \frac{nEvents}{N}$ is the chunk size (*Evaluation* stage). If the user provides a vectorized function that takes ROOT's SIMD types as parameters, while still evaluating C events, the number of function evaluations per task is reduced from C to $\sim \frac{C}{S}$. Here, S is the number of `double` types fitting into the SIMD array size of the instruction set, as the ROOT fitting interfaces are designed to only evaluate on `double`. That is to say that Equation (6.8) becomes

$$\chi^2(\theta) = \sum_{i=1}^{N} \frac{(y_i - \mu(x_i;\theta))^2}{{\sigma_i}^2} = \sum_{j=1}^{C} \sum_{i=1}^{\frac{N}{C*S}} \frac{(\overrightarrow{y_j^i} - \overrightarrow{\mu}(\overrightarrow{x_j^i};\theta))^2}{\overrightarrow{(\sigma_j^i)^2}}, \quad \forall\theta \in \Theta, \tag{6.9}$$

where j iterates over the number of chunks, i iterates over the number of SIMD arrays of size S that fit in a chunk, and x_j^i is the i-th SIMD array in the j-th chunk or partition of the data point x.

Once all the data points have been evaluated, the Reduce stage will be invoked on the returned map vector (*Reduction* stage). Note that providing a vectorized function will also reduce the size of the returned Map `std::vector`, which will diminish the cost of the Reduce phase.

All this is possible thanks to the parallelization of the fitting functions and the "templatization" of ROOT's fitting interfaces.

The fitter, the interface between the minimizer and the fitting interfaces, uses the data and the model function to implement the objective function (χ^2, Max. Likelihood), whose implementation is parallelized at task-level by TThreadExecutor and TProcessExecutor (Section 6.4). The fitter also provides the objective function to the minimizer, which can compute internally the gradient using numerical differentiation, or it can be passed by the fitter using the gradient of the model function.

The fitter requires the model function received to be described by a generic interface that accommodates the function evaluation in ROOT, which can be used for fitting and other numerical algorithms. For their convenience, we require the users to pass the model function to the fitter via the `TF1` class. The `TF1` instance is then wrapped in the generic interface, which has been extended for vectorization using templates. ROOT also provides a similar interface for gradient evaluation.

In some cases, we may want to compute externally the numerical derivatives of the model function (e.g. to save computational time). This external derivation is provided by the user by leveraging ROOT's gradient interfaces. The user's derivative function is then propagated to the gradient function of the minimizer of choice via the fitter, and evaluated sequentially or in parallel either at data-level or task-level. The implementations that use these gradient interfaces have also been parallelized and vectorized.

In the following sections we explore in more detail the challenges confronted, technical details and the implementation decisions during the exploitation of data-level parallelism and task-level parallelism of the fitting in ROOT.

6.3 Vectorization

ROOT provides two data classes for the representation of the fitting data: binned and unbinned. These classes are filled with the ROOT user data object, such as histograms or trees, and provide the necessary interfaces for the fitter to manipulate the data.

Both the binned and unbinned data classes have been adapted to exploit data-level parallelism. This type of parallelism is implemented in the fitting using VecCore (Section 5.2), to adapt the fitting interfaces to vectorize the operations in the evaluation of the model function provided by the user.

For this, we need to modify the way ROOT stores the fitting data in the data structures mentioned above (Section 6.3) and to provide padding mechanisms for safe lecture and storage of the data in the SIMD types used by the vectorized fitter (Section 6.3). Furthermore, we need to template the fitting interfaces and to vectorize the implementation of the objective function (Section 6.3) to read and operate on SIMD arrays.

From Array of Structures to Structure of Arrays

Efficient vectorization requires changes in the way ROOT stores data coordinates. Prior to these changes, ROOT stored these data in an object-oriented way; that is, for each data point, all its coordinates were stored consecutively, in arrays of structures (AOS). For example, for a three-dimensional data point, ROOT stored the three coordinates x, y, z contiguously in memory, that is, the data array was stored as

$$\overrightarrow{(x, y, z)}_n = (x_0, y_0, z_0, ..., x_n, y_n, z_n).$$

When loading memory into a SIMD array, we provide the start address of the data and we gather as much contiguous data as the size of the SIMD array indicates. This means that, for correct loading of the coordinates in SIMD arrays, the points' coordinates should be stored in contiguous positions in memory; otherwise the memory loaded into the SIMD array might be incorrect or corrupted. As we want to evaluate several points per coordinate at a time, we need to store the N coordinates in N aligned vectors of data points, or structures of arrays (SOA), that is

$$\overrightarrow{(x, y, z)}_n = (x_0, ..., x_n, y_0, ..., y_n, z_0, ..., z_n).$$

Padding

SIMD instruction sets distinguish between two kinds of instructions: *packed* and *scalar*.

A scalar instruction is evaluated on one data element at a time while a packed instruction is evaluated over a full SIMD array. Note that these are the only two options available: if the datum size is not a multiple of the SIMD array size for a certain type, the last (incomplete) array will either be loaded as a set of scalars or require padding to load it as a packed SIMD array.

We have developed a padding mechanism in order to avoid loading wrong memory addresses into SIMD arrays, i.e. filling the empty positions of the SIMD array with the last data element if the remaining data to load is less than the SIMD vector size. The choice of filling with the last data element is not arbitrary: it is a requisite for the padding to be composed of values that guarantee safe evaluations. This avoids invalid operations (e.g., dividing by zero when padding with the default value for the type) and guarantees that the operation to be performed will be correct. This padding mechanism has been implemented for the coordinates, and for the values and the error in the case of binned data representations.

Vectorization of ROOT Math interfaces

ROOT Math interfaces to functions and numerical algorithms have been designed to accept only double precision types. Instead of adding extra overloads to be able to operate with SIMD arrays, we decided to template all fitting and function interfaces, retaining the existing classes as an alias of the new templated ones instantiated on `double`. Figure 6.2 depicts the structure of the function interfaces in ROOT prior to their templatization, as an example of the complexity of this task.

VecCore is the SIMD library of choice to express vectorization in ROOT, for the reasons explained in Section 5.2. We leverage VecCore types to specify the new type

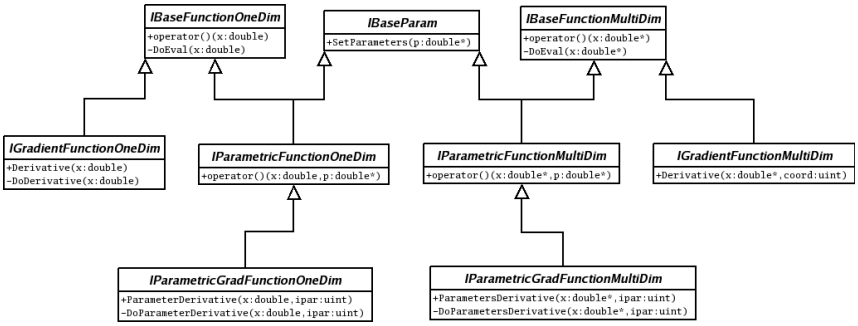


Figure 6.2: `ROOT::Math` function interface structure.

`ROOT::Double_v` that allows us to work with SIMD types inside ROOT and expose them for user-specified functions.

Once we obtain the user function, we provide full compatibility with most of the old scripts. This means that, in order to benefit from a vectorized evaluation of the fitting, users will only need to vectorize their function, which in some cases may only require changing the data and return type. The user can also easily create vectorized functions by using TMath methods or `TFormula` (Section 7.3) instances.

In order to achieve this backward compatibility, and as a fundamental class of the fitting process, we also developed templated overloads of the `TF1` function type constructors, and implemented the necessary internal mechanisms for evaluation on SIMD types, as discussed in Section 5.3.

6.4 Multiprocessing and multithreading

Our task-level parallelization of the fitting process targets the loop over the data elements in each one of the fitting functions. This approach requires making all the invoked functions within the loop body scope thread-safe, removing all dependencies on previous iterations, and adapting the code to the interface provided by TExecutor.

As a requirement from TThreadExecutor and TProcessExecutor's interface, the body of the loop evaluation on the datum has been transformed into a callable that evaluates the data on a single point (or SIMD array size points when applying vectorization) and returns the evaluation result. This callable plays the role of the mapping function. The reduction function, another callable, will be adapted to each fitting model, as well as the returned types, from the mapping function. An example of this process is provided in Listing 15.

For performance, we also propose an heuristic chunking function (Listing 16) based on empiric experience and the size of the data that, additionally, limits the potential number of tasks within a fixed interval. The objective is to generate a

```
//Halves accumulation loop
unsigned sum;
for(unsigned i=0; i<N; i++){
    sum+=i/2;
}

// Same loop parallelized with TThreadExecutor
auto singleEvaluation = [](const unsigned &i) {
    return i/2;
}
auto reduceFunction = [](const std::vector<unsigned> &v) {
    return std::accumulate(v.begin(), v.end(), 0U);
}

ROOT::TThreadExecutor pool;
pool.MapReduce(singleEvaluation, ROOT::TSeq<unsigned>(N), reduceFunction);
```

Listing 15: Porting a traditional loop to TThreadExecutor. We implement in one callable (*singleEvaluation*) the operation to perform on a single point, and in another one (*reduceFunction*), the function to reduce the multiple partial results. Finally, we create an instance of TThreadExecutor and call its `MapReduce` function, which will receive as parameters both callables and, in this case, a sequence of unsigned integers.

number of chunks which facilitates attaining work balancing to the TBB scheduler, but which is not so small as to be influenced by the TBB tasks creation overhead. We could improve this function by timing both the evaluation of a single data element and the overhead of creating a task in TBB. This proposal would make sense with data of large size, as the time to evaluate a single additional data point is negligible.

```
unsigned setAutomaticChunking(unsigned nEvents){
      auto ncpu  = std::thread::hardware_concurrency();
      if (nEvents/ncpu < 1000) return ncpu;
      return nEvents/1000;
      //return ((nEvents/ncpu + 1) % 1000) *40 ; //arbitrary formula
}
```

Listing 16: Heuristic function to set an appropriate number of tasks, dependent on the size of the datum (`nEvents`) and limiting the maximum number of tasks to process.

We also defined the new type `ROOT::Fit::ExecutionPolicy`, representing the different execution policies, with the options described in Listing 17.

In order to specify the execution policy when calling the fit from a histogram, we extended its interfaces with the options `"SERIAL"`, `"MULTITHREAD"`—not necessary if ImplicitMT is enabled—and `"MULTIPROCESS"`.

```
namespace ROOT{
   namespace Fit{
      enum class ExecutionPolicy {
         kSerial,
         kMultithread,
         kMultiprocess
      };
   }
}
```

Listing 17: Execution policies available for fitting, representing the serial case, the multithreaded case and the multiprocessed case.

6.5 Case example: A typical fit

As an example of how the changes introduced in ROOT can contribute to accelerate the fitting, we choose a fit of the diphoton invariant mass distribution resulting from a Higgs boson (Figure 6.3). Listings 18 and 19 offer a comparison of the scalar code for this benchmark fit against the code needed for a fully parallelized (multithreading plus vectorization) implementation of the same fit.

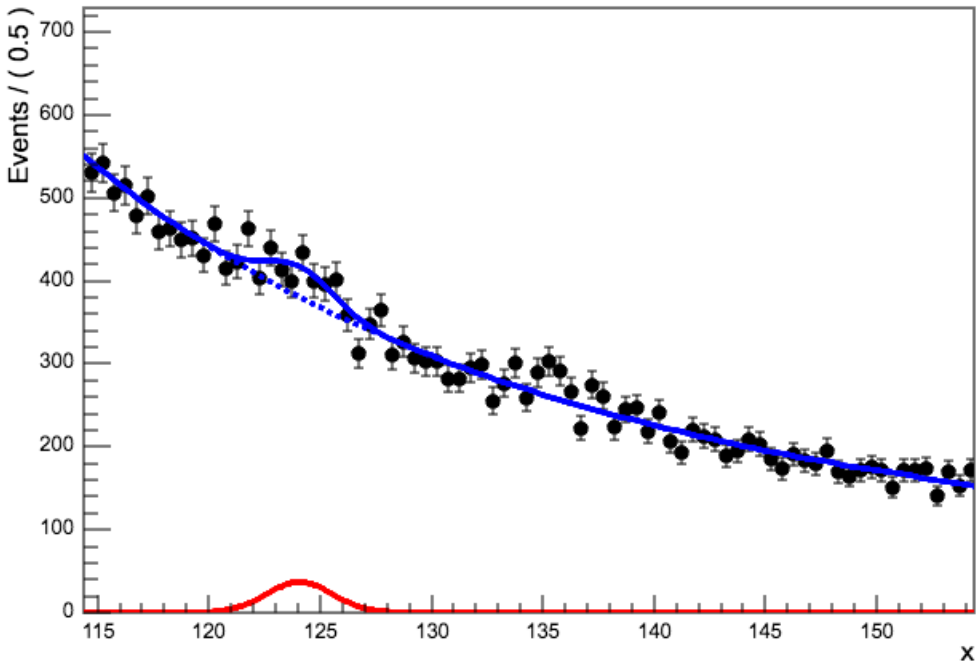


Figure 6.3: Fitting of the invariant mass distribution for the diphoton system resulting from the decay of a Higgs boson.

In Listing 19 we provide the parallelization and vectorization while maintaining

```
1  //Example Fit: Implementation of the scalar function
2  double func(const double *data, const double *params)
3  {
4     return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
5            params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
6            ((*data) * (0.01)) * ((*data) * (0.01))));
7  }
8
9  auto f = TF1("fScalar", func, 100, 200, 4);
10 f.SetParameters(1, 1000, 7.5, 1.5);
11 TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
12 h1f.FillRandom("fScalar", 1000000);
13 h1f.Fit(&f);
```

Listing 18: Traditional scalar implementation.

```
1  //Example Fit: Implementation of the vectorized function
2  ROOT::Double_v func(const ROOT::Double_v *data, const double *params)
3  {
4     return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
5            params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
6            ((*data) * (0.01)) * ((*data) * (0.01))));
7  }
8
9  // Enable implicit parallelization
10 ROOT::EnableImplicitMT();
11
12 //This code is totally backwards compatible
13 auto f = TF1("fvCore", func, 100, 200, 4);
14 f.SetParameters(1, 1000, 7.5, 1.5);
15 TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
16 h1f.FillRandom("fvCore", 1000000);
17 h1f.Fit(&f);
```

Listing 19: Vectorized plus parallelized implementation.

the previous programming model. This is an important achievement, as users can now benefit from multiple levels of parallelization without modifying their old scripts. Starting from line 9 in Listing 18, and line 13 in Listing 19, the code related to the fitting operations is exactly the same. The only changes required for fitting the data in the vectorized and parallelized case —Listing 19—are additions on top of the current workflow.

First, in order to profit from the implicit task parallelization, the user only needs to call `EnableImplicitMT` (line 10 of Listing 19). This will activate all the mechanisms in ROOT that provide thread safety and parallelize all the operations that allow implicit multithreading in ROOT, such as the fitting. For vectorization, the user needs to provide a vectorized function, which in this case only implies changing the data parameter type and the return type —to the new `ROOT::Double_v` type, representing a SIMD array of `double`—as shown in line 2 of Listing 19.

We perform three different types of fits, χ^2 (Section 6.1) and Poisson likelihood in the binned case and Maximum Likelihood (Section 6.1) in the unbinned case, to the invariant mass distribution, studying the speed up in each one of these cases.

In all experiments, the final sum of the partial results (the *Reduction* stage in Figure 6.1) is computed sequentially and in the same order for the sake of numerical reproducibility. We measured the fitting times normalized to the number of objective function calls made by the fitter, as the nature of the minimization problems makes the number of calls fluctuate between examples, due to numerical precision, and influence the measured times.

Individual evaluations of the objective function

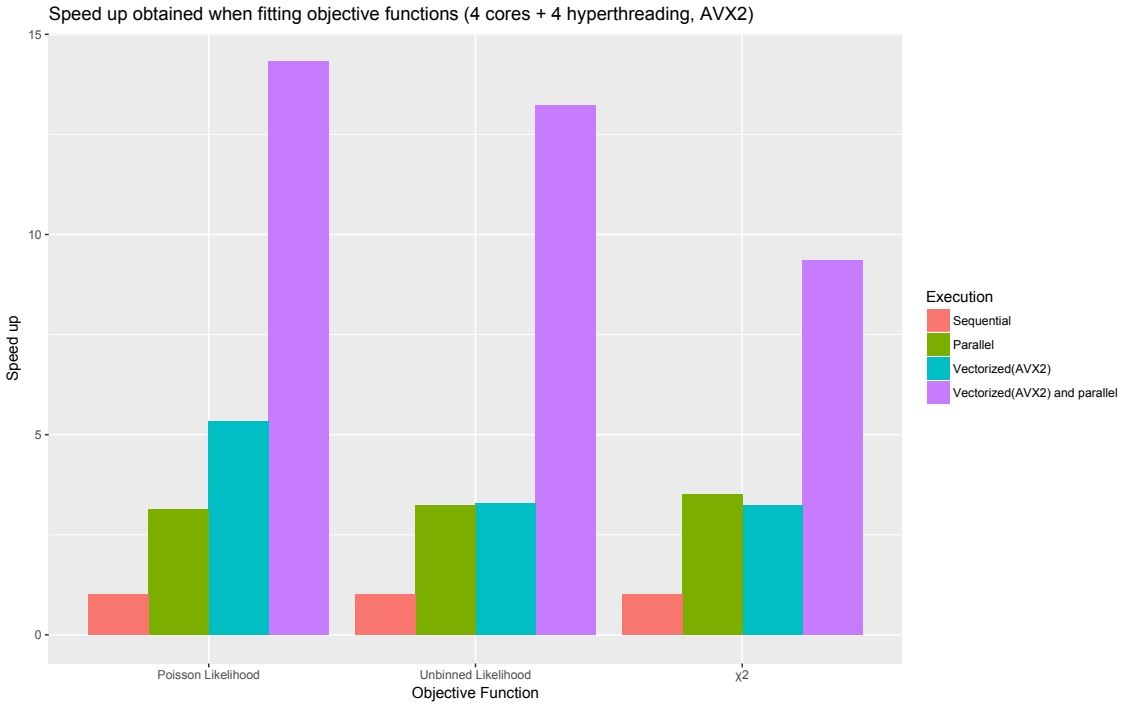


Figure 6.4: Speed up obtained for the evaluation of each type of fit evaluated over 120001 bins in the binned case and 120001 points of in the unbinned case.

Figure 6.4 and Table 6.1 expose the results obtained in single call evaluations for each model, compiled with gcc 6.2 in a 4-core Broadwell desktop server with 8 GB of RAM. We observe close to ideal results both when exploiting task-level parallelism (Parallelized) and data-level parallelism (Vectorized). The best results are obtained with the combination of the two levels of parallelism, with a remarkable speed up

Objective function/Model	Time (μs)	Speed up
χ^2/Sequential	752.14	1
χ^2/Vectorized	232.60	3.23
χ^2/Parallelized	214.82	3.50
χ^2/Vectorized +Parallelized	80.34	9.36
Poisson Likelihood/Sequential	1283.197	1
Poisson Likelihood/Vectorized	240.83	5.33
Poisson Likelihood/Parallelized	409.62	3.13
Poisson Likelihood/Vectorized +Parallelized	89.58	14.33
Unbinned Likelihood/Sequential	1188.25	1
Unbinned Likelihood/Vectorized	361.99	3.28
Unbinned Likelihood/Parallelized	367.04	3.24
Unbinned Likelihood/Vectorized +Parallelized	89.85	13.22

Table 6.1: Measured times and speed up for individual calls of the objective functions (4-cores Broadwell, 8GB RAM, AVX2). Measured by the microbenchmarking library Google Benchmark [20], averaging thousands of calls. Evaluated over 120001 bins in the binned case and 120001 points of in the unbinned case.

even in the χ^2 evaluation, though the speed up there is lower than for the other two models.

Notice the super-linear speed up when vectorizing the Poisson likelihood. This is due to improvements (e.g fused multiplication and addition) in some vectorized operations such as the computation of the logarithm, which in this case is called several times per data point evaluation.

A complete fit

For a more relevant use case, we benchmark a complete fit. A fit is a minimization process that consists in several calls to the objective function by a minimizer—external in ROOT's case—that will make educated guesses on the estimators until it reaches an optimal solution.

The interactions between the fitting model functions and the minimizer are handled through the fitter, the ROOT class that manages the fitting process. Because the calls to the fitter interface can add overhead in addition to the evaluation of the function, benchmarking a complete fit is important.

Figure 6.5 and Table 6.2 report the speed up attained by the parallel fitting, compiled with gcc 6.2 on a desktop server equipped with a 4-core Haswell processor and 8 GB of RAM, and distinct sets of vectorization instructions and evaluation functions.

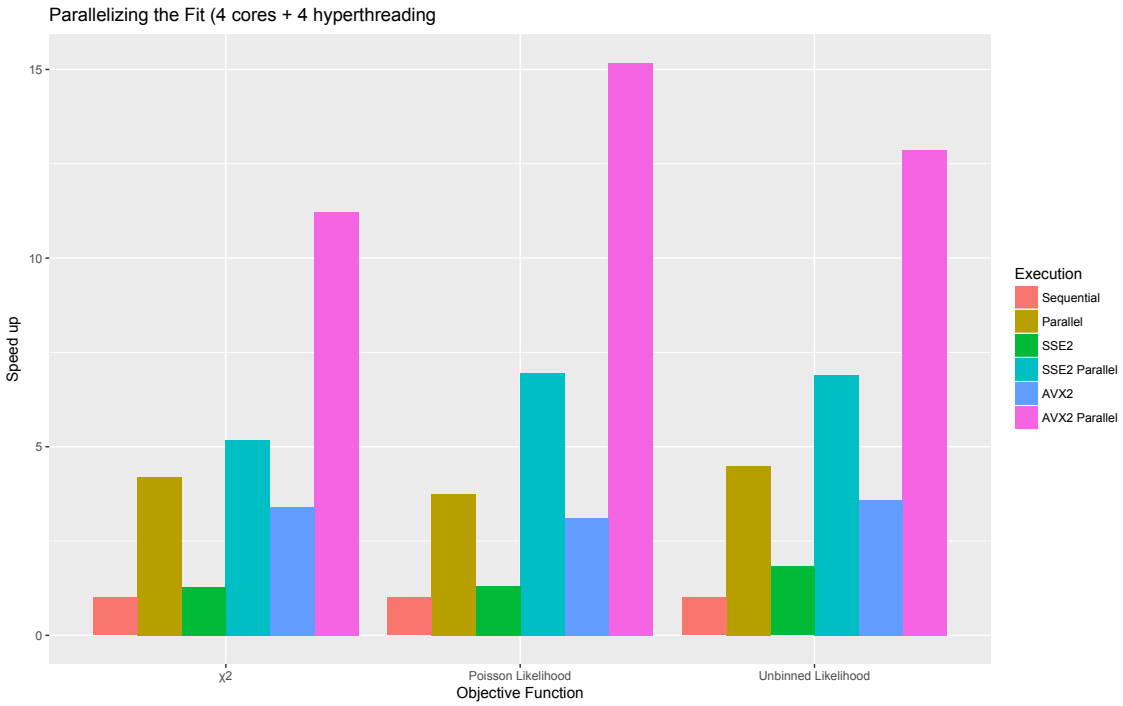


Figure 6.5: Performance of the fit (4-cores Broadwell, 8GB RAM). Evaluated over 120k bins in the binned fits and 120k points in the unbinned fit.

In Figure 6.6 and Table 6.3, we show the scaling of multithreaded operations on a 14-core Xeon processor with AVX2 and 64 GB of RAM. While it scales in the multithreaded scalar case, suboptimal vectorization limits the total speed up.

It is also interesting to show how different compilers behave when vectorizing the evaluation of the model function. We asses the impact of distinct compiler technology in the following

Observing Figure 6.7, we find out that icc 17 performs a more aggressive auto-vectorization than the other compilers, matching the explicitly vectorized times. For a fairer comparison, avoiding autovectorization when compiling, the results between compilers are similar, with icc slightly outperformed by clang and gcc (Figure 6.8).

6.6 Conclusions

In this chapter we demonstrate that, with the appropriate tools, we can exploit parallelism at several levels when fitting models in ROOT and, at the same time, preserve the user interfaces. We have shown how, as a result, the users can now vectorize and parallelize their original code and profit from a significant speedup.

Objective function/Model	**Time (s)**	**Speed up**
χ^2/Sequential	12.11	1
χ^2/Parallel	3.20	4.19
χ^2/SSE	8.54	1.29
χ^2/SSE2 parallel	4.74	5.18
χ^2/AVX2	4.18	3.41
χ^2/AVX2 parallel	1.60	11.23
Poisson Likelihood/Sequential	16.05	1
Poisson Likelihood/Parallel	4.24	3.74
Poisson Likelihood/SSE	15.25	1.3
Poisson Likelihood/SSE2 parallel	2.33	6.95
Poisson Likelihood/AVX2	6.57	3.11
Poisson Likelihood/AVX2 parallel	1.40	15.18
Unbinned Likelihood/Sequential	3.09	1
Unbinned Likelihood/Parallel	0.82	4.48
Unbinned Likelihood/SSE	1.69	1.82
Unbinned Likelihood/SSE2 parallel	0.37	6.90
Unbinned Likelihood/AVX2	0.86	3.59
Unbinned Likelihood/AVX2 parallel	0.23	12.86

Table 6.2: Times and speed up of the fit (4-cores Broadwell, 8GB RAM) for different pairs of objective function and execution policy. Evaluated over 120k bins in the binned fits and 120k points in the unbinned fit. Complements Figure 6.5.

Cores	2	4	6	8	10	12	14
Parallel	1.78	3.43	5.18	6.59	8.21	9.54	10.9
Ideal Parallel	2	4	6	8	10	12	14
Parallel + vectorized	4.80	8.99	12.66	16.08	19.24	22.51	24.57
Ideal Parallel + vectorized	8	16	24	32	40	48	56

Table 6.3: Execution time recorded in seconds for a complete fit (Haswell, 14 cores, 8GB RAM) with an increasing number of cores.

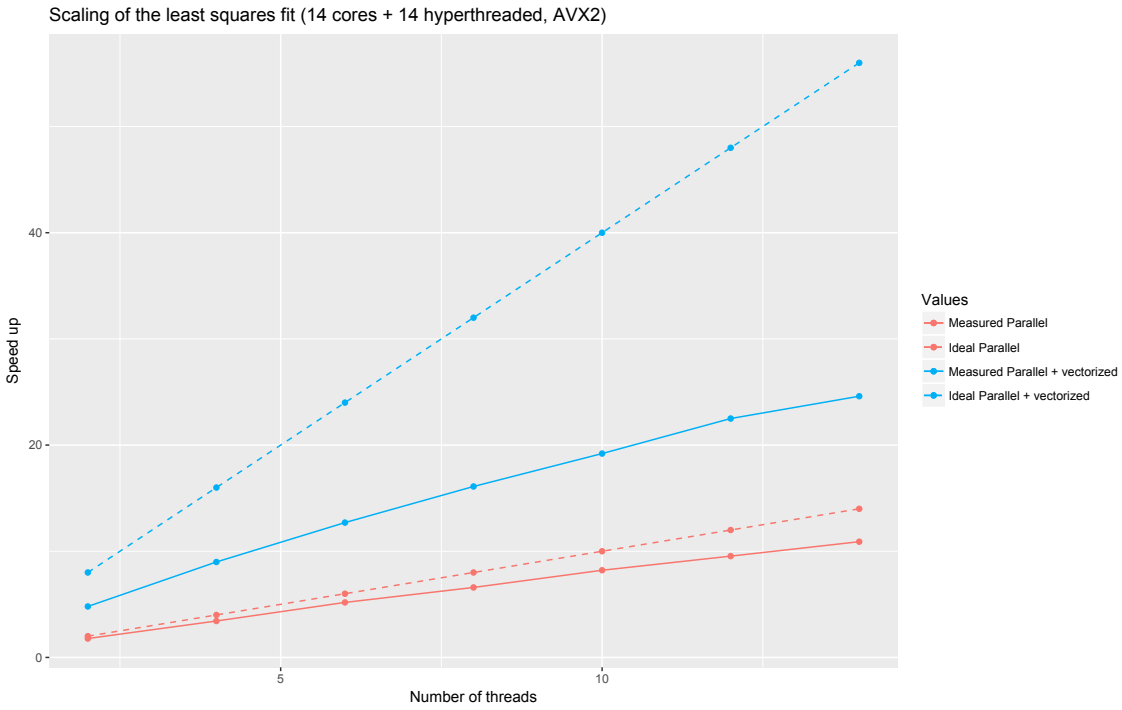


Figure 6.6: Scaling of the fit with an increasing number of cores.

We described the necessary changes to improve the performance of the vectorization introduced in Section 5, from structuring our data in a Structure of Arrays to achieve data alignment, to providing safety mechanisms, such as that described for the padding. In addition, we applied the knowledge obtained in Section 4.6 to define a heuristic that contributes to balancing the parallel workload. Finally, we parallelized the evaluation of the different objective functions by deploying the executors and adapting the processing of the event loop to a MapReduce model.

We utilized a real use case to benchmark the parallelization at task-level (multithreading) and data-level of the unbinned maximum likelihood, the Poisson likelihood and the least-square methods, on individual evaluations and on a complete fitting process for each of the objective functions.

Results differ between objective functions, but in all the benchmarked cases we obtained good acceleration factors when vectorizing, and remarkable ones when parallelizing at task-level. The combination of both task-level and data-level parallelism produces the best results, and both speed up and execution times are consistent across compilers.

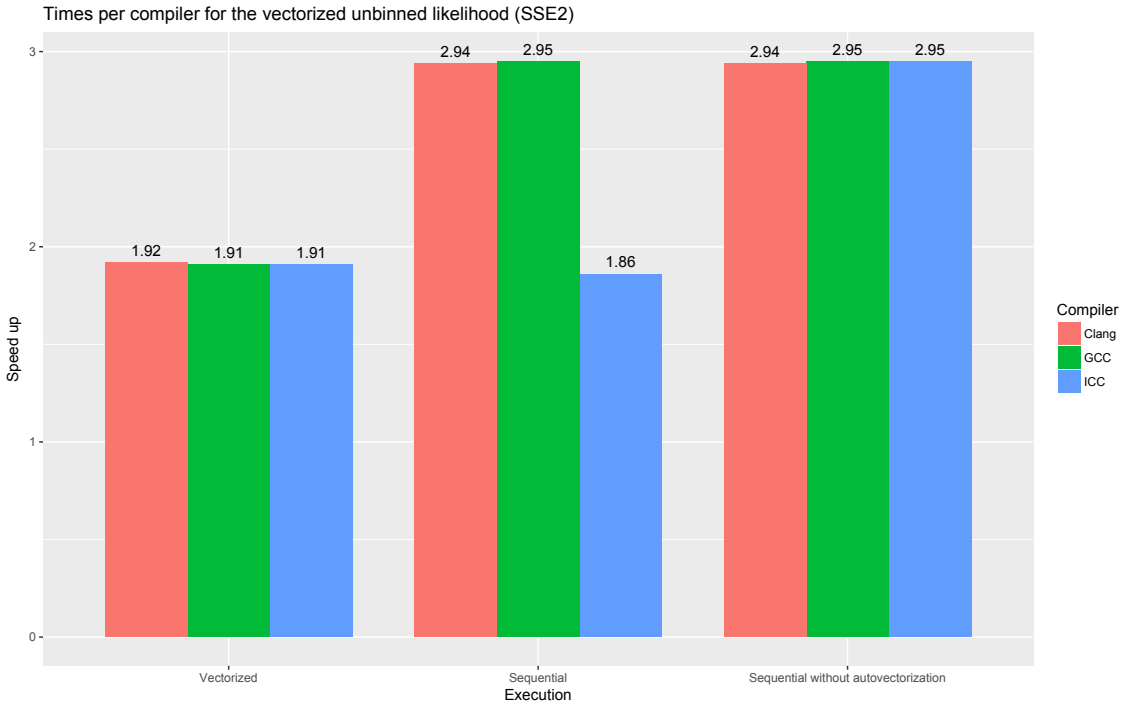


Figure 6.7: Compiler comparison (gcc 6.2, clang 3.8, icc17) for the execution time of the maximum likelihood evaluation, sequentially and vectorized with SSE2. Carried out in double precision.

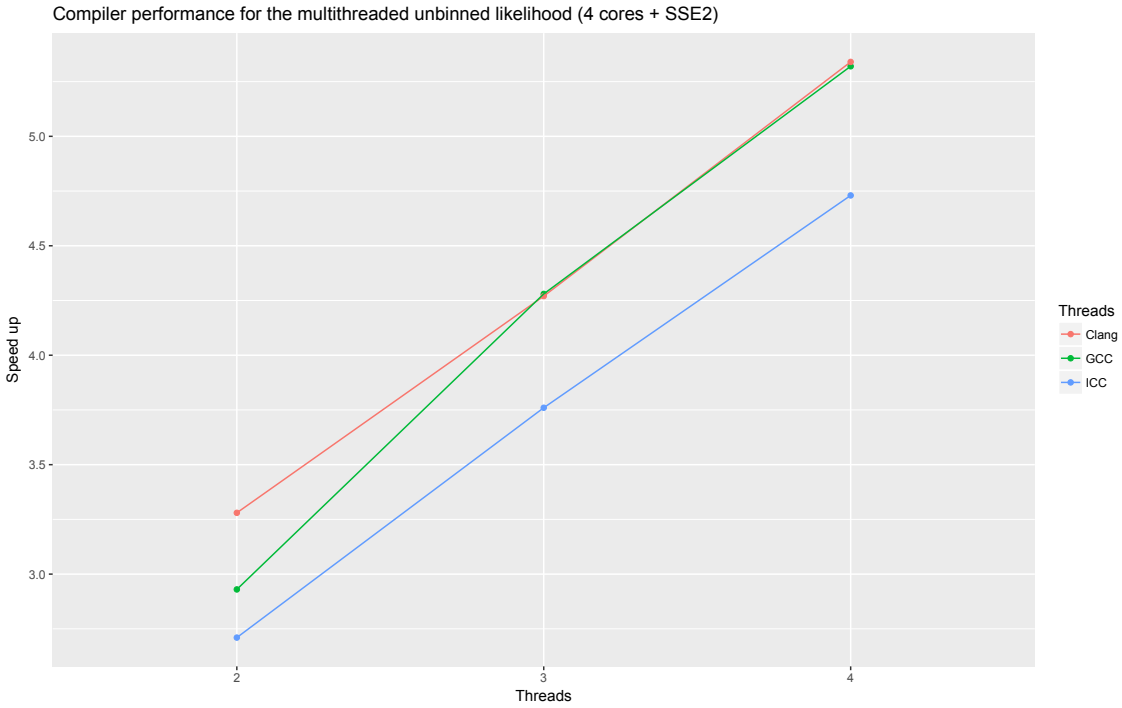


Figure 6.8: Performance scaling for the maximum likelihood SSE2 evaluation.

Chapter 7

Performance studies

The parallel tools and libraries developed in this thesis yield significant performance improvements for ROOT and align with the current modernization effort of ROOT's code. These tools have been extensively deployed throughout ROOT: training of Deep Neural Networks, parallel merging of files, parallel processing of ROOT trees, vectorizing the fit, vectorizing mathematical functions and user-defined formulas, parallel writing of histograms, parallel evaluation of the event loop in RDataFrame, and so on.

This chapter serves as a showcase for some of the performance improvements granted by the utilities for parallelism at task-level and data-level introduced in previous chapters. Section 7.1 describes the improvements obtained when parallelizing ROOT's standalone tool for merging ROOT files, hadd, using the multiprocess executor of ROOT, TProcessExecutor. In Section 7.2, we introduce a code exploring a potential model of new Physics, and we benchmark the performance of its RDataFrame implementation against the original ad-hoc highly optimized version of the same code. Finally, Section 7.3 compares and reports on the performance of vectorizing mathematical functions at runtime via `TFormula`.

7.1 Merging of ROOT files

Merging ROOT files is a common but expensive operation. ROOT offers a tool for this purpose, *hadd*, that could benefit from parallelization. Concretely, hadd is a standalone application distributed with ROOT whose objective is to merge several ROOT files containing ROOT trees or histograms into a new file, by exploiting the `Merge()` method of these classes and ROOT's file-managing utilities, e.g. `TFileMerger`.

This is an easily parallelizable procedure, for instance, by relying on recursive hierarchical merging. The most obvious implementation is to perform the merge

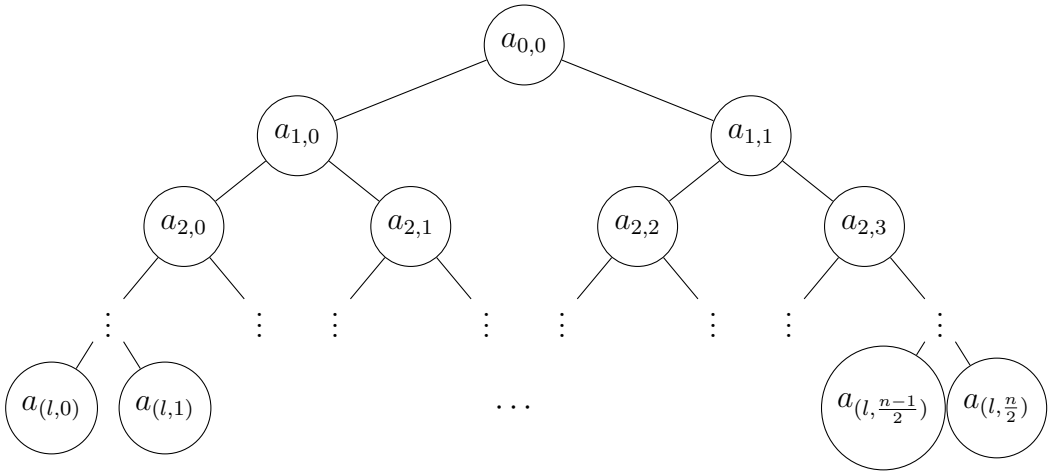


Figure 7.1: A full binary tree of $\frac{n}{2}$ leaves and l levels, where $l = \log_2 \frac{n}{2}$.

following a binary tree structure. Figure 7.1 depicts a full binary tree of $\frac{n}{2}$ end leaves. This tree can be used to merge $n = 2^{k+1}$ files with $m = 2^j \geq n$ processing units. In a tree such as that depicted, each node (and leaf) $a_{i,j}$ constitutes a task that merges two files into one. These tasks are computed concurrently. Parallelizing the merging of the ROOT files in this way would grant a potential speed up in the number of merges of

$$S_u = \frac{n}{2^{\log_2 n - \log_2 m} - 1} = \frac{n}{\frac{n}{m} - 1} = \frac{m\,n}{n - m},$$

where the sequential tree for $n = 2^{k+1}$ files has

$$1 + 2 + 3 + 4 + 8 + \cdots + \frac{n}{2} = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2k = 2^{k+1} - 1 = n - 1$$

nodes or binary merge operations and $m = 2^i$ is the number of parallel processing elements.

Nevertheless, real world problems are seldom this perfect. For instance, the number of files may not be a power of two, and then the shape of the tree does not correspond to that of a full binary tree. In addition, the number of processing units may not be a divisor of the number of files, and, in that case, it is better performance-wise to distribute the files such that the leaf operations do not constitute a binary merge. As an example, if we have 5 processing units available and $5n$ files to process, the resulting binary tree looks as represented in Figure 7.2.

Due to the particularities of the `Merge()` operation in ROOT, in which a collection of ROOT objects are merged into the first element of the collection, the full combination of the objects requires $n-1$ merges for n files. In this case, with n files

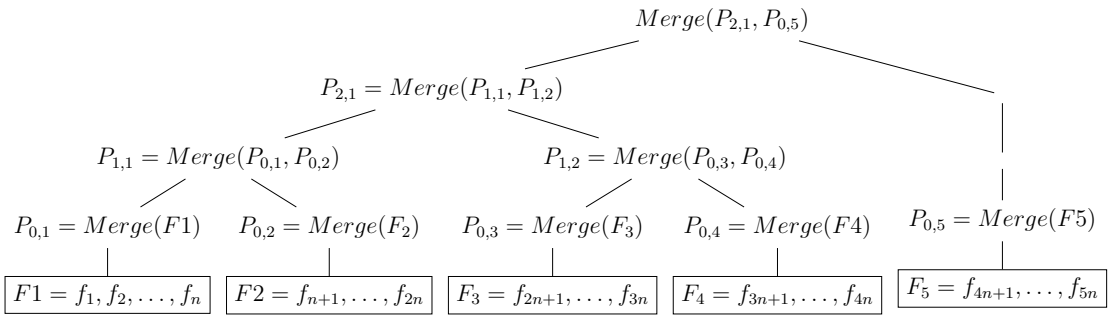


Figure 7.2: A binary tree of for the merging of $5n$ files using 5 processing units, where the number of leaves of the tree is limited by the number of processing units.

distributed among the m processing units, and each one of the leaves of the tree performing a sequential merge of $\frac{n-1}{m}$ files rounded up ($\frac{n+m-1}{m} - 1$ until the last leaf, where we merge the remaining files), the potential acceleration factor that we can obtain in the number of merges is:

$$S_u = \frac{n-1}{\frac{n+m-1}{m} - m + \log_2 m}.$$

However, while merging in a binary tree is theoretically the most efficient option, in practice, the overhead caused by the implementation of a partial binary file merging operation, and the limitations of the I/O subsystem (opening a file, creating a file merger, writing results on disk), make this option less practical. Instead, we adopt a two-step MapReduce solution where we distribute the files uniformly between the p workers, and then we perform partial reduction in each of the workers ("greedy" reduce), only to reduce these partial results one last time to obtain the final result (Figure 7.3).

The theoretical maximum speedup that we can obtain when merging files in two steps is governed by the equation

$$S_u = \frac{n-1}{\frac{n}{m} + m - 2}, \tag{7.1}$$

where n is the number of items to process and m is the number of processing workers.

We developed this last solution in a parallel MapReduce implementation of hadd, relying on the multiprocess executor of ROOT, TProcessExecutor, for merging ROOT objects stored in ROOT files. Most frequently, a ROOT file contains

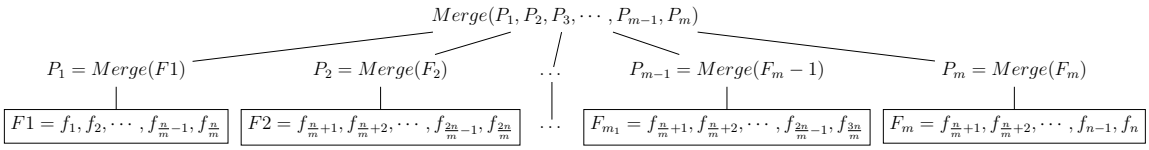


Figure 7.3: Merging of n files in two steps, using m threads.

data in either ROOT trees or ROOT histograms. We test the parallel merging of histogram files with a file from the official CMS data quality monitoring repository, a real case with around 12.5 MB of hundreds of histograms in a complex hierarchy of directories, which should result in a large workload. We merge in parallel the same file duplicated N times for balancing purposes.

Figure 7.4 showcases the speed up obtained with respect to the sequential execution when merging a different number of files with an increasing number of workers. The measurements have been obtained in an Intel Core i7-4790 desktop server with 4 physical cores (8 logical threads) at 3.6 GHz and 8 GB of RAM [26], measured with the stantard high resolution clock of C++ and each one averaged over several executions. By default, TProcessExecutor deploys as many workers as logical cores—in this case 8—which, as we observe, does not positively affect performance.

Figure 7.5 analyzes each result separately and compares them against a theoretical reference, the maximum speed up given by Equation (7.1) in combination with the sequential execution measurements. We notice that, when the number of files increases, the speed up rapidly becomes superlinear. Table 7.1 and Figure 7.6 display the throughput (execution time per file merged) of the merging process for each one of the data points collected. We can observe that the time required to merge a single file increases exponentially in the sequential case, but this variation in time attenuates as we increase the number of processing units. This is due to the size of the files and a more efficient utilization of resources when merging in parallel, especially caching, and an improved performance of ROOT's I/O subsystem when performing a reduced amount of merges per process.

Finally, in Figure 7.7 we benchmark the performance of an hadd parallel merge of ROOT trees (TTree), comparing the speedup with the reference ideal. In anticipation of a slowdown due to the more complex nature of the TTree and the optimizations introduced in the sequential case, we also compared the measurements in the case that a TTree instance is loaded in memory, emulating this by creating a RAM disk. The results obtained are discouraging in both cases, displaying a similar

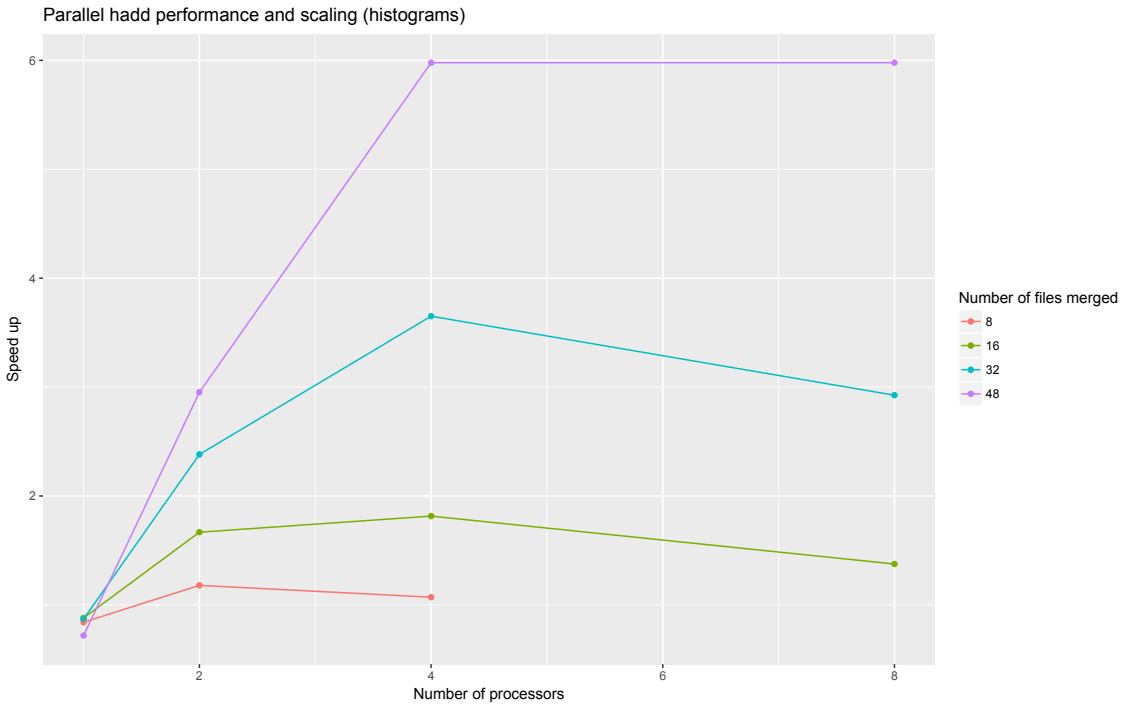


Figure 7.4: Speed up merging with hadd different number of ROOT files containing histograms with an increasing number of cores.

	Sequential	2 workers	4 workers	8 workers
8 Files	3.83 s	3.25 s	3.58 s	-
16 Files	4.56 s	2.73 s	2.51 s	3.31 s
32 Files	8.26 s	3.47 s	2.26 s	2.82 s
48 Files	15.51 s	5.25 s	2.59 s	2.59 s

Table 7.1: Throughput of seconds per file merged in hadd, with an increasing number of files for a varying number of processing units.

undesired slowdown.

Without changing the original user interfaces we can provide remarkable speed up when merging histograms, only by applying a parallel MapReduce pattern to hadd. However, while our implementation shows significant acceleration gains when merging histogram files, it does not perform so well when merging trees.

This parallelization has also been attempted in the multithreaded execution policy, but we have failed to obtain results due to non-thread-safe invocations on global variables required by ROOT's memory manager.

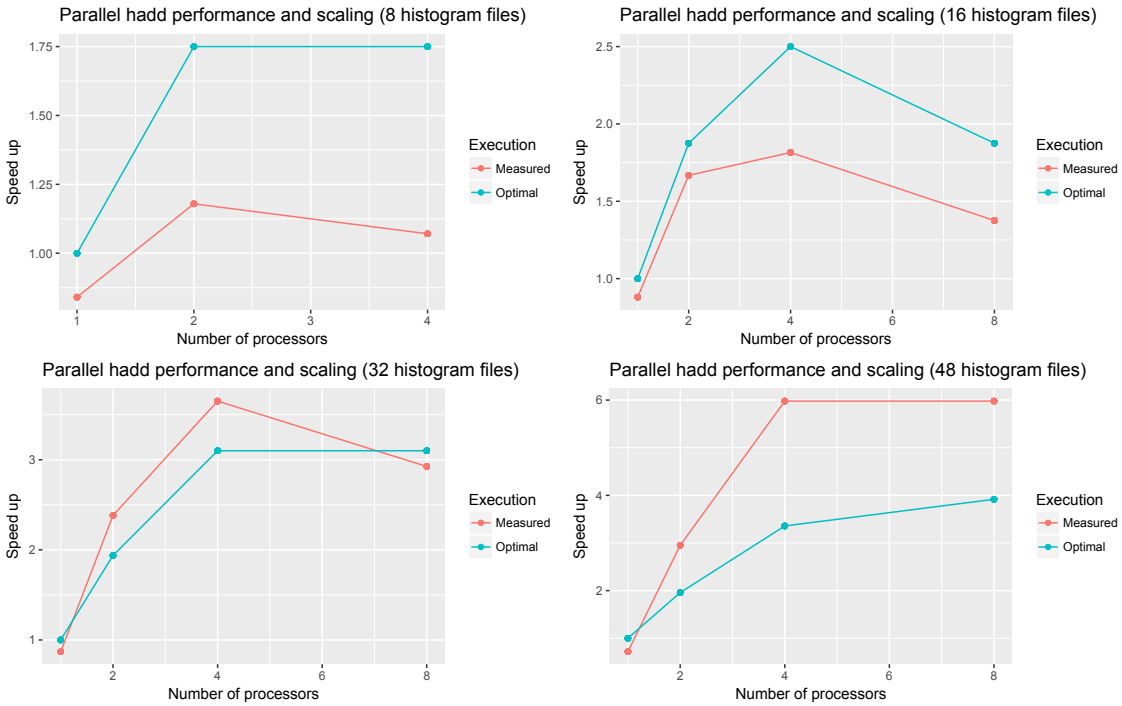


Figure 7.5: Comparison of the speed up obtained merging with hadd different number of ROOT files containing histograms against the theoretical one, with an increasing number of cores.

7.2 Parallel analysis in RDataFrame

RDataFrame (Section 3.5) is one of the most relevant developments in the ongoing modernization effort of ROOT. One of its main features is the support for implicit parallelism, exploiting TThreadExecutor directly, e.g. for the parallel execution of the event loop, and indirectly, when performing frequent operations that support implicit multithreading (IMT) and rely on TThreadExecutor for executing their operations concurrently (e.g. when reading in parallel from a TTree)

RDataFrame introduces a new programming paradigm in ROOT that reduces code complexity by designing its interfaces in a user-friendly way. However, with this design, RDataFrame also introduces several performance disadvantages as a tradeoff for the easier programming model and its PyROOT implementation. This is the case of the non-negligible overhead due to the extensive use of JIT-compiling, or the cost inherent to the expression of the analysis in a modularized, higher-level way (e.g. extra function calls, allocation of partial results).

In this section, we analyze RDataFrame's convenience and benchmark its performance by porting an ad-hoc, highly optimized, parallel code to RDataFrame. This

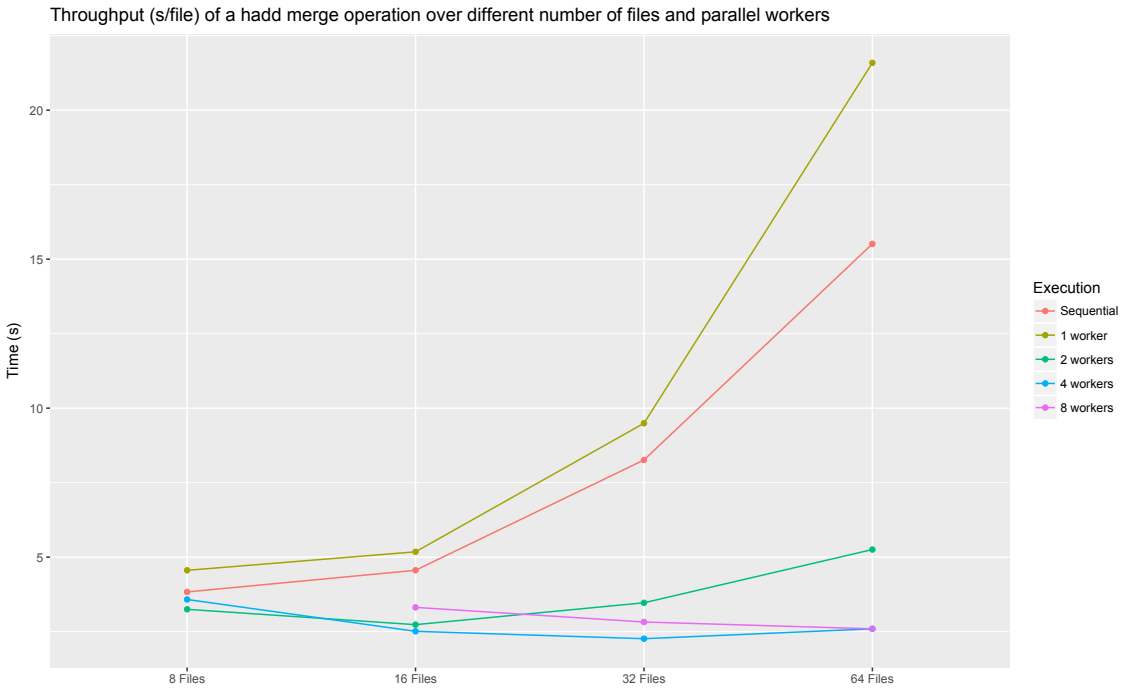


Figure 7.6: Throughput of seconds per file merged in hadd, with an increasing number of files for a varying number of processing units.

code, Totem, is used to explore an experimental model for new physics. Totem is a Monte Carlo event generator that, following this experimental model, performs a physics simulation of very forward QCD-low pT processes according to a parametric model, and writes the relevant data resulting from this simulation into histograms. It consists of a monolithic function that performs all physics-related operations, from simulation to analysis, and the generation of the resulting histograms. Parallelism is implemented by explicitly managing POSIX threads. Despite its complexity, Totem exhibits remarkable performance and scalability, as it has been designed, from inception, for execution in highly parallel architectures.

Porting this code to RDataframe offers the following advantages:

- Implicit parallelism: Instead of managing the threads directly through POSIX directives, RDataFrame's support for implicit multithreading allows the user to benefit from parallelism while writing (thread-safe) sequential code. In addition to the performance boost provided by RDataFrame, the user benefits from the growing number of operations in ROOT that support an IMT-based parallel mode.

- Declarative and functional programming model: Instead of dealing with the

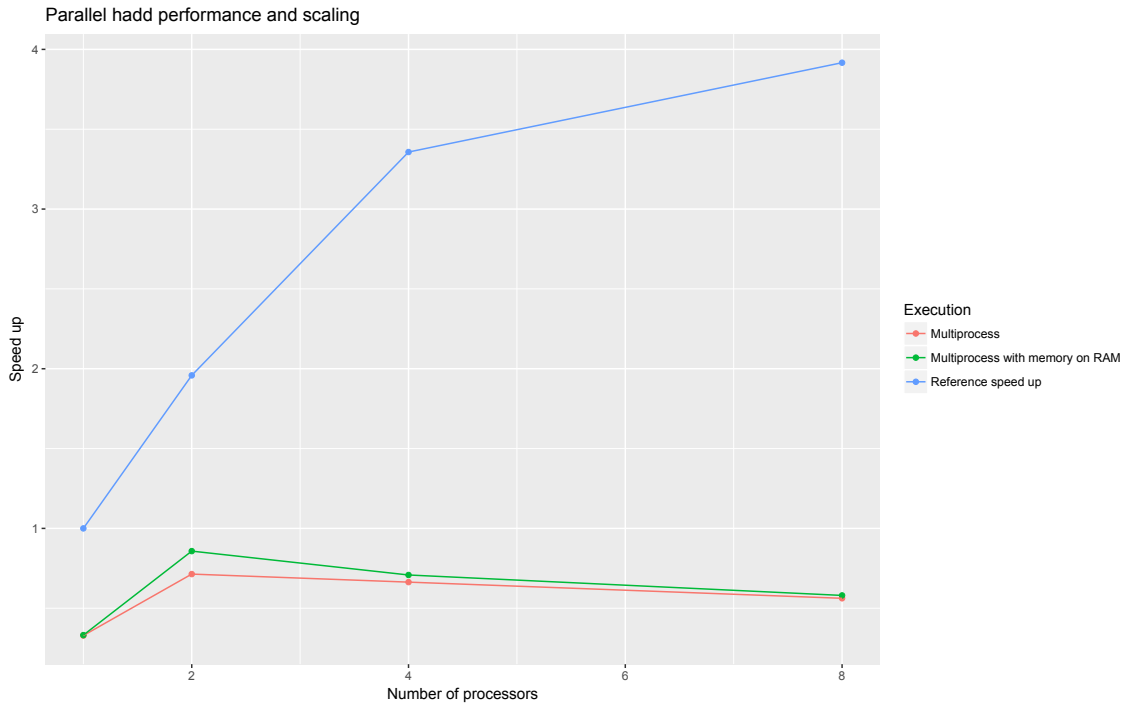


Figure 7.7: Speed Up obtained when merging in parallel with hadd 48 ROOT files containing 48 TTrees.

details of its computation, users express their analyses with RDataframe in terms of chained transformations and actions; that is, they describe which operations the analysis is composed of, instead of expressing how to perform these operations.

- Cleaner code: Due to its functional programming model, RDataframe increases code modularity by factoring transformations and actions out of the event loop.

- Lazy, efficient evaluation: The processing of the graph is deferred to the first access to one action result. This is especially convenient with RDataFrame's organization of transformations and actions in a computational graph, which allows us to reuse the results from shared nodes of the graph.

As Totem's code was developed in a previous version of the software (ROOT 5), we proceed with a two-step translation, guaranteeing reproducibility after each step. First, we port the code to ROOT 6, identifying and refactoring the event loop into a single-event evaluation lambda, and relying on TThreadExecutor for its parallelization, avoiding thread-locality and simplifying the programming model by removing all thread management directives. The second step requires isolating the

simulation from the analysis, encapsulating the former into a single-event evaluation lambda, and expressing the latter by means of RDataFrame, from the execution of the event loop, to the filling of the final histograms.

The original, complex code of Totem and the explanation of the physics behind it are outside of the scope of this section. However, it is helpful to examine RDataFrame's version of the code to showcase its simpler and more convenient programming model. Listing 20 presents the code needed for an RDataFrame implementation of Totem. Where possible, we maintained the original names of the variables. Lines 17-22 define the lambda encapsulating the Monte Carlo simulation of a single event, whose body has been omitted due to irrelevance for the example. This lambda returns, in line 21, the relevant data for the subsequent analysis, encapsulated in a data structure defined previously in lines 2-12. Line 25 enables the implicit multithreading mechanisms in ROOT and, therefore, the automatic multithreaded parallelization in RDataFrame. Lines 28 and 29 declare helpers with the objective of simplifying the code.

RDataFrame dataset can be pictured as a table, where each row represents an entry of the columnar data. In this case, Line 32 of Listing 20 initializes a new RDataFrame of nEvents (entries or rows of the data frame). Lines 35-43 execute the Monte Carlo simulation event loop in parallel and store the relevant quantities resulting from it in new *defined* variables (columns of the data frame). Finally, starting in line 46, we perform our analysis, expressing it functionally and declaratively, in terms of chained actions and transformations. For each one of the final results (histograms), we apply first, as transformations, one or more filtering operations on the data and then, as actions, the generation of each one of the resulting one-dimensional or bi-dimensional histograms.

Figure 7.8 displays the computational graph built for the RDataFrame implementation of Totem's analysis. The evaluation of the event loop in RDataFrame is performed lazily: the first time we access a result from an action (green leaf nodes) of the computational graph, RDataFrame proceeds to the evaluation of the graph for all previously booked results. However, after an event loop, in case we add another result and trigger another event loop, all parts of the graph that are necessary for the generation of these results are re-evaluated. This affects directly the way we express our analyses, as an inefficient booking of the results may result in unnecessary computational redundancy. For instance, after processing the graph for the first leaf, Histo1D-"helast", in case we book the action in the second leaf, Histo1D-"hfr", the computational graph will be re-evaluated for the nodes leading to the "notCollided" (orange tag) one. This is a redundant evaluation that could be avoided by accessing the result of the Histo1D-"helast" node after booking the result for the action in the second leaf, Histo1D-"hfr".

We benchmark the performance of both versions of Totem in two highly-parallel

```
//Data structure to encapsulate the results to return from the simulation
class collisionData {
public:
   collisionData(){}
   collisionData(bool hC, Double_t &r,Double_t &w,Double_t &f):
                  hasCollided(hC), roffset(r), wevent(w), fr(f){}

   bool hasCollided = false;
   Double_t roffset{};
   Double_t wevent{};
   Double_t fr{};
};

// Physics simulations.
// The data filtering and histogram generation with the results has been factored out,
// leaving the most fundamental operations on events.
auto collideOps = [&](unsigned int slot){
    // ....
    // intensive physics computations
    // ....
    return collisionData(hasCollided, roffset, wevent, fr);
}

//Enable implicit multithreading
ROOT::EnableImplicitMT();

//Helpers
auto hasCollided = [](const bool &hasCollided){return hasCollided;};
auto hasNotCollided = [](const bool &hasCollided){return !hasCollided;};

//Initialize an empty dataframe with nEvents rows
ROOT::Experimental::TDataFrame d(nEvents);

//Define the new per-event variables from the results obtained per-event
auto d2 = d.DefineSlot("collisionData", collideOps)
            .Define("hasCollided",
                    [&](const collisionData &colData){return colData.hasCollided;},
                    {"collisionData"})
            .Define("roffset", [&](collisionData &colData){return colData.roffset;},
                    {"collisionData"})
            .Define("wevent", [&](const collisionData &colData){return colData.wevent;},
                    {"collisionData"})
            .Define("fr", [&](const collisionData &colData){return colData.fr;},
                    {"collisionData"});

//Generate 1D and 2D histograms to visualize the results.

auto notCol = d2.Filter(hasNotCollided, {"hasCollided"});

auto helast = notCol.Filter([](const double &offset){return offset < 1.01;},
                            {"roffset"})
                     .Histo1D<double>(TH1D("helast", "elastic collisions in proton range",
                                           100, distmin, 1.01),
                                      "roffset");
auto hfr = notCol.Histo1D<double>(TH1D("hfr","p-p elastic", ntbins, tbins),
                                  "fr", "wevent");
auto hfr1 = notCol.Filter([](const double &offset){return offset < 1.01;}, {"roffset"})
                  .Histo2D<double>(TH2D("hfr1", "hfr1", 100, 0, 0.5, 100, distmin, 0.7),
                                   "fr", "roffset", "wevent");
auto hfr2 = notCol.Histo2D<double>(TH2D("hfr2", "hfr2", 100, tmin, tmax, 100, distmin, 1),
                                   "fr", "roffset", "wevent");
auto hcoll = d2.Filter(hasCollided, {"hasCollided"}).
                .Filter([](const double &offset){return offset < 1.01;},{"roffset"})
                .Histo1D<double>(TH1D("hcoll", "inelastic collisions in proton range",
                                      100, distmin, 1.01),
                                 "roffset");
```

Listing 20: RDataFrame implementation of Totem.

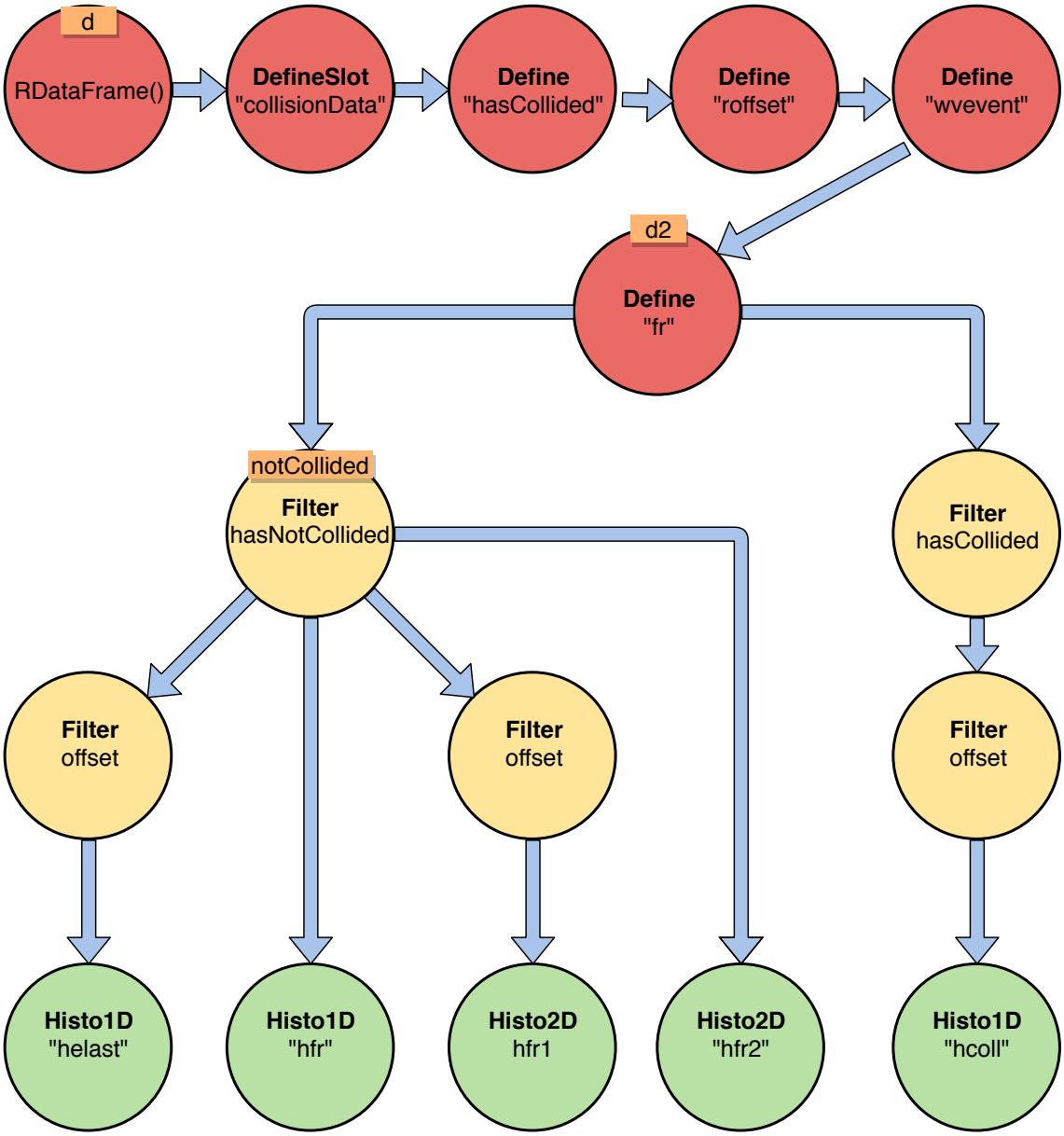


Figure 7.8: Computational graph of Totem's analysis. Red nodes represent definitions and declarations, yellow nodes represent transformations (Filter) and green nodes represent actions (generation of histograms). The orange labels indicate the intermediate variables, figuratively pointers to a graph node, we declare in our code.

computers. The first benchmarking machine is an Intel Xeon E5-2683 v3 [32] server composed of two NUMA domains with 14 physical cores and 28 logical threads each at 2.00 GHz with 32 GB of RAM per NUMA node. The second, more extreme, many-core computer is an Intel Xeon Phi 7210 [31] server composed of two sockets with 64 cores each at 1.3 GHz (supporting hyperthreading) and 110 GB of RAM in total. In order to avoid confusion, we refer to this computer by its former code name: Intel Knights Landing (KNL). The times obtained have been measured with the stantard high resolution clock of C++, each one averaged over several executions.

Table 7.2 displays, for both the original code and for the RDataFrame implementation, the execution time results obtained for a Totem execution in the Xeon server. Table 7.3 presents the results of the execution in KNL.

Number of threads	RDataFrame (seconds)	Original (seconds)	Ratio
1	2932.95	2919.44	0.99
10	297.66	304.75	1.02
20	151.32	151.64	≈ 1
30	117.38	116.55	0.99
40	96.16	96.49	≈ 1
50	81.75	81.66	≈ 1

Table 7.2: Execution time of totem for a simulation of 36000 events on an Intel Xeon server.

Number of threads	RDataFrame (seconds)	Original (seconds)	Ratio
1	13842.96	14011.58	1.01
16	887.49	869.35	0.98
32	468.13	448.58	0.96
64	253.55	237.56	0.94
96	188.66	184.81	0.98
128	156.21	151.88	0.97
146	144.21	149.09	1.03
164	132.49	146.78	1.11
182	123.58	151.10	1.22
200	120.84	159.61	1.32

Table 7.3: Execution time and speed up obtained with RDataFrame's version of totem for a simulation of 36000 events on a KNL server.

We observe near-identical performance between the execution time of the original code and RDataFrame's implementation on the Xeon server. Figure 7.9 depicts this

results in terms of time (Figure 7.9a) and speed up (Figure 7.9b) with respect to the sequential execution of each version of the code.

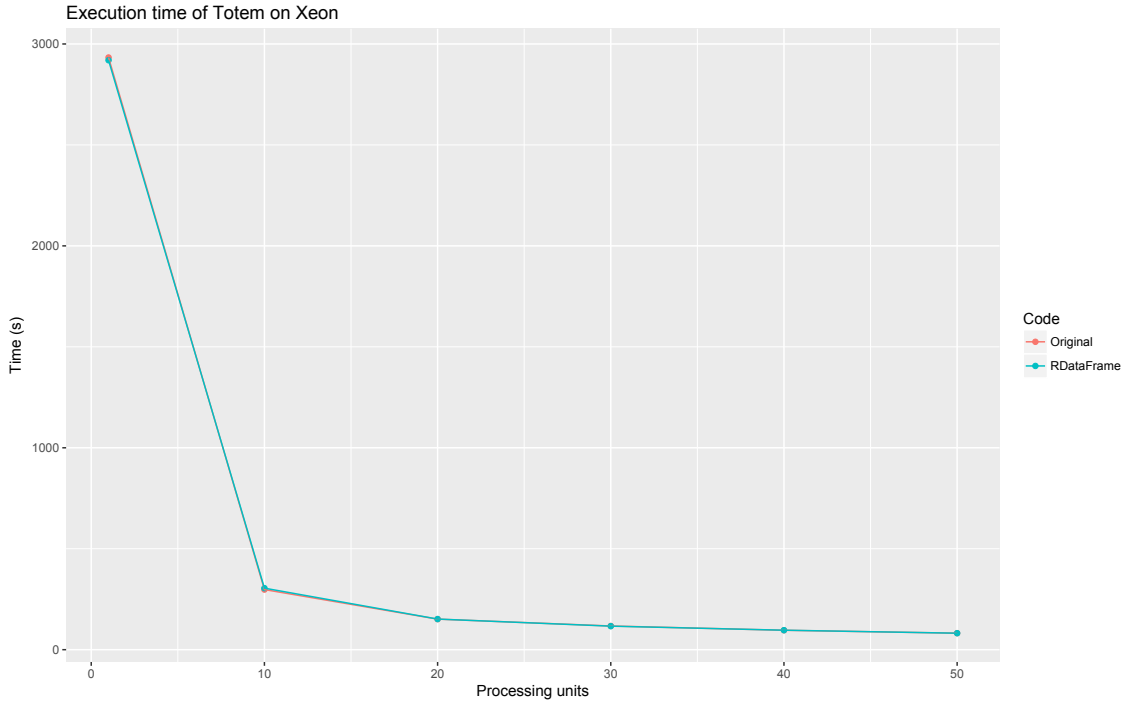
Instead, the KNL benchmark results in terms of time (Figure7.10a) and speed up (Figure 7.10b) exhibit a remarkable difference in scalability in favor of RDataFrame when approaching extreme levels of parallelism. RDataFrame, while offering a more general, convenient and user-friendly way to express the analysis, not only matches the performance of the original ad-hoc code but improves its scalability in many-core architectures.

7.3 Vectorization of mathematical formulas compiled at runtime

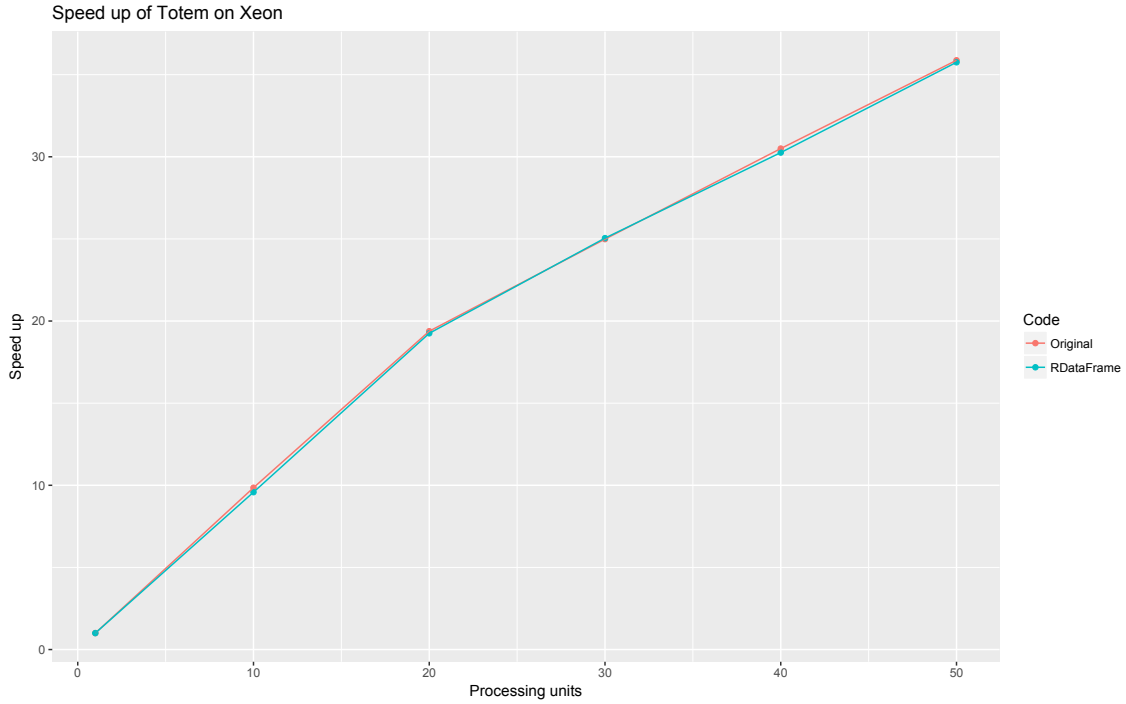
TFormula, the formula class in ROOT, is the perfect candidate to benefit from vectorization. TFormula translates user-defined formulas, expressed as a string, by compiling them into C++ code at runtime, exploiting JIT compilation with Cling. In addition to simple variables and C++ code supported by the standard, TFormula supports ROOT-defined and user-declared functions registered in a list of global functions kept by ROOT, e.g. a function defined with a TF1. Moreover, TFormula provides automatic translation of the most common functions used in HEP analysis into their C++ inline representation, such as `"gauss"` (Gaussian distribution) or `"landau"` (Landau distribution). TFormula is an important component of ROOT, and it is used extensively, from the analysis evaluation in `TTree::Draw()` to the definition of TF1 functions.

Vectorization is implemented in TFormula by compiling the formula variables into ROOT's SIMD types (`ROOT::Double_v`) and relying on VecCore's backend to perform the SIMD operations. TFormula's list of automatic translated functions has been extended to provide automatic translation for functions with a VecCore implementation (e.g., `exp` translates into `vecCore::math::Exp`).

Listing 21 displays a code example for the declaration of a formula. Line 1 creates the function `"Signal"` from a formula and registers it in the global list of ROOT. From this moment, `"Signal"` is available for usage in other formulas. Line 2 performs the same operations for a different formula, `"Background"`. Line 4 showcases the usage of user-declared functions once they have been registered, creating the new function `"Formula V"` from a formula composed of the addition of the previous two declared functions. The only significant difference between this code and that for a scalar formula is that, in Line 6, we declare the function to be vectorizable. A change in the nature of the data the formula accepts is still possible at this stage because the JIT compilation of the formula occurs upon invocation instead of on declaration.

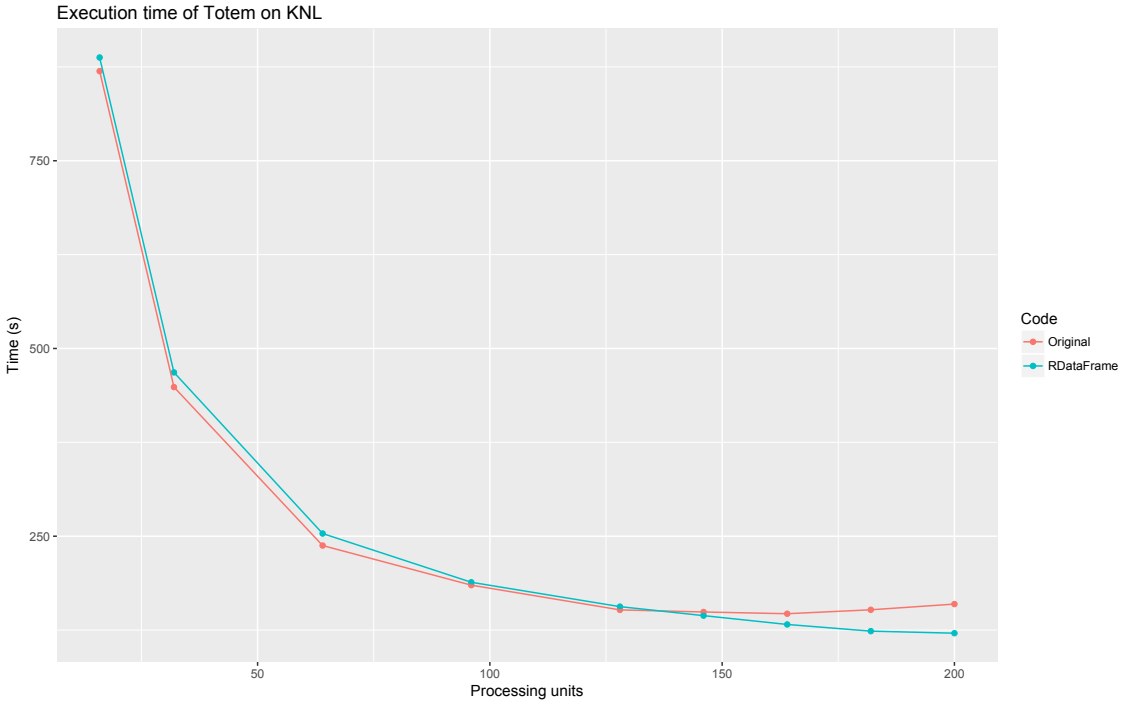


(a) Execution time.

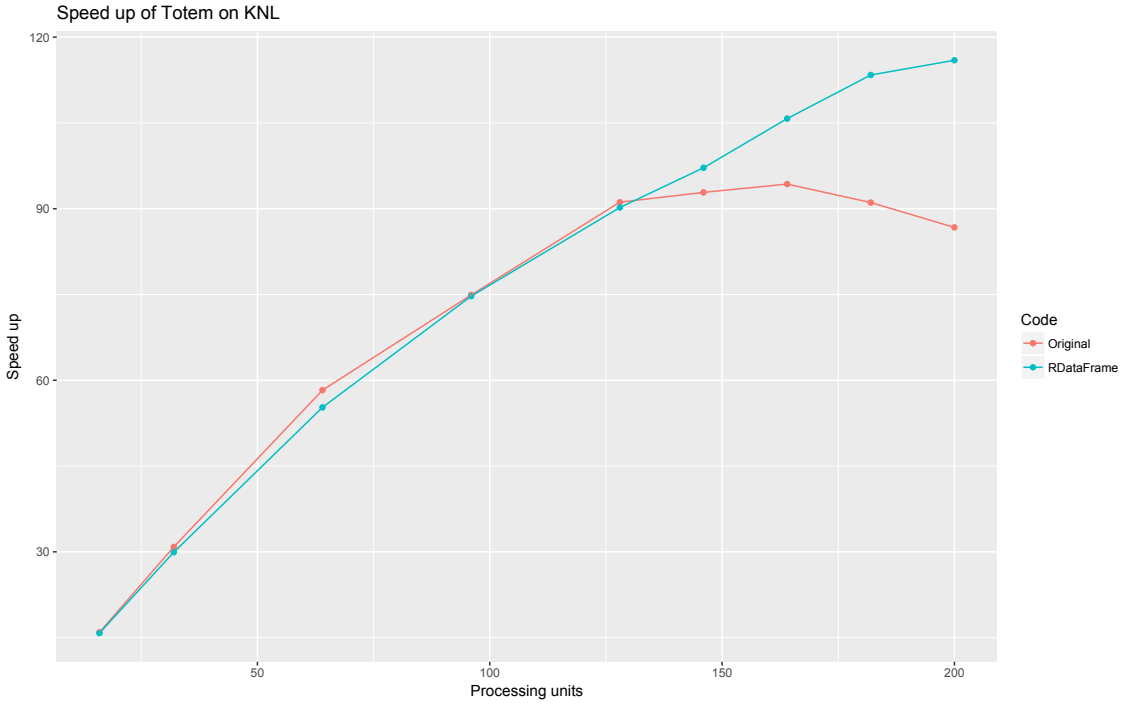


(b) Speed up.

Figure 7.9: Execution time and speed up obtained, for the original code and its RDataFrame implementation when executing Totem with 36000 pairs of protons in the Xeon benchmarking machine.



(a) Execution time.



(b) Speed up.

Figure 7.10: Execution time and speed up obtained, for the original code and its RDataFrame implementation when executing Totem with 36000 pairs of protons in the KNL benchmarking machine.

```
auto fSignal = new TFormula("Signal", "[0] * (-(x + (-130.)) * (x + (-130.)) / 2)");
auto fBackground = new TFormula("Background",
                    "[0] * (-([1] * (x * (0.01)) - [2] * (x * (0.01)) * (x * (0.01))))");

auto fFormula = new TFormula("FormulaV", "fSignal+fBackground");
auto fFormula_v->SetVectorized(true);
```

Listing 21: TFormula declaration for Equation (7.2).

We propose three different cases, or functions, to benchmark the performance of vectorization in TFormula. Case 1 evaluates a function composed of the sum of a signal given by a Gaussian distribution centered on 130 and with variance of 1, and a background given by an exponential function with a polynomic exponent. This is the function declared from a formula in Listing 21.

$$f(x,\theta) = \theta_0 e^{-\frac{(x-130)^2}{2}} + \theta_1 e^{-\left(\theta_2 \frac{x}{100} - \theta_3 \left(\frac{x}{100}\right)^2\right)}; \tag{7.2}$$

Case 2 evaluates a similar function that removes the exponential calls.

$$f(x,\theta) = -\theta_0 \frac{(x-130)^2}{2} + -\theta_1 \left(\theta_2 \frac{x}{100} - \theta_3 \left(\frac{x}{100}\right)^2\right); \tag{7.3}$$

Case 3 evaluates a simple polynomial:

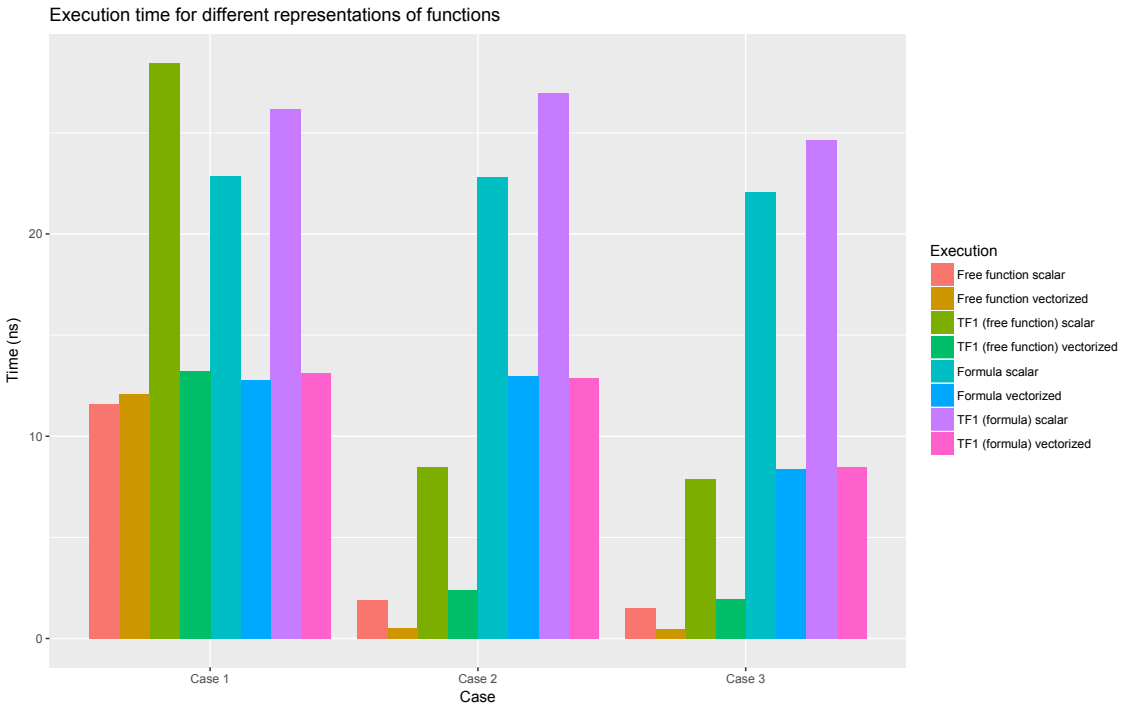
$$f(x,\theta) = \theta_0 x + \theta_1 x^2 + \theta_3 x^3. \tag{7.4}$$

For a more extense comparison, we implement each one of the cases as a free function; as a TF1 wrapping a free function; a TFormula; and a TF1 wrapping a ROOT formula. Figure 7.11 and Table 7.4 present the results in execution time (Figure 7.11a) and speed up (Figure 7.11b), averaged over several executions measured with the C++ standard high resolution clock, for a single execution for each execution policy (scalar or vectorized) of each implementation. We compiled these benchmarks with Apple Clang 9.1.0 and AVX2 as set of instructions in an Intel Core i5-4278U [25] processor with 4 cores at 2.6 GHz and 8 GB of RAM.

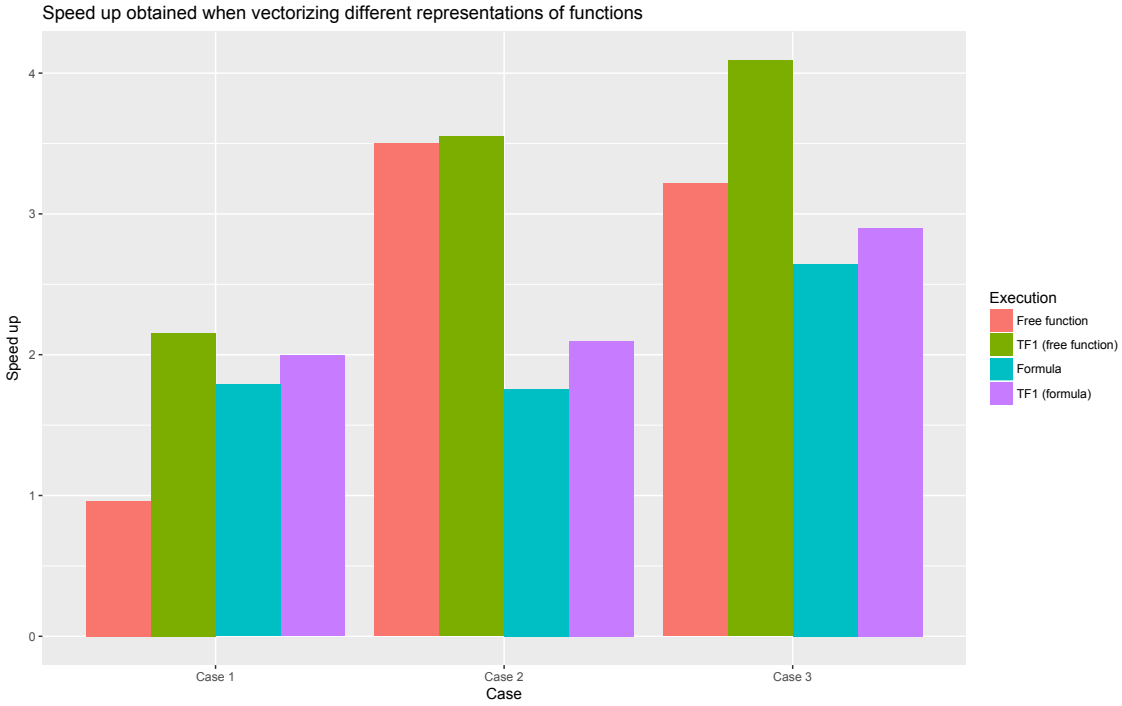
The results exhibit a disparity in results for TFormula when vectorizing. While we expect TFormula to perform worse than a free function due to the overhead and extra levels of indirection introduced by the JIT compilation, the differences in speed up indicate a suboptimal performance of the formula code, reaching in Case2 a divergence of 2× in favour of the free function vectorization with VecCore. Finally, TF1 performs as expected, experiencing little overhead (Section 5.3) when evaluating short functions, due to the introduction of virtual calls and extra levels of indirection. Note how the autovectorization affects the results in Case 1, making the vectorized implementation of the free function slower than the scalar one.

Case	Implementation	Scalar time (ns)	Vectorized time (ns)	Speed up
Case 1	Free function	11.6	12.1	0.96
	TF1 (free function)	28.5	13.2	2.15
	Formula	22.9	12.8	1.79
	TF1 (formula)	26.2	13.1	2.00
Case 2	Free function	1.89	0.541	3.50
	TF1 (free function)	8.48	2.39	3.55
	Formula	22.8	13.0	1.76
	TF1 (formula)	27.0	12.9	2.10
Case 3	Free function	1.50	0.467	3.22
	TF1 (free function)	7.91	1.93	4.09
	Formula	22.1	8.36	2.64
	TF1 (formula)	24.6	8.50	2.90

Table 7.4: Execution time and speed up obtained, normalized with the size of the data, for the scalar and SIMD evaluation of Equation (7.2), Equation (7.3) and Equation (7.4) over one data point, implementing each one as free function, as a TF1 wrapping a free function, as a TFormula and as a TF1 wrapping a ROOT formula.



(a) Execution time.



(b) Speed up.

Figure 7.11: Execution time and speed up obtained for the single event evaluation of equations (7.2), (7.3) and (7.4), when implementing each one, both to evaluate on scalar and SIMD types, as free function, as a TF1 wrapping a free function, as a TFormula and as a TF1 wrapping a ROOT formula. These results are a single-data point normalization of a minimum of 100 and a maximum of 10000 samples (until evaluation time reaches $O(s)$) of measurements taken for the evaluation on 128000 identical data points.

Chapter 8

Conclusions

With this thesis, we demonstrate that it is possible to modernize and parallelize at multiple levels an established, two-decade old code such as ROOT to take advantage of current technologies and architectures. We do this by identifying performance-critical areas and designing and introducing a set of generic utilities (classes, libraries, mechanisms) that drive efficient vectorization and multicore parallelization of these components. Furthermore, we demonstrate it is possible to do so not only while preserving existing user interfaces and functionality, but thanks to ROOT's implicit multithreading mode and SIMD types, also automatically parallelizing user analyses at multiple levels, requiring few changes in user code.

We developed classes that exploit task-parallelism by capitalizing on the parallelization of common workflow patterns such as MapReduce, providing building blocks for multithreading and multiprocessing parallelization and improving code sanity and modularization. We accommodate data-level parallelism with VecCore, a library built on top of other SIMD libraries such as Vc or UME::SIMD, preserving their performance and improving the flexibility, portability and simplicity of their user interfaces.

In our parallel utilities, we reduce remarkably the negative effects of remote memory accesses in NUMA architectures and propose means of improving parallel performance and workload balancing by tuning task granularity.

The usage of our task-level and data-level parallelism utilities has propagated rapidly in ROOT, triggering a remarkable improvement in performance for users' analyses. We showcase some of these improvements in performance, such as the vectorization of ROOT's function and formula classes. We prove that the fitting, a common and computationally intensive operation in HEP analysis, is an ideal process which can be parallelized at multiple levels, describing in detail the modifications required for a successful parallelization and vectorization of the fitting classes and interfaces. We demonstrate how, thanks to multiprocessing, we can dramatically improve ROOT's I/O performance when merging typical ROOT files. Finally,

we prove the performance and scalability of our utilities in RDataFrame, matching the performance of a highly-optimized custom code, and scaling up to more than 200 threads.

8.1 Contributions

The scope of this work covers many aspects of ROOT's modernization effort. In particular, we contribute to this task with:

- An updated description of ROOT, incorporating the new developments in the area of parallelism.
- A set of parallel tools (TProcessExecutor, TThreadExecutor, TPoolManager) to leverage parallelism at task-level.
- The integration of VecCore to implement data-level parallelism
- The extensive deployment of these new utilities in ROOT's codebase.

These contributions to improving ROOT's performance and programming model can be classified into two different lines of work: task-level and data-level parallelism.

Task-level parallelism

We contributed to the task-level parallelization of ROOT by identifying common analysis patterns and providing classes implementing them in parallel. We deployed these classes extensively throughout ROOT, always preserving backwards compatibility, improving the programming model, and abstracting the complexity of concurrent programming from the user interfaces.

In the scope of this thesis:

1. We introduce the executors, classes implementing the MapReduce pattern in parallel; implement a TBB-based multithreading executor, TThreadExecutor; and adapt the fork-based multiprocessing executor of ROOT, TProcessExecutor.
2. We prove the near-optimal performance of the executors and state how differences in task overhead for multiprocessing and multithreading executions can affect the optimal number of tasks in each case. In addition, we demonstrate how task granularity affects performance, and provide the executors' MapReduce call of an extra parameter to tune the number of tasks.

3. We design a centralized thread pool manager to safeguard interactions between ROOT's implicit and explicit multithreading execution modes and guarantee correct interoperability with TBB-based codes that interact with ROOT, such as experiment frameworks.

4. We propose a solution for NUMA effects when parallelizing with the executors on NUMA architectures. With TNUMAExecutor, we reduce by a 30% the number of lookups to remote memory and obtain a 1.8× speed up in a server composed of two NUMA domains when performing a parallel maximum likelihood fit.

5. We parallelize some of the most critical operations in ROOT by capitalizing on the executors:

    5.1. Adapt ROOT fitting's objective functions to leverage the executors' parallel MapReduce pattern, obtaining close to ideal parallel performance.

    5.2. Prove the performance, convenience and scalability in extremely parallel architectures (over 200 threads) of RDataFrame, which parallelizes the event loop with TThreadExecutor.

    5.3. Analyze and implement the two-step multiprocessing parallel merging of ROOT files, a fundamental operation in the LHC analysis chain, obtaining superlinear speed ups with respect to the original version.

Data-level parallelism

We successfully introduce data-level parallelism in ROOT by exploiting VecCore, a library that offers an extra layer of abstraction over the SIMD libraries it uses at backends, simplifying and unifying their programming model and, therefore, allowing an appropriate selection for the problem and platform the code is being executed on. Portability is another advantage provided by VecCore: the extra layer of abstraction allows for the development of platform-agnostic code, selecting the appropriate SIMD instruction set at compilation.

In this context:

1. We analyze VecCore's programming model convenience and backends' performance by comparing a code for the generation Julia sets with its VecCore implementation.

    1.1. We state that Vc is VecCore's most performant backend for every data type, compiler and SIMD instruction set it supports.

    1.2. We show that gcc 7 outperforms icc 18 and clang as the compiler for both VecCore backends tested.

   1.3. We demonstrate that, even if VecCore's UME::SIMD backend AVX512 support provides us with the most remarkable improvements in speed ups, the inferior clock speed of the server (Intel's Xeon Phi) cancels any efficiency gains against the same code executing on a commodity PC

2. We introduce two new SIMD types in ROOT, `ROOT::Double_v` and `ROOT::Float_v` for vectorized operations.

3. We provide the function class in ROOT with SIMD evaluation without changing its programming model or functionality.

4. We prove that we can generate VecCore code at runtime, transparently for the user, when JIT compiling formulas specified as strings.

5. We adapt ROOT fitting classes and interfaces to support the evaluation of the fitting with model functions evaluating on SIMD data types. We change the structure of the stored fitting data and provided padding mechanisms for safety.

8.2 Open lines of research

Future potential areas of work involve to continue deploying parallelism throughout ROOT most performance-critical areas, and continue building on the successful executors model with two new implementations:

- **GPUExecutor**: Executor for capitalizing on the massive parallelism offered by GPUs, useful for areas such as the fitting and the Machine Learning library of ROOT, TMVA.
- **DistributedExecutor**: Executor for the distributed computing of ROOT analysis, potential evolution of PROOF.

In addition, support for data-level parallelism and VecCore is being extended in ROOT's mathematical libraries, with the vectorization of TMath and the predefined mathematical functions in ROOT.

8.3 Related publications

The major part of the studies exposed in this thesis have been published as:

1. D. Piparo, E. Tejedor, E. Guiraud, G. Ganis, P. Mato, L. Moneta, X. Valls Pla, P. Canal. "Expressing Parallelism with ROOT". *22nd International Conference on Computing in High Energy and Nuclear Physics.*

2. X.Valls Pla, L. Moneta. "Parallelization and vectorization of ROOT fitting classes". *18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research.*

3. G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P. Mato Vila, L. Moneta, D. Piparo, E. Tejedor, X. Valls Pla. "Novel functional and distributed approaches to data analysis available in ROOT". *18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*

4. G. Amadio, P. Canal, G. Ganis, E. Guiraud, A. Naumann, D. Piparo, E. Tejedor, X. Valls Pla. "RDataFrame: Easy Parallel ROOT Analysis at 100 Threads". *23rd International Conference on Computing in High Energy and Nuclear Physics.*

5. G. Amadio, L. Moneta, X. Valls Pla. "Vectorization of ROOT Mathematical Libraries". *23rd International Conference on Computing in High Energy and Nuclear Physics.*

6. G. Amadio, P. Canal, G. Ganis, E. Guiraud, A. Naumann, D. Piparo, E. Tejedor, X. Valls Pla. "Supporting Future HEP Data Processing with a Parallelised ROOT". *23rd International Conference on Computing in High Energy and Nuclear Physics.*

7. L. Moneta, A. Tsang, X. Valls Pla. "Fitting and Modeling in ROOT". *23rd International Conference on Computing in High Energy and Nuclear Physics.*

Capítol 9

Conclusions

Amb aquesta tesi, demostrem que és possible modernitzar i paral·lelitzar a múltiples nivells un codi establert de dues dècades d'antiguitat com ROOT per aprofitar les tecnologies i arquitectures actuals. Per a això, identifiquem àrees crítiques per al rendiment i dissenyem i introduïm un conjunt d'utilitats genèriques (classes, llibreries, mecanismes) que impulsen una vectorització eficient i una paral·lelització multicore d'aquests components. A més, demostrem que és possible fer-ho no només preservant les interfícies d'usuari i la funcionalitat existents, sinó també gràcies al mode multifil implícit de ROOT i als tipus de dades SIMD, que paral·lelitzen automàticament les anàlisis en múltiples nivells, requerint pocs canvis en el codi dels usuaris.

Desenvolupem classes que exploten el paral·lelisme de tasques aprofitant la paral·lelització de patrons comuns en el codi, com MapReduce, proporcionant blocs de construcció per a la paral·lelització multifil i multiprocés i millorant la solidesa del codi i la seva modularització. Adaptem el paral·lelisme a nivell de dades amb VecCore, una llibreria construïda sobre altres llibreries SIMD com Vc o UME::SIMD, preservant el seu rendiment i millorant la flexibilitat, portabilitat i simplicitat de les seves interfícies d'usuari.

A les nostres utilitats paral·leles, reduïm notablement els efectes negatius dels accessos remots a la memòria en arquitectures NUMA i proposem mitjans per millorar el rendiment paral·lel i l'equilibri de la càrrega de treball mitjançant l'ajust de la granularitat de les tasques.

L'ús de les nostres utilitats de paral·lelisme a nivell de tasca i a nivell de dades s'ha propagat ràpidament a ROOT, provocant una notable millora en el rendiment de les anàlisis dels usuaris. Mostrem algunes d'aquestes millores en el rendiment, com la vectorització de les classes que representen funcions i fórmules en ROOT. Demostrem que l'ajust, una operació comú i computacionalment intensiva en l'anàlisi HEP, és un procés ideal per ser

paral·lelitzat en múltiples nivells, i descrivim en detall les modificacions requerides per a una paral·lelització i vectorització reeixida de les classes i interfícies relacionades amb l'ajust. Demostrem com, gràcies al multiprocessament, podem millorar dràsticament el rendiment d'entrada/sortida de ROOT en combinar arxius ROOT estàndards. Finalment, comprovem el rendiment i escalabilitat de les nostres utilitats en RDataFrame, igualant el rendiment d'un codi personalitzat altament optimitzat, i demostrant la seva escalabilitat fins més de 200 fils.

9.1 Contribucions

L'àmbit d'aquest treball cobreix molts aspectes de l'esforç de modernització de ROOT. En particular, contribuïm a aquesta tasca amb:

- Una descripció actualitzada de ROOT, incorporant els nous desenvolupaments en l'àrea del paral·lelisme.
- Un conjunt d'eines paral·leles (TProcessExecutor, TThreadExecutor, TPoolManager) per aprofitar el paral·lelisme a nivell de tasca.
- La integració de VecCore per implementar el paral·lelisme a nivell de dades
- El desplegament extensiu d'aquestes noves utilitats en el codi base de ROOT.

Aquestes contribucions a la millora del rendiment i del model de programació de ROOT poden classificar-se en dues línies de treball diferents: paral·lelisme a nivell de tasca i paral·lelisme a nivell de dades.

Task-level parallelism

Contribuïm a la paral·lelització a nivell de tasca de ROOT identificant patrons d'anàlisi comuns i proporcionant classes que els implementen en paral·lel. Despleguem aquestes classes extensivament a ROOT, sempre preservant la compatibilitat amb el codi anterior, millorant el model de programació, i abstraient la complexitat de la programació concurrent de les interfícies d'usuari.

En l'àmbit d'aquesta tesi:

1. Introduïm els executors, classes que implementen el patró MapReduce en paral·lel;

2. Implementem un executor multifil basat en TBB, TThreadExecutor; i adaptem l'executor multiprocés basat en forking de ROOT, TProcessExecutor.

3. Comprovem el rendiment quasi-òptim dels executors i demostrem com les diferències en la sobrecàrrega de tasques per execucions multiprocés i multifil poden afectar el nombre òptim de tasques en cada cas. A més, demostrem com la granularitat de les tasques afecta el rendiment i proporcionem un paràmetre addicional per ajustar el nombre de tasques a trucada MapReduce dels executors.

4. Dissenyem un gestor centralitzat del conjunt de fils per salvaguardar les interaccions entre els modes d'execució multifil implícit i explícit de ROOT i garantir la correcta interoperabilitat amb altres codis basats en TBB que interactuen amb ROOT, com els entorns de programari dels experiments.

5. Proposem una solució per als efectes NUMA quan es paral·lelitza amb els executors en arquitectures NUMA. Amb TNUMAExecutor, reduïnt en un 30% el nombre d'accessos a memòria remota i obtenint un speed up de 1.8× en un servidor compost per dos dominis NUMA en realitzar un ajust de màxima versemblança en paral·lel.

6. Paral·lelitzem algunes de les operacions més crítiques en ROOT beneficiantnos dels executors:

   a) Adaptar les funcions objectiu de l'ajust en ROOT per aprofitar el patró MapReduce paral·lel implementat en els executors, obtenint un rendiment paral·lel gairebé ideal.

   b) Provar el rendiment, conveniència i escalabilitat en arquitectures extremadament paral·leles (més de 200 fils) de RDataFrame, que paral·lelitza el bucle d'esdeveniments amb TThreadExecutor.

   c) Analitzar i implementar en dos passos paral·lels (multiprocés) la fusió d'arxius ROOT, operació fonamental en la cadena d'anàlisi de l'LHC, obtenint acceleracions superlinears respecte a la versió original.

Parallelisme a nivell de dades

Introduïm amb èxit el paral·lelisme a nivell de dades en ROOT mitjançant l'explotació d'VecCore, una llibreria que ofereix una capa extra d'abstracció sobre les llibreries SIMD que utilitza com a backends, simplificant i

unificant el seu model de programació i, per tant, permetent seleccionar el backend adequat per al problema i la plataforma en la qual s'està executant el codi. La portabilitat és un altre dels avantatges de VecCore: la seva capa extra d'abstracció permet el desenvolupament de codi independent de la plataforma d'execució, seleccionant durant la seva compilació el conjunt d'instruccions SIMD apropiat per a aquesta plataforma.

En aquest context:

1. Analitzem la conveniència del model de programació de VecCore i el rendiment dels seus backends comparant un codi per a la generació de conjunts de Julia amb la seva implementació amb VecCore.

    a) Afirmem que Vc és el backend de VecCore més eficient per a cada un dels tipus de dades, compiladors i conjunts d'instruccions SIMD suportats.

    b) Mostrem que gcc 7 avantatja icc 18 i clang com a compilador per als dos backends de VecCore comprovats.

    c) Demostrem que, inclús si el suport per AVX512 del backend de VecCore UME::SIMD ens proporciona les millores més notables en la velocitat, la inferior freqüència del processador del servidor (Intel Xeon Phi) cancel·la qualsevol guany en eficiència respecte al mateix codi executat en un PC convencional.

2. Introduïm dos nous tipus de dades SIMD en ROOT, ROOT::Double_v i ROOT::Float_v per a operacions vectoritzades.

3. Proporcionem avaluació SIMD en la classe funció de ROOT sense canviar el seu model de programació o funcionalitat.

4. Provem que podem generar codi VecCore en temps d'execució, de forma transparent per a l'usuari, compilant fórmules definides com cadenes de text.

5. Adaptem les classes i interfícies d'ajust ROOT per suportar l'avaluació vectorizada de l'ajust, donades funcions model que s'avaluen sobre tipus de dades SIMD. Canviem l'estructura de les dades emmagatzemades per a l'ajust i proporcionem mecanismes per completar els vectors SIMD.

9.2 Línies d'investigació obertes

Les possibles futures àrees de treball impliquen continuar desplegant el paral·lelisme en les àrees més crítiques per al rendiment de ROOT, i continuar construint sobre l'exitós model dels executors amb dues noves implementacions:

- **GPUExecutor**: Executor per aprofitar el paral·lelisme massiu que ofereixen les GPUs, útil per a àrees com l'ajust de distribucions de dades i la biblioteca de Machine Learning de ROOT, TMVA.
- **DistributedExecutor**: Executor per a la computació distribuïda de l'anàlisi amb ROOT, potencial evolució de PROOF.

A més, el suport per al paral·lelisme a nivell de dades i per VecCore s'està ampliant a les llibreries matemàtiques de ROOT, amb la vectorització de TMath i les funcions matemàtiques predefinides a ROOT.

9.3 Publicacions relacionades

La major part dels estudis exposats dins aquesta tesi han estat publicats com:

1. D. Piparo, E. Tejedor, E. Guiraud, G. Ganis, P. Mato, L. Moneta, X. Valls Pla, P. Canal. "Expressing Parallelism with ROOT". *22nd International Conference on Computing in High Energy and Nuclear Physics.*
2. X.Valls Pla, L. Moneta. "Parallelization and vectorization of ROOT fitting classes". *18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research.*
3. G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P. Mato Vila, L. Moneta, D. Piparo, E. Tejedor, X. Valls Pla."Novel functional and distributed approaches to data analysis available in ROOT". *18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*
4. G. Amadio, P. Canal, G. Ganis, E. Guiraud, A. Naumann, D. Piparo, E. Tejedor, X. Valls Pla. "RDataFrame: Easy Parallel ROOT Analysis at 100 Threads". *23rd International Conference on Computing in High Energy and Nuclear Physics.*

5. G. Amadio, L. Moneta, X. Valls Pla. "Vectorization of ROOT Mathematical Libraries". *23rd International Conference on Computing in High Energy and Nuclear Physics*.

6. G. Amadio, P. Canal, G. Ganis, E. Guiraud, A. Naumann, D. Piparo, E. Tejedor, X. Valls Pla. "Supporting Future HEP Data Processing with a Parallelised ROOT". *23rd International Conference on Computing in High Energy and Nuclear Physics*.

7. L. Moneta, A. Tsang, X. Valls Pla. "Fitting and Modeling in ROOT". *23rd International Conference on Computing in High Energy and Nuclear Physics*.

Glossary

ALICE A Large Ion Collider Experiment.

ATLAS A Toroidal LHC Apparatus.

BDT Boosted Decision Trees.

CERN European Organization for Nuclear Research.

CMS Compact Muon Solenoid.

CWS Column-wise storage.

DNN Deep Neural Networks.

HEP High Energy Physics.

HL-LHC High Luminosity LHC.

I/O Input/Output.

IMT Implicit multithreading.

JIT Just-In-Time.

KNL Intel Knights Landing.

LHC Large Hadron Collider.

LHCb Large Hadron Collider beauty.

LINAC Linear accelerator.

P.d.f. Probability density function.

PROOF The Parallel ROOT Facility.

PS Proton Synchroton.

RAD Rapid Application Development.

REPL Read-eval-print-loop.

SIMD Single Instruction Multiple Data.

SPS Super Proton Synchroton.

SWAN Service for Web based ANalisys.

WLCG Worldwide LHC Computing Grid.

Bibliography

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Sea Agostinelli, John Allison, K al Amako, Jo Apostolakis, H Araujo, P Arce, M Asai, D Axen, S Banerjee, G 2 Barrand, et al. Geant4a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003.

[3] C Aguado-Sanchez, J Blomer, P Buncic, I Charalampidis, G Ganis, M Nabozny, and F Rademakers. Studying root i/o performance with proof-lite. In *Journal of Physics: Conference Series*, volume 331, page 032010. IOP Publishing, 2011.

[4] Guilherme Amadio, Philippe Canal, and Sandro Wenzel. root-project/veccore: Release v0.4.2, August 2017.

[5] Andi Kleen, SUSE Labs. numatcl. https://github.com/numactl/numactl, 2018. [Online; accessed 2-September-2018].

[6] Apache Software Foundation. Apache arrow. https://arrow.apache.org/, 2018. [Online; accessed 2-September-2018].

[7] Steve Baker and Robert D Cousins. Clarification of the use of chi-square and likelihood functions in fits to histograms. *Nuclear Instruments and Methods in Physics Research*, 221(2):437–442, 1984.

[8] B. Bellenot, R. Brun, G. Ganis, J. Iwaszkiewicz, G. Kickinger, A. J. Peters, F. Rademakers, M. Ballintijn, C. Loizides, C. Reed, and D. Feichtinger. Proof - the parallel root facility. In *2006 15th IEEE International Conference on High Performance Distributed Computing*, pages 379–380, 2006.

[9] Jakob Blomer. A quantitative review of data formats for hep analyses. In *18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*. IOP, forthcoming.

[10] Brian Bockelman, Oksana Shadura. Zstd & lz4. https://indico.fnal.gov/event/16264/contribution/8/material/slides/0.pdf, 2018. [Online; accessed 2-September-2018].

[11] Rene Brun and Fons Rademakers. Root - an object oriented data analysis framework. In Nucl. Inst. and Meth. in Phys., editors, *Proceedings AIHENP'96 Workshop*, volume A389, pages 81–86, January 1997.

[12] Philippe Canal, Gerardo Ganis, Enrico Guiraud, Pere Mato Vila, Lorenzo Moneta, Danilo Piparo, Enric Tejedor, and Xavier Valls Pla. Expressing parallelism in root. In *22nd International Conference on Computing in High Energy and Nuclear Physics*. IOP, in press.

[13] CERN. Computer generated image of the whole atlas detector. https://cds.cern.ch/images/CERN-GE-0803012-01, 2008. [Online; accessed 2-September-2018].

[14] CERN. The cern accelerator complex. https://cds.cern.ch/record/2197559, 2016. [Online; accessed 2-September-2018].

[15] CERN. Cms detector slice. https://cds.cern.ch/record/2120661, 2016. [Online; accessed 2-September-2018].

[16] CMS collaboration et al. Measurements of properties of the higgs boson decaying into the four-lepton final state in pp collisions at $\sqrt{=}13tev$. *arXiv preprint arXiv:1706.09936*, 2017.

[17] CMS Collaboration et al. Measurements of higgs boson properties in the diphoton decay channel in proton-proton collisions at $\sqrt{s} = 13tev$. *arXiv preprint arXiv:1804.02716*, 2018.

[18] Gennaro Corcella, Ian G Knowles, Giuseppe Marchesini, Stefano Moretti, Kosuke Odagiri, Peter Richardson, Michael H Seymour, and Bryan R Webber. Herwig 6: an event generator for hadron emission reactions with interfering gluons (including supersymmetric processes). *Journal of High Energy Physics*, 2001(01):010, 2001.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Googlel. Benchmark. https://github.com/google/benchmark, 2018. [Online; accessed 2-September-2018].

[21] Enrico Guiraud and Gerardo Ganis. Enhancements to multiprocessing in root. Sep 2015.

[22] Enrico Guiraud and Gerardo Ganis. Enhancements to multiprocessing in root, September 2015.

[23] Enrico Guiraud, Axel Naumann, and Danilo Piparo. Tdataframe: functional chains for root data analyses. Jan 2017.

[24] Andreas Hoecker, Peter Speckmayer, Joerg Stelzer, Jan Therhaag, Eckhard von Toerne, Helge Voss, M Backes, T Carli, O Cohen, A Christov, et al. Tmva-toolkit for multivariate data analysis. *arXiv preprint physics/0703039*, 2007.

[25] Intel. Intel core i5-4278u processor. https://ark.intel.com/products/83508/Intel-Core-i5-4278U-Processor-3M-Cache-up-to-3_10-GHz, 2018. [Online; accessed 2-September-2018].

[26] Intel. Intel core i7-4790 processor. https://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz, 2018. [Online; accessed 2-September-2018].

[27] Intel. Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2018. [Online; accessed 2-September-2018].

[28] Intel. Intel threading building blocks documentation - scheduling algorithm. https://software.intel.com/en-us/node/506295, 2018. [Online; accessed 2-September-2018].

[29] Intel. Intel threading building blocks documentation - task scheduler. https://software.intel.com/en-us/node/506294, 2018. [Online; accessed 2-September-2018].

[30] Intel. Intel threading building blocks documentation - task_arena class. https://software.intel.com/en-us/node/506359, 2018. [Online; accessed 2-September-2018].

[31] Intel. Intel xeon phi processor 7210. https://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core, 2018. [Online; accessed 2-September-2018].

[32] Intel. Intel xeon processor e5-2683 v3. https://ark.intel.com/products/81055/Intel-Xeon-Processor-E5-2683-v3-35M-Cache-2_00-GHz, 2018. [Online; accessed 2-September-2018].

[33] Jim Pivarski. Parquet data format performance. https://indico.fnal.gov/event/16264/contribution/1/material/slides/0.pdf, 2018. [Online; accessed 2-September-2018].

[34] P Karpinski and J McDonald. A high-performance portable abstract interface for explicit simd vectorization. In *PMAM@ PPoPP*, pages 21–28, 2017.

[35] Matthias Kretz and Volker Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012.

[36] Michelangelo L Mangano, Fulvio Piccinini, Antonio D Polosa, Mauro Moretti, and Roberto Pittau. Alpgen, a generator for hard multiparton processes in hadronic collisions. *Journal of High Energy Physics*, 2003(07):001, 2003.

[37] Marc Bohr. A 30 year retrospective on dennard's mosfet scaling paper. http://www.eng.auburn.edu/~agrawvd/COURSE/READING/LOWP/Boh07.pdf, 2007. [Online; accessed 2-September-2018].

[38] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[39] Lorenzo Moneta, I Antcheva, and R Brun. Recent developments of the root mathematical and statistical software. In *Journal of Physics: Conference Series*, volume 119, page 042023. IOP Publishing, 2008.

[40] Lorenzo Moneta, I Antcheva, and D Gonzalez Maline. Recent improvements of the root fitting and minimization classes. *PoS*, page 075, 2008.

[41] Lorenzo Moneta, Kevin Belasco, Kyle Cranmer, Sven Kreiss, Alfio Lazzaro, Danilo Piparo, Gregory Schott, Wouter Verkerke, and Matthias Wolf. The roostats project. *arXiv preprint arXiv:1009.1003*, 2010.

[42] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[43] Danilo Piparo, Enric Tejedor, Pere Mato, Luca Mascetti, Jakub Moscicki, and Massimo Lamanna. Swan: A service for interactive analysis in the cloud. *Future Generation Computer Systems*, 78:1071–1078, 2018.

[44] ROOT Project. The root object i/o system. https://root.cern.ch/root/InputOutput.html, 1996. [Online; accessed 2-September-2018].

[45] ROOT Project. Gallery. https://root.cern/gallery, 2018. [Online; accessed 2-September-2018].

[46] ROOT Project. Math libraries. https://root.cern.ch/root/html534/guides/users-guide/MathLibraries.html, 2018. [Online; accessed 2-September-2018].

[47] ROOT Project. User's guide. https://root.cern.ch/guides/users-guide, 2018. [Online; accessed 2-September-2018].

[48] Alberto Rotondi and Paolo Montagna. Fast calculation of vavilov distribution. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 47(3):215–223, 1990.

[49] H Sakamoto, D Bonacorsi, I Ueda, and A Lyon. 21st international conference on computing in high energy and nuclear physics (chep2015). *Journal of Physics: Conference Series*, 664(00):001001, 2015.

[50] B Schorr. Programs for the landau and the vavilov distributions and the corresponding random numbers. *Comput. Phys. Commun.*, 7(CERN-DD-73-26):215–24, 1973.

[51] Jamie Shiers. The worldwide lhc computing grid (worldwide lcg). *Computer physics communications*, 177(1-2):219–223, 2007.

[52] Frank Siegert. *Monte-Carlo event generation for the LHC*. PhD thesis, Durham University, 2010.

[53] Matevz Tadel et al. Alieve-alice event visualization environment. 2006.

[54] Oliphant Travis E. A guide to numpy. http://www.numpy.org/, 2006. [Online; accessed 2-September-2018].

[55] V Vasilev, Ph Canal, A Naumann, and P Russo. Cling–the new interactive interpreter for root 6. In *Journal of Physics: Conference Series*, volume 396, page 052071. IOP Publishing, 2012.

[56] Wouter Verkerke and David Kirkby. The roofit toolkit for data modeling. In *Statistical Problems in Particle Physics, Astrophysics and Cosmology*, pages 186–189. World Scientific, 2006.

[57] Wikipedia contributors. Uncore. https://en.wikipedia.org/wiki/Uncore, 2017. [Online; accessed 2-September-2018].

[58] Wikipedia contributors. Dennard scaling. https://en.wikipedia.org/wiki/Dennard_scaling, 2018. [Online; accessed 2-September-2018].

[59] Wikipedia contributors. Dimm. https://en.wikipedia.org/wiki/DIMM, 2018. [Online; accessed 2-September-2018].

[60] Wikipedia contributors. Julia set. https://en.wikipedia.org/wiki/Julia_set, 2018. [Online; accessed 2-September-2018].

[61] Wikipedia contributors. Kalman filter. https://en.wikipedia.org/wiki/Kalman_filter, 2018. [Online; accessed 2-September-2018].

[62] Wikipedia contributors. Rapid application development. https://en.wikipedia.org/wiki/Rapid_application_development, 2018. [Online; accessed 2-September-2018].

[63] Wikipedia contributors. Read-eval-print loop. https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop, 2018. [Online; accessed 2-September-2018].