Universidade do Minho
Escola de Engenharia

André Martins Pereira

HEP-Frame: a Development Aid and Efficient Execution Engine where a Multi-layer Scheduler Adaptively Orders Pipelined Data Stream Applications

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho
universidade de aveiro
U.PORTO

fevereiro de 2019

REPÚBLICA PORTUGUESA
POCH PROGRAMA OPERACIONAL CAPITAL HUMANO
UNIÃO EUROPEIA
Fundo Social Europeu
FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR
COMPETE
PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE
COMPETE 2020
PORTUGAL 2020
UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional

Universidade do Minho
Escola de Engenharia

André Martins Pereira

HEP-Frame: a Development Aid and Efficient Execution Engine where a Multi-layer Scheduler Adaptively Orders Pipelined Data Stream Applications

Programa de Doutoramento em Informática (MAP-i) das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

universidade de aveiro

U.PORTO

Trabalho realizado sob a orientação do
Professor Doutor Alberto José Gonçalves Carvalho Proença
e do
Professor Doutor António Joaquim Onofre de Abreu Ribeiro

fevereiro de 2019

Acknowledgements

First and foremost, I would like to give my deepest gratitude to my supervisors, professors Alberto Proença and António Onofre, for the excellent support, dedication, and more importantly, patience throughout the last seven years we have been working together. To professor Proença, I would like to express my most sincere appreciation for helping me grow as a professional, but above all, as a person, for showing me that I have to put myself and my family first, for guiding me through tough times, and for always fighting to give me the best opportunities I could possibly have. To professor Onofre, I would like to thank for always believing in me, showing the impact that I had on the work of real people even when I doubted myself, and for always being that motivating voice in these last years. I also have to mention that without him I would not try rock climbing, which allowed me to meet the most amazing people that completely changed my life. I hope that this was just the beginning of the collaborations with both of them on a long road ahead.

I would like to give a heartfelt thank to my family, especially to my parents, my brother, and my girlfriend for the endless support, confidence, and happiness throughout the challenges that I had to overcome throughout these years. Without them I could not be where I am today, as they always believed and showed me that I had what it took.

And last, but not least, a sincere thank to all my friends, from inside and outside of the university, who helped me distract from work in many times of stress, while creating the most memorable moments of my life so far. Without them these last years would be so much harder to bear.

Finally, this work is partially funded by FCT – Foundation for Science and Technology, the

Portuguese Ministry of Science, Technology and Higher Education, through national funds, and co-financed by the European Social Fund (ESF) through the Operacional Programme for Human Capital (POCH), with scholarship reference SFRH/BD/119398/2016. This work was developed with support of the computing facilities provided by the project "Search-ON2: Revitalization of HPC infrastructure of UMinho” (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). Additional funding provided under the UTAustin Program, financed by FCT, COMPETE POCI-01-0145-FEDER-007043 within the Project Scope UID/CEC/00319/2013 and by the Laboratório de Instrumentação e Física Experimental de Partículas.

Abstract

The key component of this thesis work is HEP-Frame, a framework to aid the development and efficient execution of pipelined data stream applications in homogeneous and heterogeneous servers. A pipelined data stream application is a process that converts large amounts of experimental raw data into useful information to monitor data, test hypotheses or validate theories. Each dataset element is then processed by a pipeline of propositions, each containing a computational task that may be followed by an evaluation of a criterion; if this fails the dataset element is removed from the pipeline.

Optimising the computational performance of these applications requires expertise to efficiently vectorize and parallelise the code – namely to take advantage of the vector extensions at each processor (core) and in multiple processors at each server with accelerator devices, in a multi-server cluster – which most scientists lack, or an adequate and easy to use tool or framework, which is nonexistent. With this motivation, the HEP-Frame was designed, implemented and evaluated to provide an user-centred development interface, through the use of code skeletons, automatic code generation, and automation of the compilation process, while transparently scheduling and managing the efficient parallel execution of the code on multicore and manycore servers, with and without accelerator devices.

HEP-Frame implements a multi-layer scheduler that adapts at run-time the application to the computational server(s) and processes the pipeline propositions and various dataset elements in parallel, distributing them across the available computing resources. The top layer balances data and workloads among servers in a heterogeneous cluster environment, using a demand-driven approach to allow HEP-Frame to scale with multiple servers. The middle layer dynamically tunes the number of threads assigned to the parallel data read and setup of adequate data structures, and the pipeline execution.

The bottom layer manages the parallel execution of the dataset workload among the available computing resources in a server; this layer includes the reordering of the pipeline propositions of the same dataset element and the parallel execution of multiple dataset elements, ensuring that faster propositions that filter out more data are executed before the more compute intensive propositions.

HEP-Frame also provides a wide range of efficient dual-buffer pseudo-random number generators with uniform or Gaussian distributions, often required by these applications. These can be executed on the compute server, or offloaded to other multicore/manycore servers or to manycore/GPU accelerators. The dual-buffer strategy hides the time penalties to transfer data from other servers or accelerator devices.

The quantitative evaluation of HEP-Frame used three versions of a real world application, the $t\bar{t}H$ *particle physics event data analysis, developed and used by CERN researchers:* `ttH_as`, `ttH_sci` *and* `ttH_scinp`, *each with a pipeline with 18 propositions in a default order defined by the developers. The former analysis is latency-bound while the latter two are compute-bound. The* $t\bar{t}H$ *analyses are originally sequential, but multithreaded map-reduce parallelisations with OpenMP and StarPU were implemented for thread-by-thread comparison with HEP-Frame. Five heterogeneous servers were selected to perform a quantitative evaluation of the HEP-Frame performance: three dual-socket servers with 12-, 16-, and 24-core Xeon devices (Ivy Bridge, Broadwell, and Skylake, respectively), with the former server being coupled with an NVidia Tesla K20 Kepler GPU and two Intel Xeon Phi Knights Corner devices; a single-socket 10-core Broadwell device coupled with a NVidia GTX 1070 Pascal GPU; and a single-socket 64-core Intel Xeon Phi Knights Landing (KNL) device.*

Overall, HEP-Frame improved the performance of the `ttH_as`, `ttH_sci` *and* `ttH_scinp` *applications over their original sequential implementation, by 30x, 252x and 185x on the KNL server and by 32x, 89x and 74x on an Ivy Bridge server with a Kepler GPU. HEP-Frame ensured efficient execution of both memory- and compute-bound pipelined data stream applications portable across various homogeneous and heterogeneous servers with accelerators (in clusters and grid/cloud environments), without requiring any modification of the code, configuration by the user, or prior knowledge of the system characteristics.*

Resumo

A componente chave desta tese é a HEP-Frame, uma framework para ajudar o desenvolvimento e execução eficiente de aplicações de streaming de dados em pipeline para servidores homogéneos e heterogéneos. Uma aplicações de streaming de dados em pipeline é um processo que converte grandes quantidades de dados experimentais em informação útil para monitorizar dados, testar hipóteses ou validar teorias. Cada elemento dos dados é processado por uma pipeline de proposições, em que cada contém uma tarefa computacional que pode ser seguida da avaliação de um critério; se esta falha então o elemento é removido do resto da pipeline.

Otimizar a performance computacional destas aplicações requer perícia para vetorizar e paralelizar eficientemente o código – nomeadamente para tirar partido das extensões vetoriais em cada processador (núcleo) e em múltiplos processadores em cada servidor com aceleradores computacionais, num cluster – que a maior parte dos cientistas não tem, ou de uma ferramenta ou framework adequada fácil de usar, que é inexistente. Com esta motivação, a HEP-Frame foi desenhada, implementada e avaliada para disponibilizar uma interface de desenvolvimento focada no utilizador, através do uso de esqueletos de código, geração automática de código e automatização do processo de compilação, enquanto que gere de forma transparente e eficiente a execução paralela do código em servidores multicore e manycore, com ou sem aceleradores de computação.

A HEP-Frame implementa um escalonador com várias camadas que se adapta às aplicações e aos servidores computacionais em tempo de execução e processa as proposições e vários elementos de dados em paralelo, distribuindo-os entre os recursos computacionais disponíveis. A camada superior distribui os dados e tarefas computacionais entre os servidores num ambiente de cluster heterogéneo, usando uma abordagem demand-driven para permitir que a HEP-Frame escale com múltiplos servidores.

A camada intermédia ajusta dinamicamente o número de fios de execução alocados para a leitura e inicialização paralela de dados, e para a execução da pipeline.

A HEP-Frame também disponibiliza uma grande variedade de geradores de números pseudo-aleatórios eficientes em duplo-buffer com distribuições uniformes e Gaussianas, que são normalmente necessários por estas aplicações. Estes podem ser executados nos servidores de computação, ou passados para outros servidores multicore/manycore ou para aceleradores manycore/GPU. A abordagem duplo-buffer esconde os tempos de acesso para transferir dados de e para os outros servidores ou aceleradores.

A avaliação quantitativa da HEP-Frame usou três versões de uma aplicação real, a análise de eventos da física de partículas $t\bar{t}H$*, desenvolvida e usada por investigadores do CERN:* `ttH_as`*,* `ttH_sci` *e* `ttH_scinp`*, cada com uma pipeline com 18 proposições numa ordem inicial definida pelos programadores. A primeira análise é limitada pela latência da memória enquanto que as restantes são limitadas pela capacidade computacional. As análises* $t\bar{t}H$ *são originalmente sequenciais, mas uma paralelizações multifio com OpenMP e StarPU foram implementadas para comparar com a HEP-Frame. Cinco servidores heterogéneos foram selecionados para fazer a avaliação quantitativa da performance da HEP-Frame: três servidores duplo-socket com dispositivos Xeon de 12-, 16-, e 24-núcleos (Ivy Bridge, Broadwell, e Skylake, respetivamente), sendo o primeiro usado em conjunto com um dispositivos GPU NVidia Tesla K20 Kepler e dois Intel Xeon Phi Knights Corner; um servidor de socket único com um dispositivo Broadwell de 10 núcleos com um GPU NVidia GTX 1070 Pascal; e um servidor de socket único Intel Xeon Phi Knights Landing (KNL) com 64 núcleos.*

A HEP-Frame melhorou a performance das aplicações `ttH_as`*,* `ttH_sci` *e* `ttH_scinp` *por 30x, 252x e 185x no servidor KNL e por 32x, 89x e 74x no servidor Ivy Bridge com o GPU Kepler, em relação às suas versões sequenciais. A HEP-Frame garantiu execução eficiente e portável de aplicações de streaming de dados em pipeline limitadas pela memória e pela capacidade computacional em vários servidores homogéneos e heterogéneos com aceleradores (em ambientes cluster e grid/cloud), sem requerer qualquer modificação do código, configuração por parte do utilizador, ou conhecimento prévio das características dos servidores.*

Contents

Glossary

Accelerators hardware to boost the performance of specific computations

Acyclic graph a directional graph with no cycles

API Application Programming Interface, a set of functions to access the functionalities of a given application

CUDA Compute Unified Device Architecture, an API to develop code for CUDA-enabled devices

DP Data Processing, a task that processes a dataset element

DS Data Setup, a task that combines input reading, data structure creation, and initialisation

GPU Graphics Processing Unit accelerator device

HEFT Heterogeneous Earliest Finish Time, an heuristic for load balancing in heterogeneous platforms

Heterogeneous server a server with multicore processing units and accelerators

Homogeneous server a server with multicore processing units

KNC an Intel Xeon Phi manycore co-processor of the Knights Corner micro-architecture

KNL an Intel Xeon Phi manycore server of the Knights Landing micro-architecture

Manycore a specialised processing unit designed for a high degree of parallel processing

Multicore a processing unit with multiple processing cores

Multiprocess code parallelisation that resorts to the use of multiple software processes

Multithread code parallelisation that resorts to the use of multiple software threads

PCI-Express Peripheral Component Interconnect Express, a high speed bus for computer expansion, such as connecting accelerator devices

PRNG Pseudo-Random Number Generator

Proposition a computational task that may filter out data from a pipeline

Scheduler a mechanism to manage threads, processes, and workloads across the available computing resources

SIMD Single Instruction Multiple Data, a parallel microinstruction architecture

SMT Simultaneous Multithreading, multiple hardware threads in a single physical computing core

List of Figures

List of Tables

Chapter 1

Introduction

This chapter briefly introduces the research field of this dissertation. An overview of the of the parallel computing platforms and the software features relevant for efficient application execution is presented. The key challenges to the development and efficient execution of parallel applications are listed and characterised.

Finally, a list of all the contributions which originated during the development of this dissertation and the structure of this document are presented.

Current large-scale computing platforms are becoming increasingly complex by relying on a large amount of computing servers, each with multiple multicore and/or manycore devices and often coupled with hardware accelerators. The various computational resources available inside a server with wider devices of different micro-architectures need to be adequately used to ensure that applications execute as fast as possible. Mini-clusters are the most popular computing platforms used by the scientific community, which combine a wide variety of desktop and single-node servers. These single-node and desktop servers may be classified as homogeneous, where a server only uses one or multiple multicore devices, or heterogeneous, where one or multiple accelerators are coupled to homogeneous servers using the PCI-Express interface to create a distributed memory environment.

Multicore devices, commonly addressed as CPUs, are becoming increasingly wider for the last decade, where each new chip packs more and more computing cores. For instance, AMD latest server grade multicore device has up to 32 cores (with 2-way simultaneous multithreading per core, SMT), while Intel alternative has up to 28 cores (with 2-way SMT), which can be used in multi-socket servers. The Intel manycore servers, such as the Knights Landing Xeon Phi, provide up to 72 cores with 4-way SMT on a single chip, which reiterates this "more cores per chip" trend. Hundreds of computing cores can be available in a single server, while a few years ago they could only be found in mini-clusters.

Vector computers integrate computing cores based on the Single Instruction Multiple Data (SIMD) architecture, where operations on one-dimensional vectors of independent data are executed by a single instruction. This architecture provides significant performance improvements over scalar processing for specific workloads, and was very popular among supercomputers until the 1990s. Vector computers decrease in popularity was mostly due to its lack of flexibility for general workloads and the drop in the price to performance ratio of conventional processors. Modern multicore devices have a wide variety of SIMD extensions, commonly addressed as vector instructions. It combines the vectorization potential to accelerate specific workloads with the flexibility of conventional multicore devices. While vector instruction sets, such as AVX [2] and 3DNow! [3], do not implement all the features available in the original vector computers, their utilisation is crucial to achieve the full computational performance available in a server.

A deep knowledge of the underlying architectural details of the computing devices in a server is crucial to develop code that efficiently takes advantage of the available computational resources. It is important to comprehend the key issues that impact the computing performance and efficiency of applications, such as the relationship between the cost of numerical computations, memory accesses and data communications among available computing devices. These issues become imperative when using hardware accelerators, as the architecture of each device is significantly different and often employ distinct programming paradigms.

Developing an efficient parallel application, or adapting existing code, is crucial to ensure that the multiple computing cores in multicore, manycore and accelerators are not used to their full potential. Data races, resource contention, task and data scheduling, and, when considering heterogeneous servers, explicit memory transfers among devices are complex challenges that the programmer must overcome. The tuning of parallel code is often specific to an application on a server, requiring multiple iterative profiling and optimisation time consuming steps by a developer with extensive expertise in high performance computing to achieve good efficiency. These challenges become increasingly complex for non-computer scientists, which develop most of the applications that run on multidisciplinary computing clusters, as they often lack the expertise and/or time to optimise their code. The performance is specially relevant for applications that process large amounts of data, where results must be obtained in a reasonable time frame.

1.1 Motivation

Developing applications that efficiently use the available computational resources in homogeneous and heterogeneous servers requires expertise in high performance computing. The hardware in these servers poses challenges to the programmer that do not appear when developing sequential applications. These challenges become more difficult to overcome by non-computer scientists, which are the developers of a significant amount of the applications that run on high performance cluster environments. Several studies [4, 5, 6, 7] identified the causes that lead non-computer scientists to develop inefficient code:

- Most non-computer scientists are focused on problems relative to their domain of expertise, which leaves little time to develop efficient parallel code.

- Non-computer scientists are not aware of software engineering principles to produce efficient code that is also robust, modular and long lasting.

- Non-computer scientists often iteratively develop over the same application, producing legacy code (some applications currently in production are iterated on for the last 20 years), and not documenting it so that it can be used by others.

- Non-computer scientists are seldom aware of profiling and debugging tools, as well as parallelisation paradigms.

- Non-computer scientists cannot focus on getting into the architectural details of the newer generations of computing devices, reducing the efficiency and portability of their produced code.

One of the most common applications developed by non-computer scientists relate to the analysis of very large sets of experimental data, in a continuous input stream of *n-tuples*, which aim to monitor, test and/or prove hypotheses and theories. The performance of these applications is key to ensure that the large amounts of input data are processed in reasonable time. Most scientific analyses apply a set of pipelined tasks (typically > 10) on independent datasets [8, 9].

An efficient use of computational resources is crucial to significant improve by several orders of magnitude the performance of applications used in several fields of the scientific community. For instance, this is specially important in the high energy physics community at CERN, where a large amount of data needs to be processed by complex analysis applications. The resources expected to become available to process the data gathered in the third and fourth run of the Large Hadron Collider (LHC) will require an increase of the current computational resources by a factor of 2 to 5 [10]. It is crucial that applications adequately explore the computational resources available in the servers, as buying new hardware no longer guarantees a significant improvement of the performance of applications. Applications must be as efficient as possible in the use of the hardware so that research groups can make an assessment of their limitations and upgrade the hardware accordingly. *HEP-Frame* ensures that pipelined data stream applications can better utilise the available hardware resources, while ensuring that their efficiency is portable across newer hardware generations. The performance improvements already seen from using *HEP-Frame* with existing applications achieve the increased computational throughput required by the third and fourth runs of the LHC, from 2020 to 2029, without the need for additional hardware.

Pipeline stages in data stream applications typically have inter-dependencies and irregular execution times: several are computationally intensive and most filter out irrelevant data elements from further processing. Independent filtering stages can also be commutative. The execution order of the pipeline stages may significantly impact its efficiency, as their individual filtering rates and execution times are different and may change in run-time. This type of applications is also common in fields directly related to computer science, such as query engines and specific data streaming services for embedded and mobile devices [11, 12].

Efficient parallelisations of pipelined data stream applications may be complex to implement, as both task and data parallelism can be explored, and the pipeline execution is irregular. Additionally, the performance of the pipeline varies with different orders of its stages and datasets, which should also be considered by the programmer. An adequate scheduling strategy should analyse all these characteristics of pipelined data stream applications at run-time to provide an efficient workload balance among the computational resources of homogeneous and heterogeneous servers.

Several frameworks and schedulers are available to develop and efficiently execute parallel code for a wide range of applications. However, none of these tools are specifically suited for the characteristics of pipelined data stream applications, which often results in inefficient parallelisations. Additionally, frameworks have steep learning curves and schedulers are not easily integrated into existing parallel code, which may difficult their adoption by computer and non-computer scientists with moderate to low expertise on parallel computing.

This dissertation presents *HEP-Frame*, a framework designed to aid the development and efficient parallel execution of pipelined data stream applications. It provides an easy interface to develop applications, while automating the code generation of simple and repetitive tasks. A multi-layer scheduler transparently manages simultaneous input data setup and processing, parallel execution and reorganisation of the pipeline stages among the available computing resources, and application execution on multiple servers. *HEP-Frame* ensures a more efficient execution of pipelined data stream applications than other alternatives, with no modifications to the user code.

1.2 Contributions

Most software tools focus either on improving the performance or the development of specific applications. Tools that focus only on performance are often used in the computer science community, but are rarely adopted by the rest of the scientific community due to their steep learning curves. Tools to aid the development of code are widely used, specially in non-computer science communities, but they are usually computationally inefficient. This dissertation presents *HEP-Frame*, a software tool that aids the development of pipelined data stream applications, while transparently ensuring efficient execution of the code on homogeneous and heterogeneous servers. The goal is to provide a tool that bridges the gap between performance and usability for pipelined data stream applications.

This work presents a multi-layer scheduler with two key novel approaches to task and data balancing. The first is the dynamic tuning of threads assigned to data setup (input loading and data structure creation) and to data processing, which allows *HEP-Frame* to automatically adapt at run-time to the requirements of memory- and compute-bound code. The second is the list scheduling of propositions in pipelines, which assigns propositions to threads based on their execution time and filtering ratios, while simultaneously reordering them in order to avoid executing the most computationally complex.

A study on the most efficient approach to manage pseudo-random number generation is also provided, which exposes the impact that an adequate usage of these algorithms may have on application performance for multicore, manycore and GPU devices. *HEP-Frame* implements the most efficient approaches, which users can access through an API.

The work described in this document is also published in:

- **Multi-layer Scheduling with Adaptive Ordering of Pipelined Data Stream Analyses on Heterogeneous Servers** - André Pereira and Alberto Proença. Submitted to the *Journal of Parallel and Distributed Computing.*

- **HEP-Frame: a Powerful Tool to Build LHC Data Analyses** - André Pereira, António Onofre and Alberto Proença. Submitted to the *European Physics Journal C.*

- **Efficient Use of Parallel PRNGs on Heterogeneous Servers** - André Pereira and Alberto Proença. In *International Conference on Mathematical Applications,* 2018 [13].

- **Tuning Pipelined Scientific Data Analyses for Efficient Multicore Execution** - André Pereira, António Onofre and Alberto Proença. In *Proceedings of the International Conference on High Performance Computing Simulation,* 2016 [14].

- **HEP-Frame: A Software Engineered Framework to Aid the Development and Efficient Multicore Execution of Scientific Code** - André Pereira, António Onofre and Alberto Proença. In *Proceedings of the 2015 International Conference on Computational Science and Computational Intelligence,* 2015 [15].

- **Removing Inefficiencies from Scientific Code: The Study of the Higgs Boson Couplings to Top Quarks** - André Pereira, António Onofre and Alberto Proença. In *Proceedings of the 14th International Conference on Computational Science and Its Applications,* 2014 [16].

1.3 Document Structure

This document describes the work accomplished while developing *HEP-Frame.* It is structured as follows:

Chapter 1: presents the context and motivation for the work developed in this dissertation. It describes the challenges to develop efficient pipelined data stream applications and lists the contributions of this work.

Chapter 2: describes the state of the art in a compute server environment. It details the hardware architectures of homogeneous and heterogeneous servers, characterising the relevant performance features of multicore, manycore and accelerator devices. Pipelined data stream applications, the target code of the work in this dissertation, are presented and characterised. Software to develop efficient pipelined code for various computing devices is also presented, with an emphasis on scheduling strategies for the target applications. Finally, efficient implementations of popular pseudo-random number generators are presented, as they often account for a significant execution time of pipelined data stream applications.

Chapter 3: introduces *HEP-Frame*, providing a detailed description of its features to improve the speed of developing efficient pipelined data stream applications, for users with little expertise on parallel computing, and its multi-layer scheduler, which ensures portable efficient execution of these applications on homogeneous and heterogeneous servers.

Chapter 4: discusses the results of a quantitative evaluation of the efficiency features of *HEP-Frame* with three real pipelined data stream applications, used in high energy physics, on several servers with various computing devices.

Chapter 5: presents conclusions and identifies future lines of research to further develop the work based on the contributions of this dissertation.

Chapter 2

Parallel Computing Environments

This chapter presents a look into what has been done in the topics related to the work of this dissertation. The first section focus on the current state of the computing hardware available in desktop and cluster environments. The architecture of homogeneous and heterogeneous servers with multicore, manycore and relevant accelerator devices is presented, with an emphasis on key characteristics that should be taken into account when developing efficient parallel code.

Pipelined data streaming, the target application of this dissertation, is presented in detail, with a focus on its structure and computational characteristics that may influence the performance. An overview of common parallelisation approaches used in this type of code by the wider scientific community is also presented.

The relevant software tools and schedulers for efficient code execution in homogeneous and heterogeneous servers are assessed, showing an overview of their approach and possible limitations for pipelined data stream applications. The last Section presents popular PRNG algorithms, detailing their computing characteristics and limitation, as they often account for a significant portion of the pipelined data stream applications execution time.

Mini-clusters are the most common computing resource available to scientific research groups. These massively parallel systems are usually constituted by racks of computing nodes (also known as computing servers), interconnected by a low latency network. Each server in a cluster is an individual parallel system that can communicate and share data with other servers. Clusters can be homogeneous or heterogeneous computing platforms, depending on the architectures of their servers. Homogeneous clusters are built with sets of identical servers, such as type of multicore and manycore devices, and amount and speed of RAM. Heterogeneous clusters use servers with different characteristics, usually organised in groups oriented to fulfil specific application requirements. Clusters have also specific servers for centralised high throughput data storage.

Computing servers can also be classified as homogeneous and heterogeneous, depending on the characteristics of their computing devices. Homogeneous servers are single- or multi-socket systems with the same multicore or manycore device in each socket, which share a common memory address space. Heterogeneous servers use single- or multi-socket systems, similarly to homogeneous servers, that couple hardware accelerators to the multicore/manycore devices. This approach combines the flexibility of the main computing devices with the high performance capabilities of manycore devices for specific workloads. However, the multicore and manycore devices do not share a common memory address space with the accelerators, which has to be managed by software. The hardware characteristics of these servers are detailed in sections 2.1 and 2.2.

Pipelined data stream applications continuously execute a pipeline of tasks on large input datasets. This type of applications is common in many scientific fields, and is usually developed by non-computer scientists. The performance of the input pre-processing and the task pipeline has a direct impact on the overall execution time of an application. The development of these applications should take into account strategies to efficiently parallelise their execution to adequately explore the computational resources available in both homogeneous and heterogeneous servers.

The mainstream industry is still adopting the use of multicore architectures with the purpose of increasing their processing performance, which reflects in a lack of academic training on code optimisation and parallel programming. Efficient parallelisation of code for these types of servers requires expertise and time to tune the implementations that most developers lack, specially for servers that combine devices with different architectures and programming paradigms. These factors are specially relevant for non-computer scientists, which are usually self-taught programmers, as they often develop pipelined data stream applications for their researches.

Developing code for homogeneous servers requires knowledge of the underlying hardware architecture of multicore and/or manycore devices and their interconnections. Shared memory, cache coherence and consistency and data races are architecture-specific aspects that the developer does not face in sequential execution environments. However, these concepts are crucial not only to efficiently use the computational resources, but also to ensure the correctness of applications.

Developing code for heterogeneous servers poses additional challenges. Each accelerator device has an unique hardware architecture and programming paradigm designed for a specific workload, which must be mastered to develop code that adequately uses the available resources. For instance, algorithms optimised for multicore execution cannot be easily ported for these devices expecting high performance, and sometimes require a complete re-design of their implementation. Executing an algorithm in multicore/manycore and accelerator devices simultaneously creates an additional layer of complexity, as it requires the implementation of complex workload distribution strategies to ensure that all devices are adequately used.

Several libraries and frameworks are available that are designed to help the parallelisation and workload distribution of data and tasks among the computing devices of homogeneous and heterogeneous servers. Libraries mostly focus on the parallelisation of user-specified sections of an application, requiring minor modifications to the code. They provide simple parallelisation and scheduling techniques that are useful for a wide range of applications. Frameworks target specific types of applications and impose more restrictions to the code structure, such as requiring specific data structures to be used, but provide better computational efficiency. The most relevant libraries and frameworks for efficient code execution on homogeneous and heterogeneous servers are detailed in Section 2.4.

Pseudo-random number generation (PRNG) is a compute intensive task required by most pipelined data stream applications. Popular PRNG algorithms are presented, addressing the different approaches for execution in parallel environments. Several PRNG implementations available in Intel, ROOT, PCG and cuRAND libraries are evaluated on various multicore, manycore and GPU devices in Section 2.5.

2.1 Homogeneous Servers

Homogeneous servers are the most common computing platforms used in cluster environments, and contain a single or multiple multicore CPU or manycore devices of the exact same manufacturer and model. Figure 2.1 presents the organisation of a dual-socket homogeneous server. Each device has its own memory hierarchy, which usually contains L1, L2 and L3 caches and a RAM memory bank, and an high bandwidth interface that allows connecting to other devices inside the same server, such as Intel QuickPath Interconnect [17] or AMD Infinity Fabric [18]. Devices in multi-socket server connect to each other using one of these interfaces, which enables sharing of the entire memory hierarchy address space in a Non-Unified Memory Architecture (NUMA).

In a NUMA shared memory address space, a computing core can access data in the memory hierarchy of its device and any other connected device, but with different access latencies. Accessing data in the memory bank of any other device has an increased time penalty as the request and data transfer has to pass through the device interconnection and has to be managed by the memory controller on the other device. Ideally, the threads and/or processes of a parallel application should only access data in their closest memory bank, as accessing data in other memory banks results in an increased time penalty that has an impact on the application execution time.

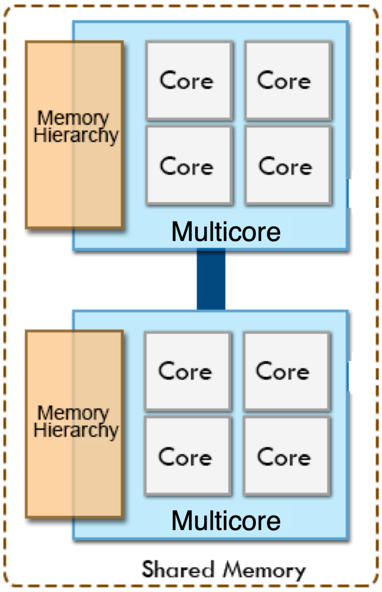


Figure 2.1: Schematic representation of a dual-socket homogeneous server with two multicore CPU devices.

2.1.1 Multicore Devices

Gordon Moore predicted in 1965 that for the following ten years the amount of transistors on CPU devices would double every 1.5 years [19]. This was later known as the Moore's Law and it remained valid until recently, and meant that smaller and, consequently, more transistors in a chip allowed for an almost linear improvement of the hardware performance. This phenomena created an environment where software developers did not spend much effort optimising the computational efficiency of their applications as the code would get faster due to the incremental improvements of the hardware.

The CPU clock frequency, which has a direct impact on how fast microinstructions are executed, improved linearly until 2005, where thermal dissipation limited further improvements. This lead manufacturers to improve the CPU throughput, rather than their speed, by adding more processing units (known as cores) to a single chip, creating the first multicore devices. These devices had lower clock speeds, as well as energy consumption and operating temperatures, but could execute more microinstructions in each clock cycle. This marked the beginning of the multicore and parallel computing era, where every new generation of CPU devices has a larger throughput, while maintaining the same clock frequencies. However, software developers now have to produce parallel code to take advantage of the multiple computing cores in multicore devices, to ensure efficient execution of applications.

Multicore devices are designed as general purpose computing units based on a set of small processing units attached to a very fast hierarchical memory (cache, whose purpose is to hide the high latency access to global memory). They are capable of delivering a good performance in a wide range of workloads, from executing simple integer arithmetic on scalar values to complex branching and vector processing. A single core implements, at the hardware level, various mechanisms to improve the execution performance of applications. The most relevant to a software developer are:

ILP: Instruction Level Parallelism is the overlapping of microinstructions that would otherwise execute sequentially, and can be performed at both the hardware and software level. At the software level, compilers attempt to identify and group independent instructions that can execute simultaneously, according to the available hardware resources. Developers can expose independent instructions to the compiler by, for instance, overlapping additions and multiplications on independent data, as these instructions can be processed simultaneously by different Arithmetic and Logic Units (ALUs).

Vector instructions: is an extension to an instruction set based on the Single Instruction Multiple Data (SIMD) model, where a single instruction is simultaneously applied to a large set of independent data. Multicore devices have specialised registers and ALUs to execute this type of instructions. Developers can produce vectorized code by using intrinsic instructions of a high level language, which is best suited for complex algorithms, or, in most cases, by indicating to the compiler which sections of the code are suited for automatic vectorization. However, the developer is responsible for assessing that code is adequate to be vectorized, as forcing the vectorization of unsuitable code can cause a degradation in the application performance.

Simultaneous Multithreading (SMT): is the hardware support for the execution of multiple threads in a single core of a multicore device, where several threads can run at any given time. This is achieved by replicating part of the core hardware resources, such as registers, to improve the use of the available ALUs in a core. If a piece of code in a hardware thread stalls waiting for data, a second thread is scheduled to execute using the ALUs that would otherwise be idle. SMT can reduce the synchronisation penalties between multiple software threads of an application, as data sharing is faster since they are both executing on the same physical core. The use of SMT can improve the performance of memory intensive applications but hinder compute intensive code, so it is the responsibility of the developer to assess if an application could benefit from this feature.

2.1.2 Manycore Devices

The Intel Many Integrated Core (MIC) architecture, implemented in the Intel Xeon Phi devices, is a manycore device available as either a coprocessor (the Knights Corner architecture) or an autonomous processor (the Knights Landing architecture). The Knights Landing manycore server is based on the Intel Atom Silvermont architecture, with an increased number of cores per device and improved vectorization capabilities over the device it is based on.

Figure 2.2 presents a schematic representation of the Knights Landing architecture. The micro-architecture of this family offers up to 36 tiles - with two cores per tile, each supporting the simultaneous execution of 4 threads - interconnected by a 2-dimensional mesh. Each core has two 512-bit vector arithmetic units that support most of the AVX-512 instruction set specification, an improvement over the previous iteration of this architecture (Knights Corner). The device has 64 KiB data and 64 KiB instruction caches per core, up to 32 MiB of L2 cache (1 MiB per tile) and access to 384 GiB of DDR4 RAM through eight memory controllers, with a bandwidth of 102 GiB per second.

It has an additional in-package 16 GiB of stacked MCDRAM memory with a bandwidth of 400 GiB per second, which can be configured, according to the specific requirements of each application, as:

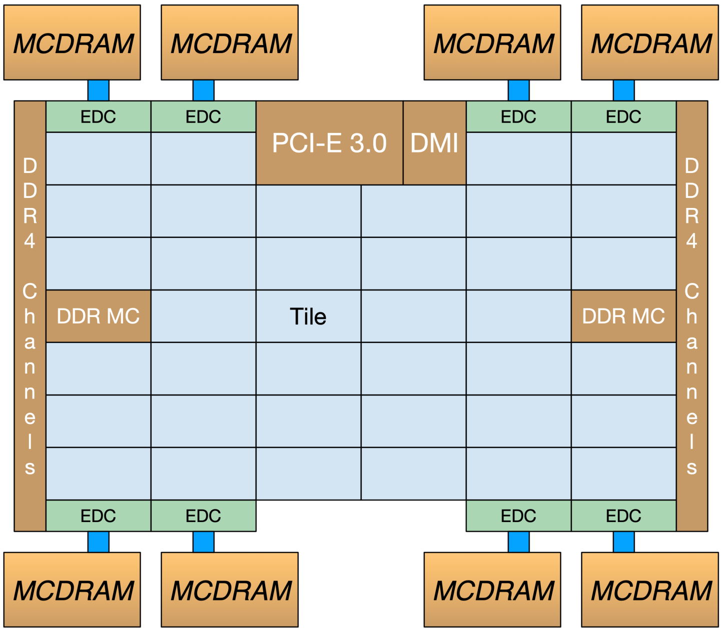


Figure 2.2: Schematic representation of the Intel Xeon Phi Knights Landing architecture.

Cache: it acts as a third cache level, and its use does not require any modifications to the application code. Frequent misses on this cache may lead to a decrease in performance. This configuration should not be used for latency-bound applications as preliminary tests show a 20% increase in latency over directly accessing the DDR4 RAM.

Flat: it becomes an addressable Section of the global memory address space. Applications can allocate data structures on this memory by using specific instructions in their code.

Hybrid: it is split evenly between the two previous configurations, with 8 GiB configured as cache and the other 8 GiB as flat memory.

The mesh structure of the cores interconnection allows the user to organise them in different clustering configurations. Each configuration has a direct impact on the cache consistency and coherence protocols, as well as in communication and latency penalties since certain tiles are closer to specific MCDRAM banks than others. The clustering configuration affects both the application parallelisation implementation and its performance:

All-to-All: the whole memory address space shared among all tiles in the device. However, a tile in the middle of the mesh will have higher cache miss penalties than a tile closer to the MCDRAM bank. This configuration also allows sharing data among a large amount of tiles, which can lead to an increased overhead of the cache consistency and coherence protocols.

Hemisphere/Quadrant: divides the tiles into 2 or 4 computing sections of the chip, which are still presented to the user as a single computing device. These sections may share data but the memory controllers only manage their respective Section, which ensures lower L2 cache and MCDRAM miss penalties. Sharing memory among sections has an increased overhead over the All-to-All configuration. Users should code their applications in order to minimise shared data but no major modifications to the code are required.

SNC-2/SNC-4: sub-NUMA cluster partitions the tiles into 2 or 4 independent computing sections of the chip, which are presented to the user as different devices. This reduces cache management overhead and penalties to access MCDRAM, as tiles in a sub-cluster can only share data among each other and access their respective MCDRAM banks. Communications between clusters must be explicitly coded using a Message Passing Interface [20] library. Users can develop hybrid multithread-multiprocess NUMA-aware code to take advantage of this organisation by pinning processes to certain clusters.

The Knights Landing device uses the same x86 instruction set as conventional multicore devices, which ensures an increased compatibility of applications and libraries. The code still has to be compiled specifically to this architecture to adequately use the AVX-512 vector instruction set. However, the user is still responsible for developing efficient code considering the architecture of this device so that significant performance improvements are obtained, as compilers optimisations are still very limited [21].

2.2 Heterogeneous Servers

Heterogeneous servers combine the flexibility of multicore devices with the specialised performance of hardware accelerators in a single system. This type of servers is the most common among general consumers, as most laptop and desktop computers contain a multicore device, used for general computation, coupled with a Graphics Processing Unit (GPU) accelerator, which is specialised for image rendering. Both the scientific community and the industry are adopting this type of servers into their computing clusters due potential of the accelerators to greatly improve the performance of specific workloads.

These servers contain one or multiple multicore devices in a shared memory environment, similarly to an homogeneous server, which share a single interface to communicate with any coupled accelerators, as shown in Figure 2.3. Applications must explicitly handle memory transfers between accelerator and multicore devices since the memory address space is distributed. This interface is usually PCI-Express, which has a theoretical peak bidirectional bandwidth of 16 GiB per second for its 3.0 revision [22] (4.0 is available but it is not yet widely adopted). This low bandwidth interface is often a significant bottleneck in applications that regularly communicate with accelerators. The IBM Power9 multicore devices support the NVLink 2.0 interface to directly connect to compatible NVidia GPU devices, offering a bidirectional bandwidth of 150 GiB per second [23], which provides a significant improvement over the PCI-Express interconnection. However, this interface is not supported by other multicore and accelerator device manufacturers.

Accelerator devices are usually built with a large amount of small processing units designed to perform simple operations, as opposed to the large and complex cores found in multicore devices. These devices are best suited for massive parallel problems, where the same instructions are applied to a large amount of independent data (SIMD execution), similarly to vectorization on multicore devices. They are designed to perform these tasks extremely efficiently, freeing the multicore devices to perform more complex computations that may benefit less from such a high degree of parallelisation. Compute intensive tasks such as particle interaction simulation, molecular docking and training and inference of deep neural

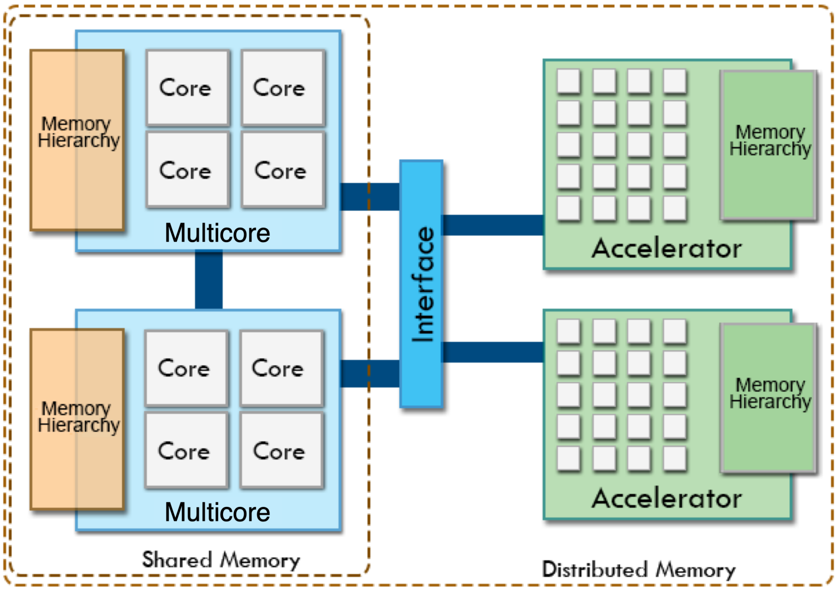


Figure 2.3: Schematic representation of a dual-socket heterogeneous server with two multicore CPU devices coupled with two accelerator devices.

networks rely on numerical methods that are processed faster on specific accelerators than on multicore devices.

As of November 2018, 41 of the first 100 clusters on the TOP500 list [24] use either NVidia or Intel accelerator devices, which reiterates the importance of developing efficient code for heterogeneous servers. NVidia GPUs are the most used accelerator, with Kepler, Pascal and Volta based devices on 38 of the top 100 clusters. A similar list focused on consumption efficiency (computing power per Watt) Green500 [25] includes 8 clusters with NVidia GPU accelerators on the top 10. The most popular accelerator devices will be presented through the next subsections.

2.2.1 Graphics Processing Units

The Graphics Processing Units (GPUs) are one of the first hardware accelerators generally used to improve the performance of specific workloads. They were initially designed to improve the performance of rendering computer graphics, which started as simple pixel drawing and evolved to support complex 3D scenes that require complex operations, such as transforms, lighting, rasterisation, texturing, depth testing and image display. The GPU architecture is based on the SIMD execution model. Image synthesising is, from the computational point of view, the processing of a large set of values that represent pixels. The processing of each individual pixel usually does not depend on the processing of any other pixel in the image. This allows all pixels in an image to be processed simultaneously by different computing units in the GPU.

Due to the industry demand for customisable shaders, GPUs later allowed some programming flexibility so that developers are able to modify the image synthesis process. Since image synthesis with custom shaders is similar, from the technical perspective, to apply a given algorithm to a 2D matrix, some researchers saw the potential to use these devices to boost the performance of numerical computation. As GPU manufacturers allowed more flexibility to program their devices, the High Performance Computing (HPC) community started to use them to improve the performance of specific massively data parallel problems. The HPC community demand for these devices pushed manufacturers to add features related to numerical computation into GPUs, such as support for double precision floating point arithmetic. This later lead to the creation of GPUs specifically designed for scientific computing.

NVidia is the main GPU manufacturer for scientific computing GPUs, with a wide range of available devices known as Tesla. These devices characteristics differ from general purpose GPUs since they have more GDDR RAM, a different physical design of the printed circuit board to fit in cluster nodes and different cooling options. These chips also have minor architectural modifications, such as more single and double precision processing units and larger memory caches.

Kepler is the second most common architecture of NVidia GPUs on the Top 500 list [24], and has been used to evaluate the performance of the framework proposed in this dissertation. Figure 2.4 shows the Kepler architecture organisation in two main components: the set of Streaming Multiprocessors (SMX), which can be loosely compared to cores in a multicore device, and the internal memory hierarchy.

There may be up to 15 SMXs in a single chip, which are complex processing units re-

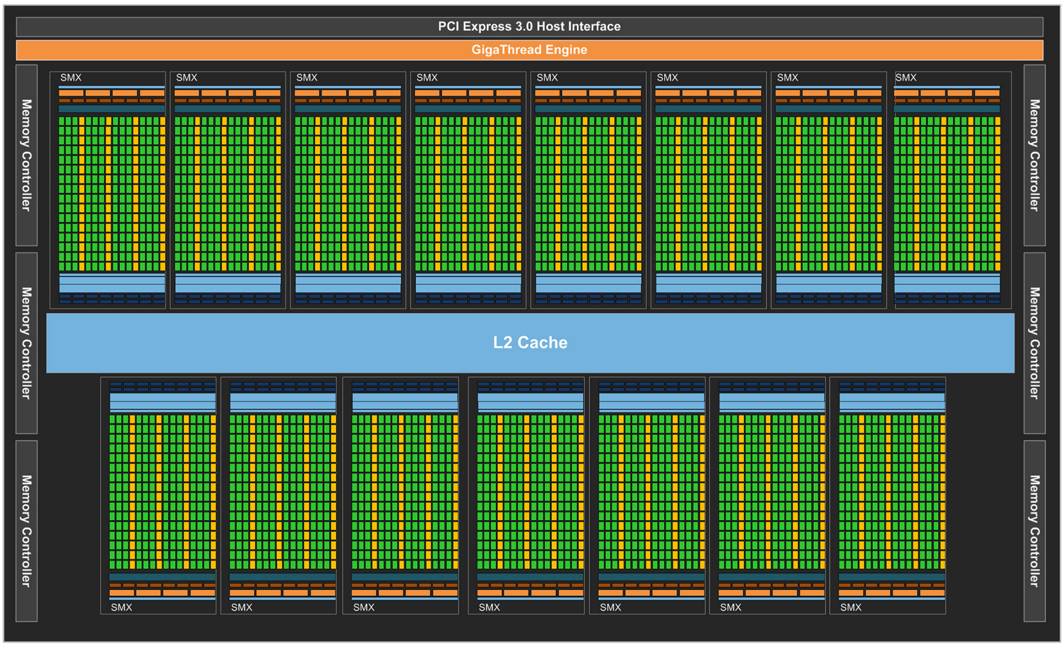


Figure 2.4: Schematic representation of the NVidia Kepler architecture (obtained from [1]).

sponsible for executing the microinstructions in the GPU. Each SMX contains 192 single precision and 64 double precision CUDA cores, small processing units capable of performing basic arithmetic computation, 32 special function units, which perform complex computations such as trigonometric operations, and 32 load and store units. These computing units operate synchronously at the GPU main clock rate.

Each SMX has 64 Ki 32-bit registers, for a maximum of 255 registers per CUDA thread (further detailed in Subsection 2.4.1), a 64 KiB very fast memory for L1 cache and shared memory, and a similar 48 KiB memory cache for read-only data. Finally, the Kepler architecture provides 1.5 MiB of L2 cache shared among all SMX units. The high end Kepler device is the Tesla K80, which has a bandwidth of 280 GiBytes per second to its main memory. The bandwidth for communications between multicore and GPU devices is restricted to only 16 GB/s (8 GB/s in each direction of the channel) by the PCI-Express 3.0 interface.

CUDA threads execute in groups of 32, addressed as a warp, which simultaneously apply the same microinstruction to 32 different data values, using the SIMD execution model.

This behaviour is similar to vector instructions used in conventional multicore devices that implement vector extensions to their instruction set.

Kepler implements several features to improve the usage of its computational resources:

Dynamic Parallelism: a kernel (algorithm coded in CUDA) running on the GPU is capable of calling itself recursively, which allows to dynamically generate new workload to process without the CPU interference. This improves irregular algorithms performance on the GPU and reduces the communications to the CPU, as it is capable of adapting to the workload.

Hyper-Q: this technology increases the amount of work queues to 32 simultaneously hardware managed connections. It allows for multiple cores in a multicore device to launch different kernels on the GPU simultaneously, improving the device resource usage. Multiple threads of an application are able to share the GPU resources and transfer memory simultaneously through independent channels.

Grid Management Unit: allows scheduling multiple grids simultaneously, which allows for different kernels, from possibly different threads, to run concurrently (in combination with Hyper-Q).

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

The most recent NVidia architecture, Volta, provides many improvements over Kepler. It has up to 80 SMX on a single chip, with up to 96 KiB of memory for L1 cache and shared memory, 6 MiB of L2 cache, and a GDDR RAM bandwidth of 900 GiB per second, an improvement from 480 GiB per second in Kepler. It is capable of connecting to multicore devices through either PCI-Express 3.0 or NVLink 2.0.

This architecture supports up to 640 tensor cores per device, which are computing units specifically designed to perform a fused multiply accumulate operations using three 4x4 matrices in a single clock cycle. It also supports half-precision floating point operations, with nearly doubles the throughput over single-precision computation. Both these improvements in the GPU architecture are incredibly useful to improve the performance of algorithms based on matrix-matrix computations, such as neural network training and inference.

2.2.2 Manycore Coprocessors

The Intel Xeon Phi manycore coprocessor, Knights Corner architecture, was the first device in the MIC architecture lineup with the purpose of providing a performance similar to the NVidia Tesla devices for scientific workloads. The Knights Corner architecture is schematised in Figure 2.5. The design of Xeon Phi devices has a various key differences from GPUs, as this coprocessor uses fewer computing units that capable of performing more complex operations, and heavily relies on code vectorization. The current high end model, the Intel Xeon Phi 7120p, has 61 cores and 16 GB GDDR5 RAM, which connects to the multicore devices through a PCI-Express 3.0 interface.

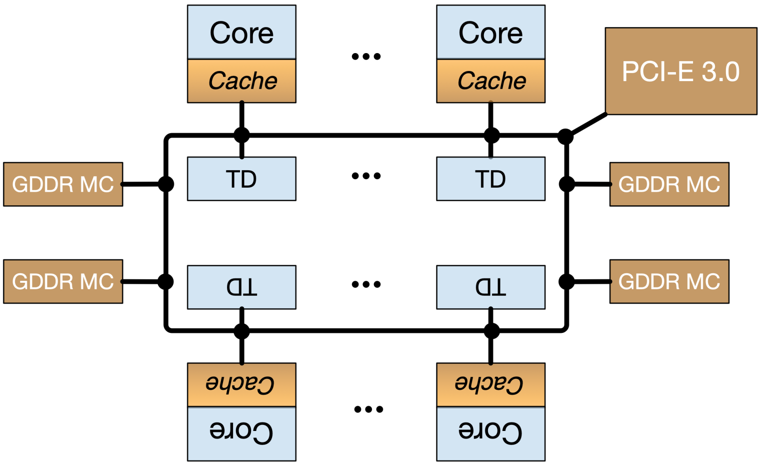


Figure 2.5: Schematic representation of the Intel Xeon Phi Knights Corner architecture.

Each core is able to run 4 threads simultaneously, and most of the parallelism is obtained by using the vectorization capabilities provided by the 32 512-bit vector registers and AVX-512 instruction set. However, only a small set of AVX-512 vector operations are implemented in the hardware, with the most complex being emulated by the compiler. This device also does not support out of order execution, which greatly compromises the use of ILP. Each core has 64 KiB + 64 KiB for data + instructions L1 cache, and 512 KiB L2 cache, and there is no shared cache among the 61 cores of the chip. The cores are connected by a bidirectional ring network that does not implement an automatic protocol for cache consistency and coherence.

The Intel Xeon Phi coprocessor supports three operating modes:

Native: the device acts as an independent server, with one core reserved for the operating system execution. The application and all required libraries must be compiled on the host multicore device specifically to run on the coprocessor, copied to the its memory along with the necessary input data, and then executed. No further interaction with the host device is required until the application finishes executing.

Offload: the coprocessor acts as an accelerator, similarly to a GPU. Only part of the application is set to run on the Xeon Phi, as implemented by its developer, and all data required by the code must be explicitly transferred between the host multicore device and the coprocessor. All library functions to be executed inside the coprocessor must be compiled and copied into its memory previous to the application execution.

Message passing: the device acts as an individual computing system in a network. Memory transfers must be handled explicitly and the code should be parallelised using a Message Passing Interface (MPI) implementation [20]. The restrictions mentioned in the previous point are also applicable.

Intel claims that current applications can be easily ported to run on the Xeon Phi coprocessor since it uses the same instruction set as conventional x86 multicore devices. This may be true for simple numerical processing applications, but an efficient port of complex applications that require the use of many external libraries is very difficult, or even unfeasible in some cases [16, 21].

2.2.3 Other Hardware Accelerators

Many alternative hardware accelerators are currently available due to the increasing popularity of GPUs and Intel coprocessors in the HPC community. Texas Instruments developed their new line of Digital Signal Processors (DSP) designed for general purpose computing while being very power efficient. They claim that these DSPs are capable of delivering 500 GFlops per second with only a 50 Watt energy consumption [26].

ARM, leaders of the mobile computing industry, are recently developing devices designed for single-node servers in cluster environments [1]. They are being adopted by the HPC community due to their high core count and vector instruction set that support 1024-bit wide vectors with a very low power consumption [27].

Field-programmable gate arrays (FPGAs) are integrated circuits that can be configured by the software developer, using a Hardware Description Language (HDL), resulting in a chip that is designed for a specific algorithm. The first FPGAs contained only a small set of programmable logic units, around 9000, but current versions support up to 50 million, which allows for complex algorithms to be translated into HDL. Currently, these devices are being used as hardware accelerators to improve the performance of specific sections of an application, similar to what GPUs do, as they can offer better performance than multicore devices at a lower clock rate and power consumption [28].

Several devices designed to accelerate specific machine learning tasks, such as training and inference of neural networks, are currently in development and/or production. These devices specialise on high throughput matrix-matrix computations, using with simple memory hierarchies and a low power consumption. The Google Tensor Processing Unit is currently on its second version and is publicly available in Google clusters. This device performs operations on 128x128 single and half precision matrices, with a theoretical performance of 45 TFlops per second [29]. The IBM TrueNorth architecture follows a similar design, where it uses large amounts of simple interconnected cores to simulate neurons [30]. The Intel Nervana architecture provides cores designed for basic matrix operations and convolutions, but

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

requires applications to explicitly manage every detail of its memory hierarchy, which reduces the need for memory management hardware and allows more computing cores to be packed in a chip [31].

2.3 Pipelined Data Streaming

A pipelined data stream application executes a set of tasks in a sequence to input data in variable sized chunks or datasets, which are previously placed into adequate data structures. Data is usually read by these applications using three different strategies: batch, mini-batch and streaming.

The batch input loads and pre-processes a pre-defined batch of data (usually an input data file) before being available to be processed by the application pipeline code. This is the most commonly used strategy and allows for input reading and initial data setup of different batches to be performed simultaneously. Pipelined data stream applications often explore parallelism by executing multiple binaries of the same application with independent batches, as it is the easiest parallelisation approach to implement.

The mini-batch input loads and pre-processes each individual dataset element, or a small set of elements, from an input batch, which allows data to be earlier available for processing by the application pipeline. However, the most common implementation of this approach relies on reading and processing each dataset element sequentially, which is often stored on global memory without an adequate structure. This approach significantly limits I/O and computing performance as data-level parallelism cannot be explored.

Streaming input continuously loads dataset elements from a given input descriptor, similarly to mini-batch, until it is signalled to stop. This approach differs from mini-batch as in the latter the amount of total data to be read is known in advance. The input data reading, pre-processing and processing have to be managed in run-time, specially when dealing with continuous processing over a large amount of time. Since the overall dataset size is unknown, this approach requires careful memory management and efficient processing of the pipeline, ensuring that both input stream and pipeline computing throughputs are similar, to avoid exceeding the available physical memory.

The input reading and pre-processing tasks may be different for each scientific field, and even among different tools within the same field. There is often standards for data file formats to ease interoperability of different software tools within a field, but each format requires different approaches, and consequently different code, to read the data from a file. This means that support for batch or mini-batch input reading depends on both the file format and the libraries to access the data.

Some file formats may also require pre-processing before loading the dataset from a file to a data structure on memory. The most common pre-processing relates to file decompression, where the data on the files need to be expanded before being moved into a data structure. This operation is often specific to a given file format, and not directly related to popular compression formats, such as *zip* or *tar*. For instance, the *root* file format, from the high energy physics *ROOT* framework, uses data compression. Applications that use `root` files need to expand its data before storing it into data structures on memory.

The processing of the dataset is based on passing each independent dataset element through a pipeline of tasks. In each task the data can be processed by an algorithm that may output a given result, may also modify the dataset element and/or may be filtered out by an evaluation of its characteristics or as the algorithm output. A dataset element that has been filtered out is not further processed by subsequent tasks in the pipeline. The pipeline processing of this type of applications is further detailed below (section 2.3.1).

Data stream applications often save information about the dataset elements that are processed by specific tasks (and/or their output) and by the whole pipeline. These results are saved in specific data structures during the processing of the dataset, which are written into files when the application finishes executing. In the case of scientific computing, these files usually follow a format specific to each scientific field and are handled by the same libraries that are responsible for reading the input files.

There is no standard for input and output file formats for pipelined data stream applications. This may introduce some limitations to the management of simultaneous I/O and pipeline processing, since a parallelisation strategy must be either fine tuned for each application or flexible enough to work with a wide range of applications.

A scientific data analysis is one of the most common types of pipelined data stream applications. It is a process that converts raw data, often obtained through experimental measurements in a scientific environment, into useful information to monitor data, test hypotheses or theory validation. This type of applications is developed by non-computer scientists, which are usually self taught programmers, and performance of the code is not their focus when creating algorithms and data structures.

A wide set of applications display the structure described in this section, such as database querying engines [11] and data streaming on compute servers and mobile and embedded devices [12]. Several scientific fields require pipelined applications with this structure. The high energy physics community relies on these data analyses, where a study of each physics channel requires a specific application to be developed. These analyses mostly filter and process data gathered from several experiments worldwide, such as the Large Hadron Collider at CERN [32], which has more than 600 institutes with thousands of researchers associated, the SNO+ Experiment at the SNOLAB Collaboration [33] and the LUX Dark Matter Collaboration [34]. The cosmology community often uses these analyses to find objects of certain characteristics, discarding most of the gathered data. Such is the case of the code being developed at the Pierre Auger Observatory [35], which includes more than 500 researchers, among others.

2.3.1 Computational Characterisation

In pipelined data stream applications, each dataset element, typically a *n-tuple* of measured data with no dependencies among different *n-tuples*, is submitted to a pipeline of propositions. In this dissertation a proposition is considered as a computational task that may be followed by an evaluation of a criterion to decide if the dataset element is discarded or further processed by the next proposition. Figure 2.6 shows a typical structure of a pipelined data stream application using a batch and mini-batch approach.

Data stream applications usually have irregular workloads: the pipeline processing time for each dataset element is variable as it may be discarded by a proposition at any pipeline stage. The execution time of each individual proposition also depends on the computational task, whose complexity may vary according to different dataset properties, and/or on memory access penalties. It is common to have several orders of magnitude separating the execution time of the simpler from the most complex tasks in the same pipeline.

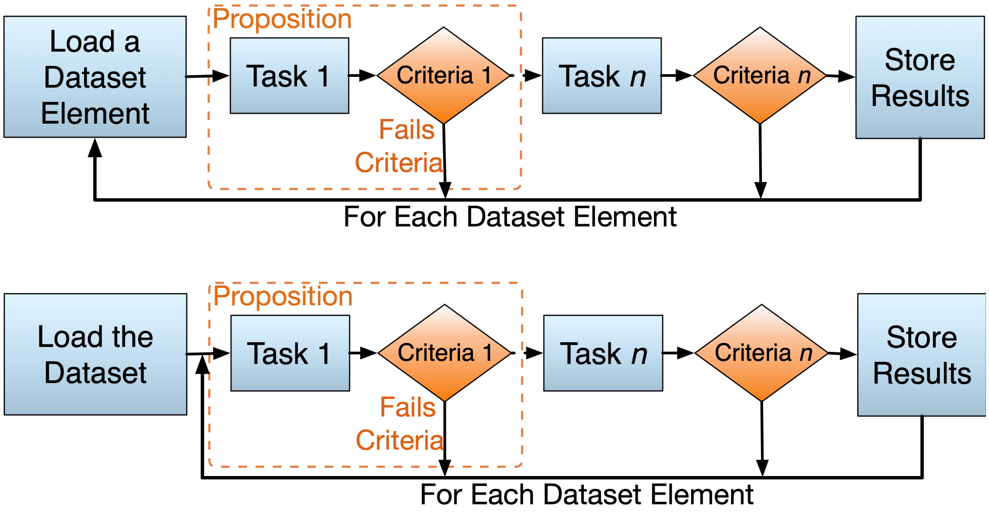


Figure 2.6: Structure of a typical flexible pipelined data stream application using a batch (top) and mini-batch (bottom) input strategies.

Pipelined data stream applications can be categorised according to their computational characteristics:

I/O-bound: the application performance is limited by the I/O latency and/or bandwidth. The pipeline does a very small amount of computation, either because the dataset elements are filtered out at its beginning or the tasks do little to no computation.

Memory-bound: the application performance is bottlenecked by the memory latency or bandwidth. Latency-bound applications require small amounts of information of the dataset element, often accessing it in irregular patterns that prevent data prefecthing. Applications limited by memory bandwidth require large amounts of information per dataset element, often accessed in regular patterns. Both perform a low to moderate amount of computation (less than 50% of the overall applications execution time).

Compute-bound: the application performance is limited by the computing power of the processing units. These applications perform a large amount of complex computations for most dataset elements.

A single application may also be I/O-, memory- and compute-bound depending on the dataset that it is processing. Consider an application with two pipeline stages: the first, p_1, is very fast and evaluates a given characteristic of the dataset element. The second, p_2, does a large amount of computation and does not evaluate any criteria. If 90% of the dataset elements in an input file are filtered out by the first proposition, the application will be I/O-bound. However, if only 20% are filtered out, 90% of the dataset reaches the second proposition, making the application compute-bound. If the first proposition filters out around 50% of the dataset, the application may be memory-bound.

The order of the propositions in the pipeline flow may have some context to the developer but it is not guaranteed that it is the most computationally efficient. Propositions with long execution times might be placed earlier in the pipeline, while propositions that filter out more dataset elements might be executed in later stages, leading to execution inefficiencies. Considering the previous example, if the proposition p_2 was placed first in the pipeline, every dataset element would be processed by it. However, most of the dataset elements would be later filtered out by p_1, meaning that most of the computation performed by p_2 would be discarded. If p_1 is placed before p_2, only a small amount of dataset elements will be processed by p_2, thus decreasing the overall execution time of the application.

An adequate ordering of the pipeline propositions may have a significant impact on the application performance, specially for compute-bound code. However, the user has to profile the code with a dataset indicative of what will be used when the application is in a production environment to order the pipeline manually. Manual tuning of the propositions order in the pipeline is usually not feasible, since:

- Input data files often contain datasets whose characteristics vary considerably, which has a direct impact on the pipeline computational performance. An evaluation of the pipeline performed with a given test dataset may provide an order that is not suited for the data processed in a production environment.

- Propositions may have complex dependency chains among themselves. Reordering them must have these dependencies into account, which may be difficult on pipelines with large amounts of propositions, without an adequate tool.

- The developer must have experience and expertise to perform this evaluation accurately.

An automated strategy to reorder the propositions in the pipeline during the application execution could overcome these limitations. It should operate transparently to the user and dynamically adapt to changes in the pipeline behaviour caused by the dataset characteristics.

2.3.2 Compute Intensive Tasks

Pipelined data stream applications, specifically scientific data analyses, often use a small pool of math and physics related functions, independently of the scientific field. These functions range from linear algebra, used for operations on vectors and matrices, numerical solvers, used to approximate solutions of equations, to Monte-Carlo methods. There are several libraries that provide efficient implementations of these routines, such as BLAS [36], MKL [37], ScaLapack [38] and OpenFoam [39]. These routines are often compute-intensive, and may be responsible for a major portion of the applications execution time.

There are two key factors to take into account when using these functions in scientific code:

Implementation efficiency: the performance of the implementation of a routine is crucial. The developer must avoid using libraries whose code does not take advantage of the available computational resources, as it may have a significant impact on the application execution time.

Adequate usage: the developer must have an idea of the underlying mathematical/physics methods implemented by the library functions, so that he/she may use the routines efficiently. For instance, excessive reset of the seed of a pseudo-random number generator with an extremely large period may have a significant impact on the application performance [16]. Also, adequate management of calls to these routines and offload to accelerator devices may provide significant performance improvements (this is further detailed in Subsection 3.3.2).

While it is crucial to use efficient implementations of compute intensive routines, an adequate usage of such algorithms has been proven to have a significant impact on an application performance. This can be achieved by either using appropriate management of these routines at software-level, or using hardware accelerators specialised for specific types of compute intensive workloads. For instance, matrix computation is often offloaded to Graphics Processing Units (GPUs), by using libraries that can manage data transfers between multicore and GPU devices and implement the required matrix operations. This can be as easy as calling any regular function, which internally manages memory transfers and executes the code on the accelerator device.

Offloading compute intensive tasks to accelerators, a task currently simplified due to specific libraries, may provide a significant improvement to the performance of an application. However, offloading code to an accelerator may lead to an inefficient use of available computing units (cores) in the server. Applications are often executed on a multicore device or on an accelerator, but not on both, as this often requires the developer to explicitly manage and schedule simultaneous computing on both devices. It requires an high degree of expertise in parallel computing as an application has to be designed and developed from its conception with interleaved computation in consideration. This may require using complex frameworks and/or libraries to run code and balance the workload on both devices simultaneously. Figure 2.7 presents a schematic representation of the offload without and with interleaving.

Finally, it is not enough to run the code on both multicore and accelerator devices. Ideally, the code should take advantage of both all cores in a multicore device and the available resources at the accelerator. An application performance can be significantly improved by minimising the downtime of the available computing resources.

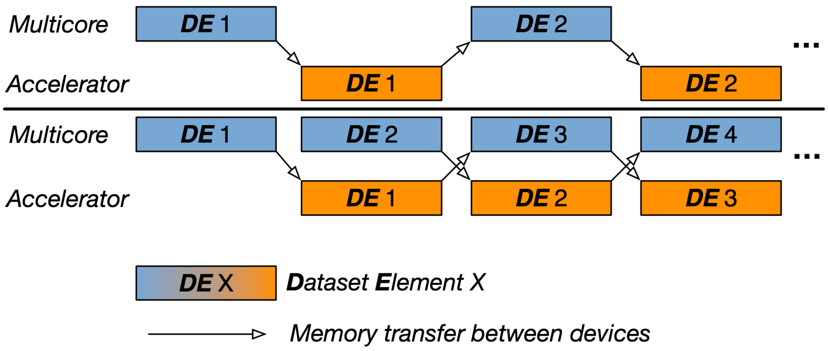


Figure 2.7: Processing a dataset element using a simple offload (top) and an interleaved offload (bottom).

2.3.3 Parallelisation Approaches

Developers of pipelined data stream applications, specially non-computer scientists, often resort to two basic approaches to parallelise the code. The first targets shared memory environments, using multiple threads, while the other focus on distributed memory environments, using multiple processes. Figure 2.8 presents a schematic representation of these approaches to process multiple input batches.

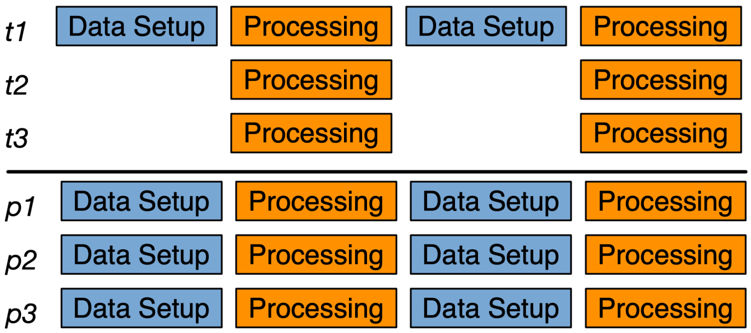


Figure 2.8: Schematic representation of the conventional multithread (top) and multiprocess (bottom) parallelisation strategies for scientific code, using 3 threads tX and processes pX.

In a multithread approach the code in execution (a process) contains sequential and parallel sections. The pipeline can be simultaneously applied to different independent dataset elements using multiple threads, through a *Map-Reduce* strategy. The dataset is mapped into various threads, using any given workload balance strategy, processed by the pipeline, and the results can either be stored in a shared or a thread private data structure. A shared data structure requires managing concurrent accesses of different threads, which may be a significant bottleneck to performance, while using a thread private structure requires a reduce procedure after processing the dataset, to merge and output the results. The input file reading, pre-processing and data structure creation (addressed as data setup) is usually performed sequentially, due to either limitations of the library used to read the files and/or it requires the developer to code a complex implementation to parallelise these tasks. This parallelisation approach is best suited for compute-bound applications, as only the computation of the pipeline is parallelised.

The multiprocess parallelisation relies on using independent parallel processes, each implementing sequential code, to execute both the data setup and the pipeline processing of different dataset elements. One approach is to implement a multiprocess parallelisation on the application, which may require the integration of a workload balance strategy to distribute the dataset elements among the processes and a reduction of the final results, similarly to the multithread parallelisation. An alternative approach is to execute the binary of the application multiple times with different input data files. The final results must be merged by an external tool after executing all instances of the application. This requires an adequate balancing of the amount of data in each of the input files, so that the various processes have similar execution times. However, statically dividing the data before the application execution is not guaranteed to ensure proper balancing of the workload, since different characteristics of the data may cause the application to vary its execution time. This is the most common alternative as it does not require the user to have any expertise in developing parallel code. This parallelisation approach is best suited for compute-bound applications, but can also benefit memory- and I/O-bound applications to a lesser extent.

Both multithread and multiprocess approaches are not adequate to explore the computational characteristics of pipelined data stream applications. They do not take into account efficient data setup, adequate ordering of the pipeline, dynamic workload balance according to the irregularity of the pipeline execution, and the complexity of the available computational resources (as seen in sections 2.1 and 2.2), which are crucial to improve the efficiency of these applications.

2.4 Software for Efficient Parallel Execution

The development of code for homogeneous or heterogeneous servers has to take into consideration different aspects of the computing and memory hierarchies of the server devices. For instance, in homogeneous servers all devices have the same computational throughput on a shared memory environment, which often does not require complex task and data scheduling algorithms. Alternatively, heterogeneous servers use devices with different computational throughputs, which depend on the type of code, in a distributed memory address space. Different code versions often have to be developed according to the architecture and programming paradigm of the target hardware device. Scheduling tasks and data on these servers requires the evaluation of many hardware characteristics during the application runtime.

The following subsections present the most relevant libraries and frameworks for efficient parallelisation and scheduling of code on homogeneous and heterogeneous servers, with a focus on what may be useful for pipelined data stream applications. Schedulers designed for the reorganisation and parallelisation of tasks in pipelined data stream applications are also presented.

2.4.1 Libraries and Schedulers for Efficient Parallel Computing

OpenMP and OpenMPI

OpenMP [40] is one of the most popular high level libraries for parallel programming in homogeneous servers. The OpenMP API is designed for multi-platform shared memory multithread programming in C, C++ and Fortran, for most multicore architectures available. It is portable, as it is implemented by each compiler, its performance is scalable for simple applications, and it aims to provide a simple and flexible interface to develop parallel applications, even for the most inexperienced developers. Its scheduler is based on a work sharing strategy, where a master thread spawns a set of worker threads to simultaneously compute a task, or different tasks, on a shared data structure. This approach is also efficient for irregular workloads. The latest OpenMP version also supports parallelisation on GPU devices, with the integration of simple offloading directives for code that can be executed either on multicore or GPUs. However, there is yet no support for simultaneous execution and load balancing among multicore and accelerator devices.

The Open Message Passing Interface (MPI) [20] is a specification designed by a consortium of both academic and industry researchers, and aims to provide a simple API for process based parallel programming in distributed memory environments, typical in computing clusters. It relies on point-to-point and group messaging communication, and is available for C, C++ and Fortran.

OpenMPI is often used in conjunction with a shared memory parallel programming API, such as OpenMP, where it handles work sharing among computing servers, while OpenMP ensures an efficient parallelisation inside each server. The OpenMPI standard does not specify the implementation of any scheduling strategies. Developers have to implement data and task scheduling strategies, explicitly code the communication of data and tasks, and create and manage an adequate amount of processes for the applications and servers used. A high degree of expertise is required to develop efficient parallelisations for complex applications.

CUDA

The Compute Unified Device Architecture (CUDA) is a computing model for hardware accelerators launched in 2007 by NVidia that aims to provide a framework to program devices with a hardware architecture similar to NVidia GPUs. It allows developers to use C with some extensions to program CUDA capable GPUs, which it converts into a specific instruction set. In CUDA, a parallel task is constituted by a set of CUDA threads that compute the same instructions in each clock cycle on different data, following a SIMD execution model.

The CUDA thread is the most basic data independent parallel task, which can run simultaneously with other CUDA threads, and it is organised in a block-grid hierarchy. A block is a set of CUDA threads that is allocated by the global scheduler to a specific streaming multiprocessor. The thread blocks are organised in a grid, which represents the whole parallel kernel (algorithm coded for the GPU). Note that both the blocks and the grid sizes must be defined by the developer according to the algorithm and dataset to process, within the maximum values supported by each different GPU architecture.

The code in a kernel indicates the task that a single thread will execute on a specific Section of the dataset, without the need for explicit parallelisation directives. The CUDA runtime engine creates a copy of the kernel and assigns it to each thread that will execute. Efficient parallelisation of tasks in CUDA should be based on computing simple tasks to large amounts of data, ideally using thousands of threads. Such large amount of threads requires a very large register bank but it contributes to guarantee that at any point in time there is a subset of these threads that has all the data it needs to execute. This strategy hides the long memory access latencies and saves chip area by including a simpler memory hierarchy on these devices. However, a high degree of expertise is required to develop efficient CUDA code.

CUDA also provides an API for C, C++ and Fortran, which is used to launch the kernels on the GPU and transfer memory between host and GPU. An application that uses CUDA must be compiled using the NVidia Compiler, which translates the kernel into GPU instructions and uses an user-defined compiler for the rest of the non-GPU code.

NVidia provides a set of libraries that supply efficient implementations of functions required in a wide range of scientific applications, which can be easily used without any expertise on parallel computing. The most popular are cuBLAS [41], cuSPARSE [42], cuSOLVER and cuDNN [43]: all automatically handle kernel execution and memory transfers of well-defined datasets between multicore and GPU devices. cuBLAS is a library for basic linear algebra operations, whose implementation is based on the BLAS library for multicore devices. cuSPARSE is similar to cuBLAS but it is optimised to handle sparse matrices. cuSOLVER is based on both cuBLAS and cuSPARSE and implements complex operations on matrices, such as factorisation, permutation and numerical solver functions. cuDNN implements primitives for deep neural network training and inference, which are based on matrix operations.

OpenACC and OpenHMPP

OpenACC [44] is a framework to efficiently parallelise code in heterogeneous servers with accelerator devices. It is originally designed to simplify the programming paradigm for servers with GPU devices by abstracting the memory management, kernel creation and code execution on the GPU. This framework focus on creating an abstraction of the GPU device to the developer, as well as ensuring functional portability across different heterogeneous servers as implemented by the compilers, rather than on ensuring efficient execution of the code. It is designed for C, C++ and Fortran and provides both an API and compiler directives. It allows only different tasks to run on both multicore and GPU devices simultaneously. OpenACC does not provide any task or data schedulers, as it does not allow a task to process different data on multicore and GPU devices simultaneously. The current specification addresses both NVidia and AMD GPUs, as well as the Intel Xeon Phi Knights Corner coprocessor.

OpenHMPP [45] is a standard that aims to provide an abstraction layer between the hardware of heterogeneous servers and the developer to ease the development of parallel applications, similarly to OpenACC. It only supports GPU devices. In the current specification, OpenHMPP uses a superset of the OpenACC directives to offload code to the GPU and manage the data transfers. It supports asynchronous task execution, but is not possible to use this framework to execute tasks on multicore and GPU devices simultaneously. This standard is only implemented by the CAPS compilers and PathScale ENZO Compiler Suite.

Schedulers for Pipelined Data Stream Applications

Scheduling traditional pipelined applications into available parallel resources has been addressed by several list schedulers, as most list schedulers can be adapted to the characteristics of this workload. However, the pipelines in pipelined data stream applications have specific characteristics, such as pipeline tasks filtering out dataset elements, that may limit the usability of typical list schedulers. The most relevant tools and libraries that provide schedulers for pipeline task reordering and parallelisation are presented next.

The authors of [46] present a scheduler to reorder the pipeline execution of database queries, where sub-queries that produce a smaller amount of relevant tuples are processed first. It does not support parallel execution of sub-queries. However, this approach only takes into account the amount of tuples each sub-query produces to calculate a weight, discarding their individual execution time. The proposed scheduler reorganises the sub-queries, enforcing simple barriers when the resulting tuples of two sub-queries have to be processed by a third sub-query. This approach does not ensure that dependencies between two sub-queries are respected, which is crucial for pipelined data stream applications. Even for applications with no pipeline dependencies, this approach may produce inefficient pipeline orders when each of the pipeline stages have different execution times. It is not clear that sub-queries that operate on different database tables (i.e., datasets) are managed and executed simultaneously by the proposed scheduler.

The authors of [47] present a scheduler for pipelined streams that reorders simple filters at run-time, while adapting the order to changes in their filtering ratios. This approach takes into account the correlation of the filtering ratios of stages in a given order to provide a better prediction of the best possible pipeline order. It performs synchronous data join operations among various filters, but does not consider their execution time, similarly to [46]. It also assumes that there are no direct dependencies among stages, which can greatly limit the utilisation of this scheduler in most pipelined data stream applications. Finally, this scheduler does not support parallel execution of the pipeline or its stages.

The authors of [48] propose a scheduler that mixes task and data parallelism for pipelined streaming services. Their approach rely on StreamIt, an architecture independent language to program streaming applications, to which they integrated a map-reduce data parallelism approach, and divide the pipeline in independent sub-pipes that are mapped to different computing cores in a device, similarly to instruction pipelining in the ALUs of multicore devices. This approach assumes that data is always processed by the whole pipeline, which is not always ideal for pipelined data stream applications. It may provide an efficient scheduling strategy to parallelise this type of applications, but the lack of pipeline reordering may greatly affect its performance, as reducing the amount of data processed by compute intensive pipeline stages is crucial to reduce the overall application execution time.

The authors of [49] propose a *Predict Earliest Finish Time* scheduling algorithm to parallelise the execution of stages in a pipeline (defined as graphs of tasks) on heterogeneous servers. It considers an optimistic cost table, which considers the weight of the pipeline stages and the attempts to predict the impact that a change to the order will have on the pipeline performance, taking that prediction into consideration to produce a final weight. The order of the pipeline stages execution is defined based on these weights. This approach assumes that all stages of the pipeline are executed for each dataset element, and poses the same limitations of [48] for pipelined data stream applications.

The authors of [50] propose a programming model to parallelise and schedule the execution of irregular pipelines in stream applications, specifically designed for highly parallel heterogeneous servers. This approach dynamically adapts to changes in the execution times of pipeline stages during the application execution, resorting to statistical information to proactively react to the irregularity of the pipeline processing. It considers that the whole pipeline is always processed, which is the case of most pipelined data stream applications.

The authors of [51] propose a set of algorithms for list scheduling of tasks in multicore devices. However, the schedulers support parallel tasks, whose implementation by the user may have a direct impact on the overall application performance, by adapting task distribution according to the computing device throughput on these tasks. These algorithms reorganise tasks in a pipeline according to their inter-dependencies, but assume that none filters out dataset elements, similarly to other schedulers, and was only tested in multicore devices.

The authors of [52] propose RaftLib, a C++ template library for efficient parallel processing of stream applications. This library provides an extensive amount of features required to build a data stream application, allowing users to develop simple processing kernels, and providing primitives to build a computing pipeline with those kernels. It provides several approaches to define forks, joins and synchronisations among kernels, similarly to [46, 47], but does not focus on data and task scheduling.

The authors of [53] present StreamBox, an out-of-order data parallel engine with parallel execution of pipelines on independent data streams. This tool is the one that is the closest to the *HEP-Frame* scheduling pipeline strategy presented in this dissertation. However, this tool displays yet a set of limitations that makes it unsuitable for the specific type of pipeline data stream applications this dissertation addresses:

- Data parallelism is achieved by simultaneously processing concurrent input streams, as well as batches of data inside each stream. However, out-of-order pipeline parallelism does not include parallel execution and reordering of pipeline tasks. Instead, StreamBox improves the processing latency of specific data tuples by reordering and processing data in parallel inside data batches.

- StreamBox assumes that all tasks in a pipeline are executed and no data is filtered out, while *HEP-Frame* tackles performance issues where data may not be processed by the whole pipeline, which increases workload irregularity and is harder to schedule efficiently.

- StreamBox focus on scheduling according to the characteristics of the input data, assuming that the time to process the pipeline is negligible. Most data stream applications do not behave like this, including those in the *HEP-Frame* case studies.

- StreamBox provides a set of operators to manage multiple concurrent streams, such as merge and synchronise, as well as data reordering based on time stamps, which is out of the context of the target applications for this work.

2.4.2 Frameworks for Efficient Parallel Computing

StarPU [54] and Legion [55] are the closest frameworks to *HEP-Frame*: both target the development of efficient code for heterogeneous platforms, schedule data and task processing among threads and support efficient execution of irregular tasks. However, these tools were designed for advanced developers and lack support for flexible pipelines, where dataset elements may be discarded in intermediate pipeline stages. Other tools, such as OmPSS [56] and DAGuE [57], are available to specific balance the workload of irregular pipelined code, but do not provide multiple scheduling options and other features that are available in StarPU and Legion, and also lack support for flexible pipelines. OmPSS and DAGuE are the alternative tools that may be best suited for regular pipelined data stream applications.

StarPU

StarPU is a unified run-time system that consists of both compiler directives and an API that aims to allow developers to efficiently map parallel code into heterogeneous servers by abstracting the architecture details of these systems. This framework frees the developer of the workload scheduling and data consistency inherent from the distributed memory environment of heterogeneous servers. Task submissions are handled by the StarPU task scheduler, and data consistency is ensured via a data management library.

StarPU employs a task based approach to the programming model of an application, where the user supplies a set of kernels for each device that can be processed in parallel. Based on the provided kernel implementations (i.e., can only run on CPU, GPU, or both) and scheduling strategy, the framework handles in which device and how much data each task will compute.

StarPU attempts to improve the performance of an application by carefully considering and attempting to reduce memory transfer costs among multicore and accelerator devices. It provides a refinement of the traditional queue-based scheduling strategy by using task priority information to select which task to process among all tasks in a ready queue. However, to obtain an efficient scheduling each task priority must be defined and updated by the user during the application execution, this requires comprehensive knowledge of the problem in order to define an adequate heuristic that provides efficient data and task scheduling among the computing devices.

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

The memory consistency is automatically ensured by the framework, as it transfers the data asynchronously without the developer interaction. The data dependencies are determined by the scheduler, with some interaction of the developer, when declaring if a data structure is read/write or both. However, this requires the users to define their dataset structures according to the constructors and limitations provided by StarPU. The granularity of the data and task distribution among devices must be statically defined by the user.

StarPU requires that applications use its data structures and poses many limitations to the tasks code organisation and behaviour. It allows little flexibility in the overall application structure, which restricts the feasibility of porting existing code, specially if performed by a developer with little experience with this framework. StarPU has a steep learning curve that hinders its adoption by users with little expertise in high performance computing.

Legion

The Legion framework is a data-centric programming model for heterogeneous platforms. It relies on an extensive configuration of a pre-defined data structure at must be adopted by the applications to provide high performance parallelisation and load balancing for both multi-core and accelerator devices. It is targeted for users with extensive programming experience for heterogeneous servers with MPI, OpenCL and CUDA, and for users that aim to create high level libraries optimised separately for each architecture.

The developer needs to use a set of specific Legion data structures to ensure a subset of the provided data properties, such as partitioning and coherence. These properties need to be explicitly managed so that the framework is able to achieve an efficient distribution of the data among the computing devices. With the specified data properties, Legion uses automated mechanisms to perform the execution parallelisation, data scheduling and memory transfers of a single task on multiple devices. Similarly to StarPU, the user should supply the framework with an implementation of the parallel task for each computing device to be used. The framework uses logical regions to abstract the data handling, which can be used by the developer to enforce dependencies among different tasks. However, simultaneous execution of different tasks on the same data structure, which occurs in pipelined applications, is not the focus of this framework.

Legion handles parallel execution of irregular workloads by adapting the data chunk size scheduled for each computing device during the application execution. It dynamically partitions the data according to an history of the execution time of the tasks for each device. The algorithm for slicing the data structure must be provided by the user.

Applications can be coded in the provided Legion language or in C++, which integrates with its run-time API. The framework also provides a low level C++ API to allow programming for each specific architecture. It can be used in a shared memory mode, or in a distributed memory configuration that supports execution on large heterogeneous clusters. Legion has a steep learning curve, similarly to StarPU.

OmPSS and DAGuE

OmpSS is a *pragma* based programming model that extends OpenMP to support code offload to accelerator devices and asynchronous parallelism, which allows easy integration in existing codes by users with some experience in parallel programming. It is based in a *task* clause that defines the parallel region, in which data dependencies and transfers to and from accelerator devices is specified. It supports OpenCL and CUDA capable devices, as well as the Intel Xeon Phi Knights Corner coprocessor. However, OmpSS is limited as its parallelisation needs to be tuned for each application code and requires, in addition to other limitations in scheduling common to other *pragma* based programming models.

DAGuE is a run-time system that dynamically manages the execution of tasks, which are represented by directed acyciclic graphs, on multicore devices. It relies on knowledge of the application obtained in a pre-compilation process, such as task dependencies and ordering. Its workload scheduling is based on a simple work stealing strategy. However, flexible pipelined data analyses tasks can only be represented by directed cyclic graphs, as detailed in Section 3.2.

2.5 Random Number Generation

Random numbers are used in a wide spectrum of applications where unpredictability is required, including statistical data sampling, scientific computing, gaming and cryptography. Generally, pipelined scientific data analyses, a subset of pipelined data stream applications, often require large amounts of random numbers, which generation represent a significant amount of the execution time. Different applications often require specific properties from random numbers, for which different random number generators may be used. In the context of computer science, these can be broadly classified as True Random Number Generators (TRNGs) or Pseudo-Random Number Generators (PRNGs).

TRNGs are based in physical random processes to generate random bit strings. The most common example of a TRNG is the coin toss of a symmetrical coin, where one can expect either heads or tails with a 50% certainty. A set of coins, or a series of coin tosses, can be used to generate a random sequence of bits. However, coins are not perfectly balanced and there is a small probability of landing on its side, slightly deviating the 50-50 chances of expecting heads or tails. Post-processing may be used to remove the bias of these processes. There are no correlations among generated numbers but these generators are usually slow, not suited for large scale computing and their results cannot be replicated, which makes debugging code harder.

PRNGs attempt to approximate key properties of truly random numbers, such as no repetition of sequence of values for a long period and no correlation between generated numbers. However, the generated values are not truly random as they are determined by an initial value (seed). A proper mathematical analysis of the generator algorithm is required to assess its quality and if they are close enough to truly random for the specific use that they were designed for. The main benefit of this type of random generator is their performance, which, depending on the algorithm, may scale with the increase of available cores. The use of a seed ensures that the results are reproducible, which eases the process of debugging code. This type of generators are mostly used for scientific applications due to its higher performance and adequate mathematical properties. However, most PRNGs can only generate sets of uniformly distributed PRNs, which may require a transformation algorithm to convert them into any specific distribution.

A short introduction to the most popular PRNGs, distribution transformations and libraries follows through the next subsections.

2.5.1 Popular PRNG Algorithms

There is a wide range of algorithms to generate PRNs currently available, each with strengths and weaknesses that may make them best suited for different uses. The statistical quality of a PRNG randomness is usually evaluated by a set of benchmarks, such as the Diehard [58] and TestU01 [59] suites. An ideal PRNG has an infinite period, covers the entire range of possible PRNs (usually 32/64-bit numbers), and has no correlation between generated PRNs. Other mathematical characteristics may be equally important, but are not as relevant in the scientific community when choosing a PRNG.

The scientific community has been using several PRNG algorithms, such as the popular r1279 and Wichmann-Hill PRNG available in GSL [60], MKL [37] and NAG [61], but one stands above all other in popularity: the Mersenne Twister [62]. This algorithm was developed in 1997 and features a period of $2^{19337}-1$, passes most statistical tests, and it is extremely fast to generate both 32 and 64-bit numbers. This generator is also implemented in most programming languages and is available in most scientific computing libraries. It has some limitations, such as low throughput, but they are often overcome by alternative implementations of this algorithm, which take advantage of vector/SIMD instructions, GPU architectures and multithreaded environments.

Recently, the PCG family of PRNGs was proposed [63], claiming better statistical quality and computational performance, for both single and multithreaded environments. Even though it is not yet fully accepted by the scientific community, the PCG RXS-M-XS 64 generator (a Linear Congruential Generator, LCG) will be included in *HEP-Frame*, as the authors claims it is one of the best performing PRNGs currently available. Since the PCG generators only generate uniformly distributed numbers, they will be paired with an efficient implementation of the Box-Muller algorithm to produce Gaussian distributed numbers.

2.5.2 Transforming Uniformly Distributed PRNs

PRNs are usually generated in an uniform distribution, but other distributions may be required. Gaussian distributed PRNs are often used in scientific computing, so having PRNG implementations that support that functionality is crucial.

Since most algorithms only generate uniformly distributed PRNs, this distribution may require post processing. One of the most common algorithms is the Box-Muller transformation [64], which generates a pair of independent Gaussian distributed PRNs based on a set of uniformly distributed numbers. It is not one of the most computationally efficient transformations, due to its iterative nature and reliance on square roots, logarithmic and trigonometric functions.

The Inverse Transform Sampling [65] is a method that transforms uniformly distributed numbers into any distribution, given its Cumulative Distribution Function (CDF). The CDF maps a PRN into a probability between 0 and 1 and then inverts this function, providing the final non-uniformly distributed number. This number can be adjusted to a specific mean and standard deviation afterwards, as required by a Gaussian distribution. The lack of an analytic CDF for the Gaussian distribution may affect the algorithm performance, favouring other transformations such as the Box-Muller. However, current implementations, widely accepted by the scientific community, use an extremely accurate approximation of the Gaussian CDF, which is faster than most transformations.

The computational performance of both Box-Muller and Inverse Transform Sampling methods (with the CDF approximation) will be assessed and evaluated on real scientific case studies. Other methods could be used, such as the Ziggurat transformation [66], but are not included in this work as they are not used as often by the scientific community.

2.5.3 PRNG Libraries

Most scientific computing libraries and frameworks provide efficient implementations of a wide variety of PRNGs. MKL is one of the most popular scientific computing libraries that offers a wide range of mathematical functionalities. It features several PRNGs, from which only the Mersenne Twister will be considered as it would be the most likely to be used by the scientific community. The Box-Muller and ICDF (Inverse Transform Sampling) transformations are available in this library and will be used to convert uniformly distributed PRNs into a Gaussian distribution. MKL also provides an implementation of most algorithms that generates a batch of PRNs. This library is highly optimised, with multithreaded and vectorized functions, for most Intel multicore and manycore devices.

The fastest PRNG available in the PCG family, the RXS-M-XS 64 (LCG), will be coupled to the Box-Muller algorithm to provide Gaussian distributed pseudo-random numbers.

To offload the PRNG to the CPU accelerator the NVidia CUDA toolkit includes a library of PRNGs, cuRAND [67]. This library provides an efficient implementation of several algorithms and transformations, from which the most relevant for scientific computing is the Mersenne Twister algorithm and the Box-Muller transformation.

2.6 Summary

This chapter presented the structure of homogeneous and heterogeneous servers, with a detailed overview of the hardware architecture of the multicore, manycore and accelerator devices present in these servers. The advantages and limitations of different hardware architectures were discussed, as well as their impact on real world software applications. The most relevant libraries, frameworks and schedulers for the development and efficient parallel execution of pipelined data stream applications for these servers were presented, with a description of their programming models, advantages and limitations.

A pipelined data stream application, the target type of applications of the work in this dissertation, is a process, often developed by non-computer scientists, that converts raw data (often from experimental measurements) into useful information to monitor data, test hypotheses or prove theories. Large amounts of experimental data are read in variable sized chunks or datasets, and placed into an adequate data structure. Three approaches are commonly used to input data into a data stream application: in batches, where files containing large chunks of data are read before processing the dataset; in mini-batches, where dataset elements are individually read and processed; in streams, where dataset elements are continuously read and processed until the application is signalled to stop.

Each dataset element read, which usually consists of measured data and is independent from other dataset elements, is processed by a pipeline of propositions. A proposition may be composed by a computational task followed by an evaluation of a criterion. Failing the evaluation discards the dataset element from the pipeline.

Pipelined data stream applications usually have irregular workloads, as the processing time of different dataset elements is not constant since they may be discarded at different points in the pipeline. The execution time of each individual proposition also depends on the computational task, whose complexity may vary according to different dataset properties and/or on memory access penalties.

Scientific data stream applications often require large amounts of pseudo-random numbers, whose generation may account for a significant amount of its execution time. There are several algorithms for PRN generation (PRNG) used by the scientific community, each designed for a specific usage and a given statistical quality. The Mersenne Twister is the most popular PRNG due to its high statistical quality and efficient implementations for multicore, manycore and GPU devices. Using an efficient implementation of an adequate PRNG may provide significant performance improvements for pipelined data stream applications.

Optimising the computational performance of pipelined data stream applications requires a high degree of expertise that most computer and non-computer scientists lack. Improving the performance of this type of code poses various challenges:

- Different applications may be limited by different characteristics of the hardware, making them I/O-, memory-, or compute-bound. A single application may also be limited by these different factors, as processing various datasets may require different amounts of computation. This prevents a single optimisation approach to be useful for every pipelined data stream application.

- The default order of the propositions in the pipeline may be inefficient. Propositions with long execution times should be placed in the later stages of the pipeline, while propositions that filter out more dataset elements should be placed earlier. However, these characteristics of the propositions cannot be measured prior to the application execution, as they often vary while processing a dataset and between different datasets.

- Applications often use compute intensive math- and physics-related routines that contribute to a significant part of their execution time. Using efficient implementations of these routines may not be enough to ensure efficient execution of the code. Adequate management of data and/or computation of these routines is as important as their implementation, specially when offloading their execution to accelerator devices.

- Simple multithread and multiprocess parallelisation approaches can be implemented by non-computer scientists. They may work to improve the performance of compute-bound applications with simple pipelines, but are usually not enough to take advantage of a server computational resources for most applications, specially if they are I/O- and memory-bound.

- Optimising a single application for a specific target computing system requires a large amount of time, which scientists often prefer to spend developing new features and improving existing algorithms. Moreover, it is common to work with multiple applications simultaneously, which would require to replicate and tune those optimisations for every application and computing system.

- Optimisations tuned for a specific application often require modifications to the scientists implementation of some algorithms, which if performed by a third party may jeopardise the scientists confidence on the correctness of the code. It also hinders the maintenance and upgradability of the existing code.

Chapter 3

HEP-Frame: a Highly Efficient Pipelined Framework

This chapter presents HEP-Frame, a framework to aid the development and efficient execution of pipelined data stream applications on homogeneous and heterogeneous servers. The structure of the framework is detailed. A simple demonstration on how interact with the HEP-Frame tools and how to develop the code for the key components of an application is also provided.

The multi-layer scheduler, a key component of HEP-Frame to improve the performance of pipelined data stream applications, is presented. The parallelisation strategies implemented in each scheduler layer, and how they influence the performance of these applications, are described in detail. Finally, the mechanisms used to parallelise proposition on manycore servers and co-processors, as well as offload PRNG to manycore and GPU accelerators, are presented.

HEP-Frame is a user-centred framework to aid scientists to develop applications to analyse data from a large number of streamed dataset elements, with a flexible pipeline structure. It not only stresses the interface to domain experts so that code is more robust and is developed faster, but it also aims high-performance portability across different types of parallel computing platforms and desirable sustainability features. This framework aims to provide efficient parallel code execution without requiring user expertise in parallel computing.

In data analysis, non-computer scientists are faced with applications that inefficiently handle data processing. Since optimising applications is very time consuming and the main goal of non-computer scientists is to obtain results relevant to their scientific fields, often within strict deadlines, improving the performance of the code is usually overlooked. Having a framework dedicated to aid code design and development, through the automation of repetitive tasks, while ensuring efficient data processing, is key for many scientific fields.

Frameworks to aid the design and deployment of scientific code usually fall into two categories: (i) resource-centred, closer to the computing platforms, where execution efficiency and performance portability are the main goals, but forces developers to adapt their code to strict framework constraints, being the most relevant StarPU [54]; (ii) user-centred, which stresses the interface to domain experts to improve their code development speed and robustness, aiming to provide desirable sustainability features but disregarding the execution performance.

HEP-Frame attempts to merge this gap so that users develop code quickly and do not have to worry about the computational efficiency of the code. It handles (i) by ensuring efficient execution of applications according to their computational requirements and the available resources on the server through a multi-layer scheduler, while (ii) automatically generating code skeletons, transparently managing the data structure and automating repetitive tasks.

This framework provides a multi-layer scheduler that provides task and data level parallelism for pipelined data stream applications on heterogeneous servers. It also provides an API that provides efficient implementation and management of pseudo-random number generators (PRNGs) for multicore and accelerator devices. The key performance features of *HEP-Frame* are:

- Balancing the input data among multiple multicore and/or manycore servers (with Xeon Phi Knights Landing, KNL).

- Scheduling and execution of the loading and pre-processing of raw data into *HEP-Frame* data structures (the data setup) in parallel with the pipeline execution, through a dynamic adaptation of the number of threads assigned to read and to process the pipelined data stream.

- Adaptive reordering of the pipeline execution flow and the distribution of its stages across heterogeneous computing resources, exploring parallel execution of independent tasks in a pipeline (task parallelism) with multiple dataset elements/input streams (data parallelism).

- Balancing the data and workloads among the computing devices of heterogeneous multicore servers with accelerators, such as the manycore coprocessor Intel Xeon Phi Knights Corner (KNC) or GPU devices.

- Managing efficient implementations of several PRNG algorithms on multicore devices, KNC coprocessors and NVidia GPUs, using a single and dual-buffer approaches.

Next subsections address the *HEP-Frame* usability (the skeleton generator and associated pipeline inter-dependencies, the data structure setup, plug-ins to store results), a detailed description of the multi-layer scheduler and other performance features.

3.1 *HEP-Frame* Structure and Usability

HEP-Frame aids the development of pipelined data stream applications by providing an user-friendly code skeletons and transparently managing common repetitive tasks. The framework automatically manages the efficient execution of the code across different types of parallel computing platforms, from laptops to clusters, without requiring the user to tune the code for each platform.

The most recent version of *HEP-Frame* can be downloaded at `https://bitbucket.org/ampereira/hep-frame`. After unziping the downloaded file, the *HEP-Frame* directory is created with 3 basic sub-directories (*lib*, *scripts* and *tools*) and an additional *Analysis* directory to store the user applications.

3.1.1 Initial User Interaction

HEP-Frame should be downloaded and decompressed into the user directory where it will reside. Figure 3.1 presents a schematic representation of *HEP-Frame* directory organisation, where blue directories are standard for every installation of the framework, green directories contain plugins for a specific scientific field, and orange directories are automatically created for every data stream application.

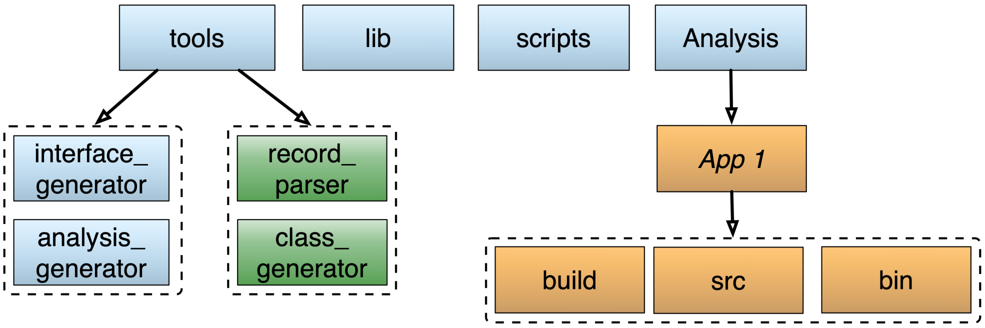


Figure 3.1: Schematic representation of the *HEP-Frame* directory structure.

The four key directories are:

`tools`**:** this directory holds the *HEP-Frame* and external tools required at compile time and may contain other field-specific compatible tools. *HEP-Frame* has four tools, which are used when creating and compiling a new pipelined data stream application (later detailed in Subsection 3.1.2).

`lib`**:** this stand-alone directory holds all the specific code and library files of the *HEP-Frame* core components.

`scripts`: this directory holds all required scripts for internal *HEP-Frame* usage, framework setup, analysis creation, pre-processing and compilation. The user should only interact with a pre-defined set of scripts (see below).

`Analysis`: this directory is automatically created when the user creates his/her first application. It automatically stores all code files and user generated in a sub-directory with the application name, which contains the typical `src`, `bin` and `build` directories. Applications may be shared among users by copy-&-paste the respective directory.

HEP-Frame provides three key scripts for the user to interact with, stored in the scripts directory: `install.sh`, `update.sh` and `newAnalysis.sh`.

The `install.sh` script installs the *HEP-Frame* core components and compiles the tools in the `tools` folder. The framework requires that the *BOOST* library is installed on the system, as its core depends on several functionalities provided by this library.

Pre-defined optional dependencies can be set when compiling an analysis, which currently are Intel Math Kernel Library (MKL) and NVidia CUDA. MKL provides several computationally efficient numerical algorithms and functions that can be used when coding a pipelined data stream application. *HEP-Frame* internally uses MKL to provide the user with efficient Pseudo-Random Number Generators (PRNGs), a compute intensive task used in several data stream applications. A detailed discussion in how PRNGs are used in *HEP-Frame* is later presented in Subsection 3.3.2. NVidia CUDA is also used to improve the computational efficiency of the *HEP-Frame* PRNG functions, by offloading this intensive task to GPUs through the cuRAND library. Use the latest CUDA Toolkit and an adequate CUDA capable GPU. *HEP-Frame* can be installed using Clang, Intel and GNU compilers. Other compilers may work but they have not been tested yet. The compiler to be used to install *HEP-Frame* and compile an analysis is set by executing `export HEPF_COMPILER=INTEL/CLANG` in the bash session. GNU compiler is used by default.

The `update.sh` script automatically downloads and installs the latest version of *HEP-Frame*, replacing the contents of the `lib`, `scripts` and `tools` folders, as well as re-compiling the framework core. The `Analysis` folder will remain untouched.

The `newAnalysis.sh` script creates a new pipeline data stream application folder and skeleton files, which are generated by the tools in the `tools` folder. This is schematised in Figure 3.1, where an application named *App 1* was created.

3.1.2 Tools to Automate the Application Development

The *HEP-Frame* run-time system handles all repetitive tasks usually performed during an application code execution, while requiring the user to provide code snippets to fill the remaining gaps, as shown in Figure 3.2.

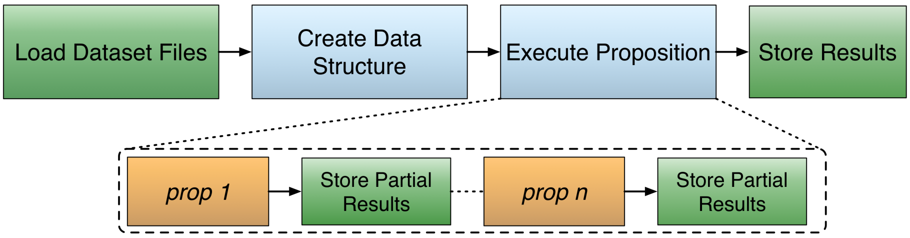


Figure 3.2: Execution flow with the *HEP-Frame*: the user provides code for the darker boxes (orange and green) and the framework run-time system manages the blue boxes.

HEP-Frame provides a toolset containing scripts for the development and compilation of pipelined data stream code. Four tools are available in *HEP-Frame*: `skeleton_generator`, `class_generator`, `record_parser` and `interface_generator`. *HEP-Frame* can also include third-party tools to automatically and transparently create the load/store code snippets and data structure specification for a specific case study (the green boxes in Figure 3.2). Additional tools can be developed by the users, similarly to plugins, so that other specific needs of their pipelined skeleton can be addressed. *HEP-Frame* currently provides the `class_generator` and `record_parser` tools, which were created to automatise repetitive code generation of applications related to high energy physics (since this is the scientific field of the case studies used to validate the performance and usability of the framework), which are used by the framework when compiling an application.

The `skeleton_generator` tool creates a skeleton with function prototypes for the user to fill in with the required code to run an application, such as propositions and their inter-dependencies, dataset file loading, dataset element class structure and result storage. Users code their pipeline stages as propositions that *HEP-Frame* will consider as black boxes. The framework does not modify the code to ensure that users trust the correctness of the algorithms and that they have total control of their code. This also allows applications to be easily updated and expanded, while working out-of-the-box with updated versions of *HEP-Frame*. Users can also organise their propositions and auxiliary functions in multiple source files, as *HEP-Frame* will automatically detect and compile these files. Propositions are passed to the *HEP-Frame* engine by calling the `addProposition (propFuncPointer* prop, string propName)` method in the `main` function in the skeleton file, which receives a proposition function pointer and an user-defined proposition name. The order by which the propositions are added will be used as the initial pipeline order. Users define the dependencies between propositions in the `main` function through the method `addPropDependency (string prop1, string prop2)`, which tell the *HEP-Frame* scheduler that `prop2` should be executed after `prop1`. The *HEP-Frame* scheduler balances proposition execution without compromising the correctness of the results based on these user-defined dependencies.

The `class_generator` automatically creates the dataset structure specification and the code to load the data from an input data file. This tool is automatically called when the user creates a new application with *HEP-Frame*, in the `install.sh` script. Currently, it supports `.root` files commonly used in high energy physics data stream applications, as the ones used as case studies, and outputs the C++ class file expected by *HEP-Frame*, which specifies the variables of a dataset element and the code to read these variables from the input file. Users can manually provide this code in the skeleton file or develop a tool that handles different file extensions to replace `class_generator`. Partial and final results can also be stored: the user provides the code in a specific function on the skeleton file. Users specify the dataset element variables to be stored, or a composition of variables (such as `var1 * var2`) by indicating their name on a specific section of the skeleton file, for the elements that either pass each proposition or the whole pipeline. The code to store these variables for each dataset element is automatically created by the `record_parser` tool when the user compiles the application. By default, it outputs a `.csv` file for each defined variable, but also supports the `.root` file format. Again, users can either supply their own code to write the variables or develop a tool to automatise this process.

The `interface_generator` creates an interface at compile time to access the *HEP-Frame* internal data structure. The propositions access each dataset element as if it is stored in global memory.

The prototype of a proposition function receives only an `unsigned` counter parameter as input, to be managed by the *HEP-Frame* run-time engine, and returns a Boolean that indicates if the current dataset element passes to the next proposition or is filtered out. The `interface_generator` translates the access to these variables into accesses to the *HEP-Frame* data structure that holds all dataset elements, through a define-based header file. For instance, in the following code `val1` and `val2` are variables of a dataset element that can be accessed as if they are declared in global memory.

```
bool prop1 (unsigned this_event_counter) {
    if (val1 > val2)
        return true;
    else
        return false;
}

// ... prop2 defined here ...

int main (void) {
    // ... automated initialisation above ...
    analysis.addProposition (prop1, "prop1");
    analysis.addProposition (prop2, "prop2");
    // prop2 must execute after prop1
    analysis.addDependency ("prop2", "prop1");
    analysis.run();
    // ... automated cleanup below ...
}
```

This example shows a simple proposition that only evaluates one criterion. Propositions can contain complex algorithms, calls to user or library functions and additional data structure creation. A proposition being executed by a given thread or process can only access the information of the dataset element assigned to it by the *HEP-Frame* scheduler. It can also access other user-defined data structures, but not dataset elements that were not yet assigned to it. These propositions are replicated by a group of threads and automatically applied to all dataset elements, according to the scheduler assessment, similarly to the execution of kernels in CUDA. How to code a sample application is further detailed in appendices A and B.

3.2 *HEP-Frame* Multi-layer Scheduler

HEP-Frame uses a multi-layer scheduler, where each layer is designed to manage and optimise a specific part of the execution of a pipelined data stream application. The multi-layer structure reflects the need to distribute the workload among distributed servers (such as clusters or grid/cloud), and to manage compute- and memory-bound codes on multicore and manycore servers with accelerators.

3.2.1 Structure of the Scheduler Layers

The top layer of the *HEP-Frame* scheduler (Figure 3.3) manages multiple datasets using a master-worker demand-driven approach on a distributed environment. Worker processes request dataset files of a predefined size to a master process until all data is processed. The mid and bottom scheduler layers present a novel approach to overcome limitations of other schedulers and tools for pipelined data stream applications.

The main focus of the *HEP-Frame* scheduler is on desktop and single-node servers, as the number of computing cores in current devices is rapidly increasing. More complex strategies could be later implemented on the top layer to efficiently schedule the workload among nodes in larger cluster environments.

The middle layer implements the multithreaded execution of a file/data-stream reading with a data structure creation and pre-processing, addressed as data setup (*DS*), in paral-

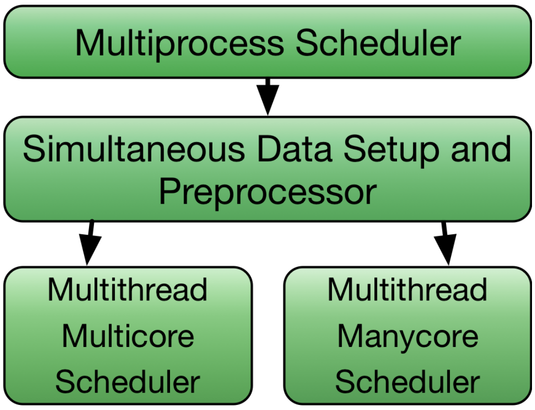

Multiprocess Scheduler
Simultaneous Data Setup and Preprocessor
Multithread Multicore Scheduler
Multithread Manycore Scheduler

Figure 3.3: Multi-layer structure of the *HEP-Frame* scheduler.

lel with the pipeline processing, addressed as data processing (*DP*). The number of parallel threads allocated to each component (*DS* and *DP*) is adjusted during the execution of the application, adapting to memory- or compute-bound code.

The bottom layer implements parallel data processing for multicore and manycore devices, which focus mostly on improving the performance of compute bound code. Propositions of the same or different dataset elements are concurrently executed, prioritising the execution of faster propositions that filter out more data, while respecting dependencies among propositions in the pipeline. This pipeline order is periodically updated. This layer also manages the distribution of the workload in a compute server coupled with manycore KNC coprocessors.

3.2.2 Multiprocess Scheduling

The top scheduler layer implements a master-worker strategy using MPI, where the master creates a pool with a set of input file names, and their respective file path, accessed by each worker to retrieve a set of files to process. Once a worker finishes processing its current set of files it gets another set from the pool (a file per *DS* thread, which is later detailed in Subsection 3.2.3). This process is repeated until the pool is empty. Several scientific fields, such as high energy physics, often process large amounts of data that is stored in relatively small files (up to 2 GiB), which provides enough data granularity for an adequate load balancing using this approach.

File sharing among processes was used instead of sharing the *HEP-Frame* main data structure because of two key limitations:

- When sharing the data structure a master process has to load all data before passing it to the worker processes. If only the files are shared, each process is able to load into its data structure, diminishing communications and parallelising the input file reading, which may have a significant impact on performance for I/O-bound code.

- The *HEP-Frame* main data structure is a container of instances of a class, whose structure and variables are defined by the user, since it depends on the input file. Communicating the data structure among processes would require either its specification to be limited, which is what other frameworks do and goes against the flexibility that *HEP-Frame* aims to offer, or the user to provide a serialisation method. Serialising a class restricts the usage of instances of other complex classes and pointers, which happens in the case studies presented in Section 4.1, and may be out of the expertise of a non-computer scientist.

Each process runs its own independent version of the subsequent layers of the scheduler, with no communication to share information about the pipeline behaviour used by the subsequent layers. Sharing scheduler information would require either synchronisation among processes, which is considerably longer than when using threads on a shared memory environment, or the data to be sent asynchronously, which meant that the subsequent scheduler layers would not use recent information about the pipeline behaviour. The latter may have a negative impact on performance when dealing with highly irregular workloads, where quick adaptation of the scheduler is crucial. The processes output the results in independent files, which are later merged by the user using third party tools depending on the file format. *HEP-Frame* cannot support file merge as it is the user responsibility to provide the required code to load input files and to write the results.

The amount of processes is defined by the user when executing an application. A generally good configuration uses one process per computing server, independently of the characteristics of the code (I/O-, memory-, or compute-bound). However, multiple processes may provide better performance for some applications on multi-socket servers, where a process is created and assigned to each multicore/manycore device on the server. This forces *HEP-Frame* to create a data structure per multicore device, which will be stored in its respective memory bank and eliminates the memory access penalties of threads accessing data on a NUMA server configuration. The impact of NUMA accesses is assessed for a set of case studies in Chapter 4. However, the current version of the scheduler does not automatically define and manage the amount of processes based on the server characteristics, as the performance improvement of using multiple processes is not consistent across applications.

3.2.3 Dynamic Tuning of Data Setup and Processing

For each supported mode to input data into the pipeline (batch/mini-batch from files and streaming, as seen in Section 2.3) the user must provide the code to load a single data element from the chosen input type. The *HEP-Frame* scheduler simultaneously processes the data setup (*DS*) and data processing (*DP*) tasks, assigning a file, or input stream, to be read by each *DS* thread. This core scheduling layer ensures that both compute- and memory-bound codes are efficiently executed.*DS* and *DP* tasks should finish close to each other, balancing the amount of threads assigned for each type of task to minimise the overall execution time.

The following heuristic leverages the amount of threads for the *DS* tasks, addressed as *DSt*, and for the *DP* tasks, *DPt*:

- Create one *DS* and one *DP* thread per physical multicore PU core, but only one thread will be active at any time. The unnecessary *DS* or *DP* threads are put asleep so that they do not increase the scheduler overhead, and are only switched on when needed. Preliminary tests showed that using a separate thread for *DS* and *DP* in the same core did not increase the overhead and was simpler for the scheduler to manage over using a single thread that switches between both tasks.

- Activate the same amount of threads for *DSt* and *DPt* as the initial default configuration.

- Measure the execution time of a *DSt* (DSt_t) and a *DPt* (DPt_t) for the same chunk of dataset elements in the file, with a pre-defined size.

- Compute the time impact of each thread by periodically dividing the chunk time by the amount of threads used, for both data setup (DSt_t) and processing (DPt_t) execution times. If the *DP* takes much longer than the *DS*, the scheduler allocates more threads for the *DP* tasks, and vice-versa. If both *DS* and *DP* execution times are similar, use an intermediate configuration.

- Dynamically tune the number of threads for each task, ensuring only one active thread per compute core, which is performed at pre-defined checkpoints.

- Periodically compute how many threads (n) should be shifted from setup to processing and vice-versa, according to equations 1 (if $DS_t \leq DP_t$) and 2 (if $DP_t < DS_t$).

$$DS_t - n * DSt_t = \frac{DS_t + DP_t}{2} \tag{3.1}$$

$$DP_t - n * DPt_t = \frac{DS_t + DP_t}{2} \tag{3.2}$$

- Double the gap between checkpoints if n does not change in two consecutive checkpoints to reduce the scheduler overhead. Afterwards the scheduler tunes the threads configuration at a double rate, if n changes in two consecutive checkpoints, to deal with the dataset irregularity.

- Allocate all threads to *DP* once the *DS* is complete.

Compute-bound code benefits from having more *DP* threads active, while still simultaneously reading the input data at a lower rate. The scheduler assigns the largest amount of *DP* threads possible where the application does not wait for data to be loaded.

Memory-bound code performance is significantly improved if more resources are assigned to *DS*, when input data can loaded in simultaneous chunks. Using a large amount of *DS* threads allows for data to be loaded at a higher rate, while ensuring that the *DP* threads do not have to wait for data to be loaded. This approach may also improve IO-bound code, but it was not tested yet.

Preliminary experimental results showed that there are no significant benefits of using simultaneous multithreading (Intel Hyper-Threading) in both compute- and memory-bound codes, using the case studies later presented in Section 4.1. However, this may not apply to every pipelined data stream application.

The current memory monitoring system in *HEP-Frame* only frees the whole data structure when all data is processed by the pipeline. This will be later improved to continuously free memory as soon as the dataset elements are consumed by the pipeline. When there is no data in the data structure the *DP* threads are put asleep.

3.2.4 Pipeline Ordering and Parallel Execution

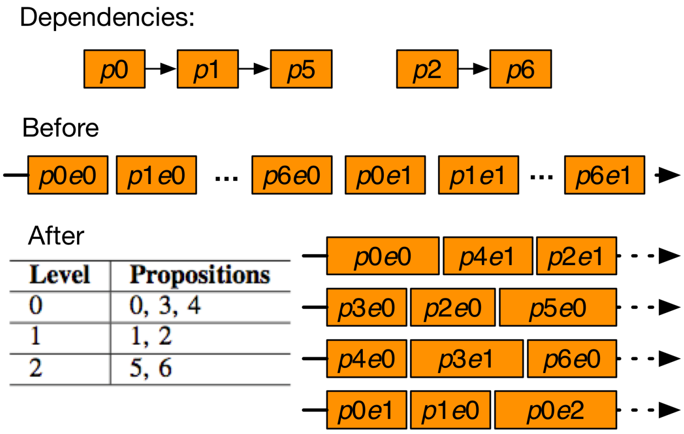
The initial order of the propositions in the pipeline is defined by the order that each is added in the code by the user, which may not be the most computationally efficient. Reordering the propositions often leads to a faster execution of the pipeline, while respecting the dependencies among propositions to ensure the correctness of the results. If the propositions that discard more data elements are placed earlier in the pipeline, and the heavier propositions in later stages, fewer data will be processed by the heavier propositions, which reduces the overall execution time of the pipeline.

The execution flow of the propositions in pipelined data stream applications can only be defined by a directed cyclic graph, unlike the applications that most list scheduling algorithms tackle. In the absence of dependencies, $prop_0$ and $prop_1$ can be executed in any order, which is represented by a bidirectional edge between the nodes that represent these propositions, which allows cycles to be present on the graph. More computational power is required to find the best pipeline order on cyclic than on acyclic graphs.

The *HEP-Frame* scheduler implements a simple approach to deal with cyclic graphs, as there is no need to follow a strict order of the propositions since propositions with no dependencies can be executed in parallel for the same dataset element. The scheduler still creates a graph, where a directed edge between two nodes (propositions) represents a dependency, which is previously defined by the user. For instance, if $prop_0$ needs to be executed before $prop_1$ there is only a directional edge from $prop_0$ to $prop_1$. A traditional list scheduler would compute the path with the lowest cost that passes through all nodes in the graph, based on an arbitrary weight given to each edge connecting two propositions. This path would later be used as the order of the propositions in the pipeline. The drawback of this approach is the unnecessary overhead of computing all possible paths and sorting them by weight, in every pipeline reordering checkpoint, as finding the directional Hamiltonian path is a NP-Complete problem.

Th scheduler uses the Breadth-First Search (BFS) algorithm [68], once during the application initialisation, to compute a list of all paths in the graph with directional edges that correspond to a list of all dependencies among propositions. BFS has a complexity of $O(|E| * |N|)$, where E and N are the amount of edges and nodes, respectively. Other algorithms, such as Depth-First Search [69] or the more computational efficient Linear Breadth-First Search [70], could be used if BFS had a significant overhead. However, tests later presented in Chapter 4 showed that the overall overhead of the scheduler, which includes the BFS computations among other operations, is negligible.

A table is built with n rows, where n is the amount of propositions in the longest dependency chain, and all propositions are inserted into a given row according to their position in their respective dependency chain. If there are no dependencies, a table is built with 5 rows, which preliminary tests have shown to be adequate, to ensure that some degree of pipeline reordering can be applied. Propositions with no dependencies are inserted in the first row of the table. Figure 3.4 illustrates the parallel execution of a pipeline with 7 propositions ($p0...p6$) on a 4-core server, for various dataset elements ($e0, e1...$), as well as a list of dependencies among propositions. The scheduler assigns propositions from a given table row to be processed by the available *DP* threads, but does not assign a proposition of the next row until all propositions on the current row are processed.



Level	Propositions
0	0, 3, 4
1	1, 2
2	5, 6

Figure 3.4: Sample pipeline execution with the *HEP-Frame* scheduler.

This heuristic ensures that all dependencies are respected since dependent propositions are placed on different rows. If a proposition criteria evaluation fails, or if there are more *DP* threads available than propositions left in the current row, the scheduler starts assigning propositions of the next dataset element.

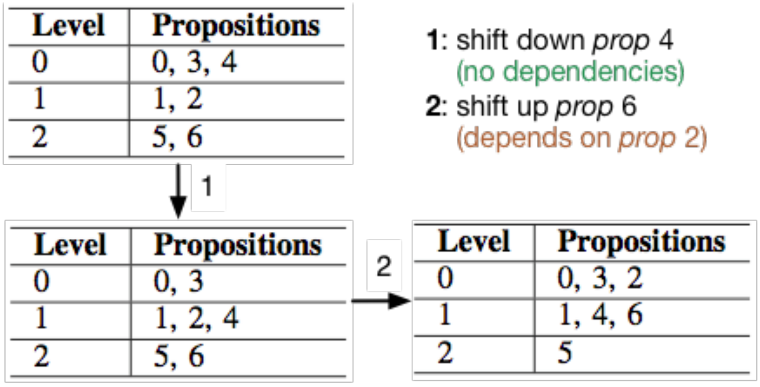
Propositions are ordered according to their weight within each row; those that weight less will be processed earlier. The weight of each proposition p_w is based on the ratio of discarded dataset elements (p_{dde}) with its normalised execution time (p_t), according to equation 3.3.

$$p_w = r_1 * p_{dde} + r_2 * p_t \tag{3.3}$$

The default values of r_1 are set to 70% and r_2 to 30% to achieve a weight between 0 and 1, which were obtained through extensive testing of the case studies presented in 4.1. This weight is only valid for each proposition on its position in the pipeline when the calculation is performed. Considering a weight for every order would require the scheduler to consider a different weight of each proposition depending on the order change it would want to test for the pipeline. This approach would result in a noticeable scheduler overhead for possibly minimal gains. Currently, the scheduler generalises the proposition weight regardless of the proposition position in the pipeline.

The range of weights for a row of a table is calculated by dividing the maximum possible weight, which is 1, by the amount of rows in a table. For instance, a table with four rows would have propositions with a weight of $]0, 0.25]$ in the first row, $]0.25, 0.50]$ on the second, and so on.

Propositions are be moved between table rows during the application execution: propositions with long execution times and that filter out fewer data are placed later in the pipeline. Propositions with no dependencies can be moved to another row introducing an artificial dependency. This allows lighter propositions, which filter out a larger amount of dataset elements, to be processed earlier so that less dataset elements are unnecessarily processed by heavier propositions. Propositions with dependencies are also moved if two requirements are met: subsequent propositions in its dependency chain can also be moved; the subsequent propositions in the chain cannot be moved further than the amount of rows in the table. This is performed efficiently as for each proposition on a dependency chain a list of its subsequent depending propositions is kept. Figure 3.5 shows how two propositions are moved to higher or lower rows in the table due to their increase or decrease in weight, respectively.



Level	Propositions
0	0, 3, 4
1	1, 2
2	5, 6

1: shift down *prop* 4
(no dependencies)
2: shift up *prop* 6
(depends on *prop* 2)

Level	Propositions
0	0, 3
1	1, 2, 4
2	5, 6

Level	Propositions
0	0, 3, 2
1	1, 4, 6
2	5

Figure 3.5: Proposition table update as the scheduler moves propositions 4 and 6.

The scheduler periodically calculates the weight for all propositions and assesses if they are in an adequate row for their weight. A signal is sent to all computing threads to finish executing propositions until all current dataset elements currently are processed and then are put asleep. The threads resume execution once the reordering is complete.

Figure 3.6 compares a typical list scheduler, which implements parallel proposition execution for each dataset element e, with the *HEP-Frame* scheduler for a case study with 4 independent propositions ($p0 \dots p3$) on a 4-core server. Proposition $p2$ filters out the dataset element eN (red box) and proposition $p3$ takes significantly longer to execute than the remaining propositions.

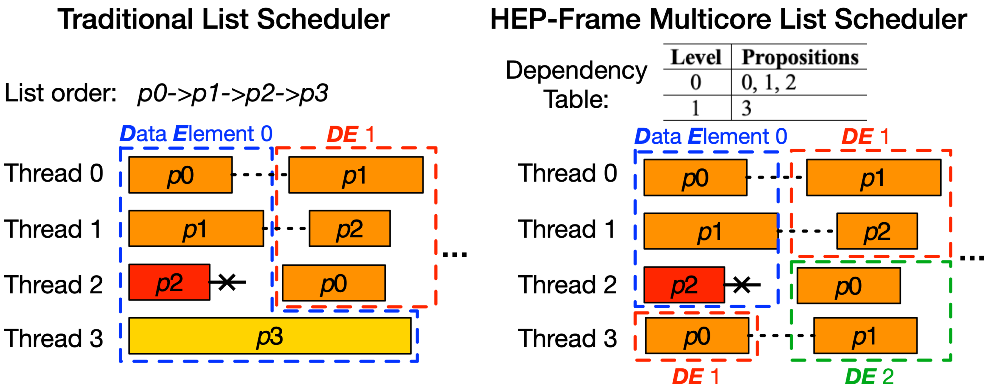


Figure 3.6: Typical list scheduler *vs. HEP-Frame* list scheduler for a pipeline of 4 propositions with no dependencies.

A traditional list scheduler would assign propositions $p0$ to $p3$ to a different thread in this 4-core server: $p3$ is executed (yellow box) wasting computational resources since its output is discarded for eN due to the failure of $p2$. The *HEP-Frame* scheduler introduced an artificial dependency based on the weight of $p3$, which forced $p3$ to be executed after all other propositions. Proposition $p0$ for the next dataset element was assigned to the last thread. This approach avoided unnecessary computations, as $p3$ was never executed for eN. This heuristic may provide significant performance improvements for propositions that take considerably longer to execute than others (sometimes by a factor of 10^6, as is the case in the applications presented in Section 4.1).

A more sophisticated list scheduler could consider that a task is the combination of a proposition and a given dataset element. Based on the weights, this scheduler could schedule $p3$ of several e to execute much later than the $p0-2$ of those e, so that if $p0-2$ failed it could discard $p3$ before its execution. However, separating the execution of $p0-3$ of a given e

by a significant amount of time leads to a poorer usage of the data locality, since if $p3$ of a given e is executed much later than $p0-2$ the e data needs to be reloaded into the cache, causing unnecessary cache misses. The *HEP-Frame* scheduler takes advantage of the spatial and temporal locality of the data, as for a given e it schedules $p3$ to execute as soon as $p0-2$ finish.

Traditional list schedulers are extremely efficient to manage pipelines of tasks that do not filter out dataset elements, but are not designed to schedule this type of propositions. This novel strategy of simultaneously processing propositions of the same and different dataset elements, while reordering the pipeline and respecting proposition dependencies, reduces unnecessary computational load and leads to a faster adaptation to irregular workloads.

3.3 Using Accelerator Devices

Using accelerator devices can improve the performance of compute-bound code, and usually follows two alternative approaches: (i) to offload a significant portion of the code to the coprocessor, ideally processing it simultaneously with the host devices when available, or (ii) to offload specific sections of code more suitable for the device characteristics, which may account for a significant portion of the overall execution time of the original application. *HEP-Frame* can use both offload strategies: it is currently compatible with Intel Knights Corner (KNC) coprocessor, Intel Knights Landing (KNL) manycore server and NVidia GPU accelerators.

The KNC device poses some limitations to the user. Firstly, it requires to explicitly transfer the required data for the propositions from the multicore memory to the coprocessor memory, forcing the user to use simple data structures. Developing the code to transfer complex data structures, based on containers, classes and pointers, may require a complete redesign of all data structures in an application. This limitation is also present on GPU devices. Secondly, the libraries required by the propositions must be compiled specifically for the device, which is often not feasible due to several compatibility issues of the architecture and/or compilers.

GPUs use a different programming paradigm than regular multicore code, which adds to the same limitations present in the KNC device. Most libraries are not available in CUDA or OpenCL, and porting them into the accelerator may not be possible for most cases. Therefore, it may be unfeasible to use the offload approach (i), since most propositions use functions from libraries that were not ported for GPU devices.

HEP-Frame can take advantage of KNC and GPU devices by providing an API with efficient implementations of frequently used code for scientific applications. Currently, Pseudo Random Number Generation (PRNG) functions are supported, but more functions can be added to the API based on user feedback.

HEP-Frame is also able to process the proposition pipeline on manycore KNL servers, both as a single server and simultaneously with other multicore/manycore servers. It uses a variation of the list scheduler for multicore devices, as seen in Subsection 3.2.4. An *HEP-Frame* prototype was developed to assess the feasibility of offloading PRNG to KNL servers, following the offload approach (ii), but preliminary tests showed that it provides a smaller performance improvement than processing the whole pipeline in the server.

3.3.1 Offloading Propositions into the KNC Coprocessor

A *HEP-Frame* proof-of-concept prototype was developed to assess the feasibility of using KNC devices using approach (i). Offloading propositions requires the user to develop the code to transfer required data to and from the coprocessor, and to adapt the proposition code. A layer of the scheduler balances the datasets among host multicore and KNC devices, using a demand-driven approach, where the KNC requests fixed-size data chunks to process before becoming idle.

The scheduler was adapted to use the KNC in offload mode with some minor modifications: the pipeline tasks are executed in serial mode, while the datasets are processed in parallel, one per active thread (Figure 3.7) and fully exploring the multithreading capabilities of this device. The filtering ratio and execution time of each task on the KNC pipeline are measured and transferred to the host device with each new data chunk request, so that the scheduler may compute a new pipeline order specifically for the KNC.

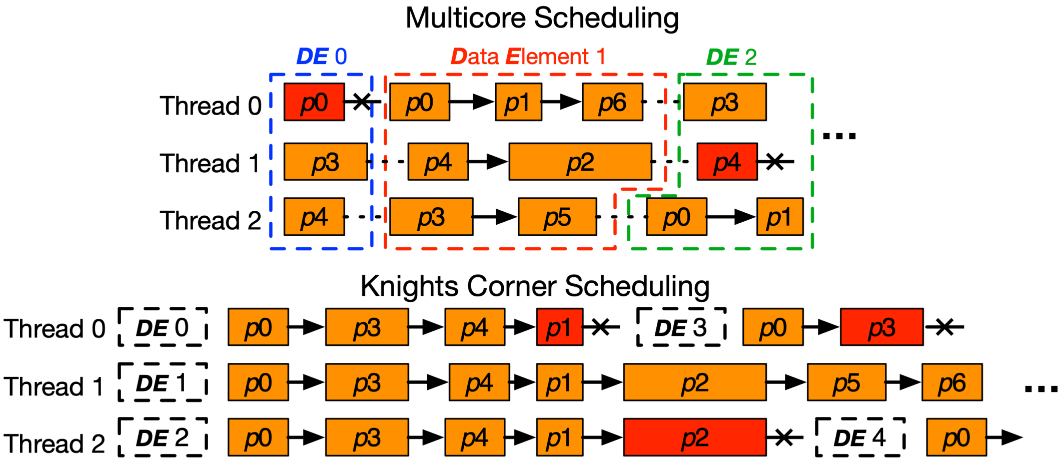


Figure 3.7: Comparison of the scheduling of a pipeline of propositions in the multicore and KNC devices.

The scheduler creates a single pipeline order based on the propositions placement in the dependency table, which is processed by each thread on different dataset elements. Preliminary tests showed that using 4 threads per core on the KNC provided the best performance. A thread is created in the multicore device to exclusively handle the communications between multicore and KNC and to calculate the KNC pipeline order. The data chunk to be delivered to the KNC needs to be large enough to efficiently use the 4 threads at each core, while minimising data transfers, but not so large that the memory transfer significantly degrades performance. Preliminary tests showed that using 10 to 50 dataset elements per hardware thread was a reasonable heuristic, using the case studies presented in Section 4.1, since execution times of such chunk sizes did not vary more than 5% (this is described in depth in [71]).

One major limitation of this approach is that the code of the offloaded propositions needs to be compatible with the KNC, which often is not the case since most pipelined data stream applications relies on complex libraries that cannot be easily ported to work simultaneously on multicore and KNC devices. For instance, the case studies used to evaluate the performance of *HEP-Frame* had to be heavily modified, from the data structures to the pipeline, to account for the explicit data transfers and different parallelisation strategies used in the KNC. Also, scientific libraries had to be compiled specifically for the KNC, which proved to be a laborious task.

HEP-Frame does not provide support for this device in its public version, since most users cannot dedicate the time required to develop efficient code for KNC devices, while only having the potential to provide a small performance improvement. It only improved the performance of a compute-bound code up to 33% using two KNC devices (this performance evaluation is later detailed in Section 4.3.4).

3.3.2 Offloading PRNGs to Multicore, Manycore and Accelerator Devices

The current version of *HEP-Frame* implements several Pseudo-random Number Generators (PRNGs) and distributions on its API (available in MKL [37], ROOT [72] and PCG [63]). Some that can be offloaded to NVidia GPUs, using the CUDA cuRAND library [67], while the rest can be offloaded to other multicore, KNL and KNC devices. A request for a new PRN in a parallel multithreaded environment usually follows one of these approaches:

- A single PRNG to feed all concurrent threads, where each PRNG execution is atomic. Results would not be reproducible as Pseudo-Random Numbers (PRNs) consumed by each thread varies between runs [73]. It does not support concurrent execution of the PRNG.

- A single PRNG to feed each stream request using a transition function to guarantee that there are no correlations among streams, known as leapfrog. Used in the cuRAND implementation of Mersenne Twister [74]. It supports concurrent execution.

- A single PRNG to feed all concurrent threads with a different pre-computed seed for each stream, causing the generated PRNs to be equally spaced in the overall PRN sequence, which may be slow [75], known as splitting. It supports concurrent execution.

- An independent PRNG per compute thread initialised with different sets of parameters. If these parameters are not adequate, streams may not be truly independent, as referenced in [76]. The most common and portable approach.

An implementation of a PRNG is as important as the approach used to interact with the PRNG itself in the code to the overall performance of an application. For instance, one can generate all PRNs upfront, or request a PRN when needed, which will have a different performance impact depending on the application and hardware environment. Using an adequate approach may have an increased impact on the performance of parallelised code, where there may be multiple threads accessing shared PRNs and/or PRNGs.

HEP-Frame provides three API alternatives to use PRNGs in parallel environments:

- To call the PRNG whenever a single PRN is needed.
- To generate a batch of PRNs and store the result in a thread private buffer. When a PRN is needed the compute thread removes it from the buffer. When the buffer is empty, a new batch is requested.
- To generate a batch of PRNs and store the result in a thread private dual-buffer. While the one buffer is being consumed, the other is being filled.

The efficiency of each strategy will depend on the computational server and the characteristics of the application. Preliminary results showed that sharing buffers among compute threads degrades performance due to contention when accessing shared resources. Therefore, the proposed buffer implementations are thread-private.

The dual-buffer approach minimises the overhead of the PRNGs execution. Figure 3.8 illustrates this approach. This approach attempts to hide the costs of memory transfers over PCI-Express interface and ethernet, when the PRNG is offloaded to a GPU and to another server over MPI. Using an alternative computing device exclusively for PRN generation, such as a separate server, GPU, or manycore device, frees computing resources on the host server that can be used to process other sections of an application.

Single PRN generation were not implemented for multiple nodes connected by ethernet and manycore devices connected by PCI-Express, since the lack of parallelism and overhead of memory transfers would significantly degrade the application performance.

As seen in Figure 3.8, the single- and dual-buffer implementations resort to a management thread for each *DP* thread on the host device. The management thread is responsible

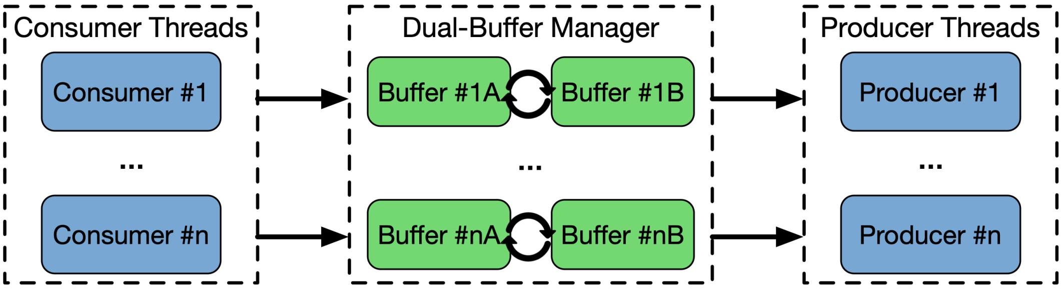


Figure 3.8: Dual buffer implementation in the PRNG management threads.

to allocate the buffers on memory, interact with the PRNGs on the devices using specific libraries and handle data transfers. The overhead of the management threads is minimal as they are only used to fill the buffers and are asleep the rest of the application run-time. On the dual buffer approach, a buffer is filled as soon as the it is depleted.

The memory bank in which the PRN buffers are allocated may have a significant impact on performance on NUMA servers. If a *DP* thread has to access a buffer allocated on the memory bank of another multicore device it will have an increased penalty over accessing a memory bank of the device it is on. To mitigate this problem, management threads are bound to the core of its assigned *DP* thread, ensuring the buffers are allocated in the memory bank closer to each *DP* thread.

In the parallel implementations using the NVidia GPU, each PRNG management thread in the multicore devices uses a different stream to the GPU to launch kernels and perform the memory transfers, ensuring that concurrent management threads can simultaneously generate and receive PRNs. Tests showed that for up to 32 computing threads (and associated management threads) the NVidia K20 GPU device was not fully utilised in the case studies used (later discussed in Chapter 4 for a description of the GPU and case studies), meaning that it could scale for a greater number of multicore threads. This behaviour may vary with different applications, according to their PRN demand.

The MKL library was used to generate PRNs on the KNL manycore server (over an Ethernet link) and the KNC manycore coprocessor (over PCI-Express), as it provides optimised implementations for these devices. Preliminary tests showed that the PCG RXS-M-XS PRNG did not perform as well as MKL, as its code does not take advantage of the 512-bit vector instructions available in these devices.

Evaluation of Sequential PRNG Performance on Multicore Devices

MKL offers a wide range of implementations of various Gaussian transformations to generate a single PRN or batches of PRNs using the Mersenne Twister algorithm. To offer all these implementations on *HEP-Frame* API would unnecessarily increase its complexity and require the user to choose an adequate algorithm and implementation to ensure efficient execution of the application. *HEP-Frame* should only offer the fastest implementation of each PRNG algorithm and transformation for each supported computing device, to simplify the options provided to the users.

Figure 3.9 compares the sequential execution times to generate 10^6 PRNs of various implementations of the Mersenne Twister with the ROOT (a library for high energy physics, later detailed in Section 4.1) and MKL libraries, the former coupled with the Box-Muller (*ROOT-BM*) transformation and the latter with the two available Box-Muller implementations and the Inverse Transform Sampling (*MKL-BM1*, *MKL-BM2* and *MKL-ICDF*). MKL also offers implementations optimised to generate batches of PRNs (suffix *-A*) with a single function call. The PCG was coupled with the Box-Muller transformation(*PCG-BM*). A dual socket server with 12-core Intel Xeon E5-2695v2 Ivy Bridge devices, at 2.4 GHz with 64 GiB RAM was used for this test.

This test shows that there is a small difference between the execution time of ROOT and PCG generators, with 29 and 25 ms respectively. The MKL batch generator using the Inverse Transform Sampling only required 3 ms to generate these numbers, being the fastest of all tested generators. Single number generation using MKL with any transformation performed the worst, up to 12.2x slower than ROOT, since it has an increased overhead due to an internal PRNG management overhead that is unnecessarily performed for every number. The *MKL-ICDF-A* generator will be used with the dual-buffer approach as the default MKL generator on *HEP-Frame*, on both multicore, KNC and KNL devices.

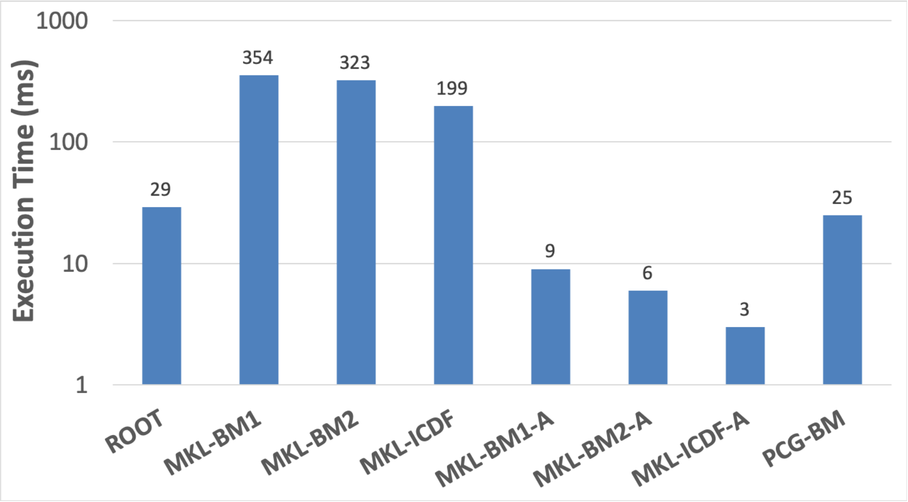


Figure 3.9: Sequential execution time of each PRNG implementation for multicore devices to generate 10^6 PRNs.

3.3.3 Pipeline Reordering and Parallelisation in the KNL Server

The scheduling algorithm presented in Subsection 3.2.4 was tweaked to explore the KNL massively parallel architecture. Pipelines with complex dependencies require scheduler task synchronisations that may limit performance when using large amounts of threads (up to 512 on this device) on a device with such a low clock rate (1.1 to 1.3 GHz) and high communication costs among different cores.

Each *DP* thread on the KNL server computes the whole pipeline for a dataset element, instead of a combination of a dataset element and a proposition. The execution time and the ratio between processed dataset elements and those filtered out is still measured for each proposition, as it will still contribute to the reordering of the pipeline. After processing a data chunk, an average of the normalised measured values of all threads for a given proposition is computed, which attributes a global weight to each proposition.

A directed cyclic graph is built, where each vertex represents a proposition and the edges to a given node have the weight of the respective proposition. A dependency where $p1$ must be executed after $p0$ is represented by an edge with an "infinite" weight on the edge from $p1$

to $p0$, ensuring that a path with "infinite" weight is never used. A list of all nodes that can be used to start a path is stored when building the graph, which greatly reduces the amount of paths that the algorithm to find the best path has to test.

The best pipeline order is obtained by computing the shortest path that passes through all vertices (propositions) of the graph using a recursive backtracking algorithm. The shortest path, i.e., the path with the least weight, is computed for each of the nodes that can be used as the beginning of a path on the graph. Of these paths only the shortest is considered, which is used as the best pipeline order at the moment. The backtracking algorithm, *findPath*, to determine the shortest path for a given starting node is shown in Figure 3.10. *findPath* is called for each proposition that can be used to start a path. It receives this proposition and discards it if it is already on the path (which is initialised empty) or if the edge connecting this proposition with the last on the path as an infinite weight. If not, the current proposition is added to the current path and calls itself recursively for each proposition that is neighbour to the last on the path. Once the path is complete, i.e., all propositions are on the path, *findPath* stores the current path as the best pipeline order if its overall weight is lower than the previously stored path. Preliminary tests showed that the overhead of this algorithm is less than 1% of the case studies, since it is only computed for a small set of starting nodes, and these graphs usually have a small number of overall nodes (18 for these case studies, see Section 4.1 for more information on these case studies).

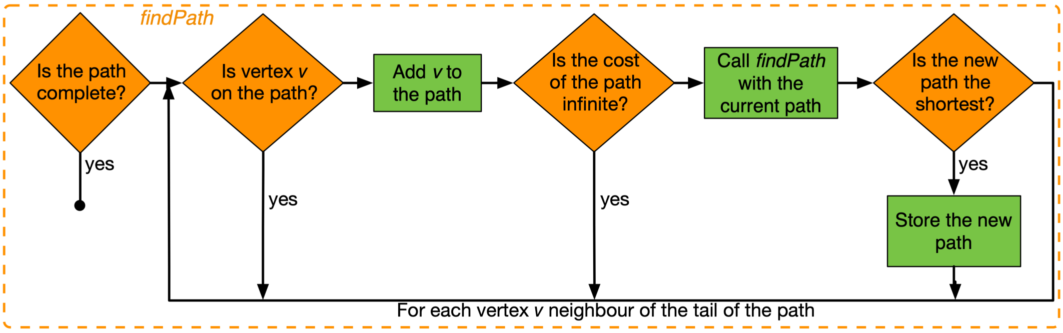

findPath
Is the path complete?
yes
Is vertex v on the path?
yes
Add v to the path
Is the cost of the path infinite?
yes
Call findPath with the current path
Is the new path the shortest?
yes
Store the new path
For each vertex v neighbour of the tail of the path

Figure 3.10: Scheduler pipeline reordering backtracking algorithm for the KNL server.

The pipeline is reordered at given checkpoints during the application execution, as shown in Figure 3.11. These checkpoints enforce a barrier to ensure that all threads do not process any dataset elements before computing a new pipeline order (green boxes in Figure 3.11). Once this order is calculated, all threads use it to process the next dataset elements until the next checkpoint.

Scheduling the whole pipeline better explores the vectorization capabilities of the KNL (it is easier to predict the instructions that each thread will execute), but is not ideal for highly irregular code, due to a coarser task grain size.

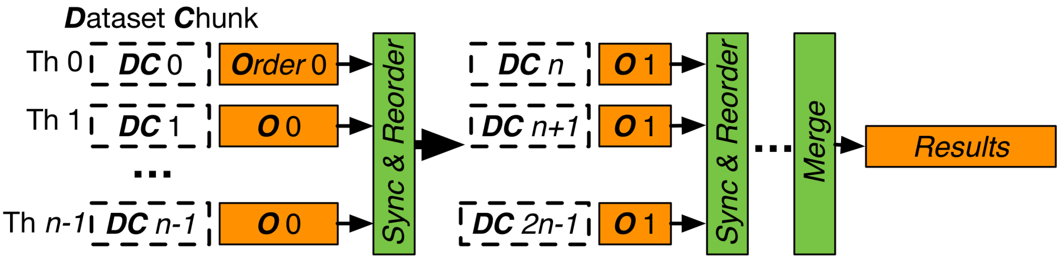


Figure 3.11: Parallel execution of the pipeline in the KNL server for n threads (*Th*).

3.4 Summary

HEP-Frame is a user-centred framework to aid the development of pipelined data stream applications that analyse data from a large number of streamed dataset elements. It provides a simple code development environment, which is especially important to improve the code development speed and robustness, as it provides a set of code skeletons and automatises repetitive tasks. The framework ensures efficient parallel execution of pipelined data stream applications portable across homogeneous multicore and manycore servers, and heterogeneous servers coupled with manycore and GPU accelerator devices, transparently to the user.

HEP-Frame structures the application code into input file reading, pipeline definition using propositions and output storage. As a case study, it currently provides tools to automatise the code creation for input file reading and output storage for high energy physics applications, reducing the amount of code that an user has to develop. Other tools can be developed by users and integrated in the compilation process to automatise any part of the code development in the provided application skeletons.

HEP-Frame implements a multi-layer scheduler for efficient execution of pipelined data stream applications for homogeneous and heterogeneous servers. This scheduler adapts to the computational characteristics of the server and application at run-time, and is designed to process pipeline propositions and various dataset elements in parallel, distributing them across the available computational resources. It specialises in balancing irregular workloads for I/O, memory- and compute-bound applications without prior knowledge of the application and compute server.

The first scheduler layer parallelises the execution of the application input file reading, pipeline processing and output storage for a pool of input files with multiple processes. It balances the input files among the processes using a demand-driven approach, allowing *HEP-Frame* to scale with multiple compute servers.

The second scheduler layer parallelises the execution of the input file reading and data structure creating (data setup, *DS*) with the pipeline processing (data processing, *DP*), using multiple threads. It manages the amount of threads assigned for *DS* and *DP* tasks at run-time, according to each application requirements. This strategy allows *HEP-Frame* to adapt to I/O-bound code, by assigning more threads for simultaneous execution of *DS* tasks, compute-bound code, by assigning more threads to *DP* tasks, and to memory-bound code, by adopting an intermediate distribution of threads.

The third scheduler layer focus on the efficient parallel processing of pipeline propositions of the same or different dataset elements (*DP* task). This layer implements a strategy for proposition reordering and parallel execution in the pipeline, to ensure that faster propositions that filter out more data are executed before the more compute intensive propositions. This strategy ensures that less dataset elements reach the heavier propositions, reducing the overall execution time of the applications.

The proposition execution is performed in Intel Xeon Phi manycore devices, using a variation of the third scheduler layer. The whole pipeline is executed by each thread on different dataset elements, eliminating the parallel execution of propositions of the same dataset element. This induces a coarser grain for the scheduler workload balancing, which may affect highly irregular applications, but reduces the high overhead that the original layer has on these specific devices.

Finally, *HEP-Frame* provides a wide range of efficient pseudo-random number generators that can be executed on the compute server, offloaded to other multicore and manycore compute servers, and offloaded to manycore and GPU accelerator devices. The scheduler provides a dual-buffer management system that transparently generates a batch of pseudo-random numbers on these devices, while a second batch is consumed by the application, with a negligible overhead. This strategy hides the time penalties of transferring data from other servers or accelerator devices, while freeing compute cores of the main server to process other sections of the application.

Chapter 4

HEP-Frame Performance Evaluation

This chapter presents a quantitative performance evaluation of the HEP-Frame key features on various parallel environments. This evaluation is performed using as case studies three real world scientific data analyses, which represent pipelined data stream applications with different computational characteristics. HEP-Frame is evaluated using various single- and dual-socket multicore and manycore servers, some coupled with manycore and GPU accelerators, representing different device micro-architectures relevant in desktop and mini-cluster environments.

The performance of the layers of the HEP-Frame scheduler is individually assessed, presenting and discussing the results for the three case studies on various servers. An evaluation of different strategies for PRNG available in HEP-Frame, a key component in most pipelined data stream applications, is also presented and discussed. Finally, HEP-Frame is compared against StarPU, a direct competitor designed for a wider range of applications.

4.1 Case Studies: $t\bar{t}H$ Scientific Data Analyses

Pipelined data stream applications should have a well-defined set of characteristics to be used as case studies to evaluate the performance of every *HEP-Frame* feature:

- Be representative of one or more classes of code (I/O-, memory-, or compute-bound).
- Be real applications developed by computer or non-computer scientists and actively used in a production environment.
- Contain the key features of a pipelined data stream application as described earlier, namely process a very large set of mini-batch or streaming *n-tuple* input data, and over at least 10 processing pipeline stages. Some propositions should be commutative, filter out dataset elements, and/or perform heavy and irregular computational tasks.
- Its behaviour should vary with small modifications to the code or to the input data: either more compute-bound or more memory-bound. This allows to assess the impact of the memory access and computation bottlenecks, while ensuring that it is not caused by changes in the application code.

Scientific data analyses, a subset of pipelined data stream applications, are frequently used by high energy physics scientists at CERN to study the building blocks of the universe. Proton beams are accelerated in opposite directions close to the speed of light, at the Large Hadron Collider (LHC), and collide at the core of specific particle detectors, such as the one used by the ATLAS Experiment [77]. The particles that decay from this head-on collision interact with different sub-detectors, which measure their energy, momentum, and position. A particle collision, and associated decaying particles, is known as an event.

Scientific data analysis applications in high energy physics usually process intermediate size data formats, rather than the information directly measured by the particle detectors. These data formats are developed at some stage of the overall analysis of an event (from measuring the event in the detectors to the obtaining the final physics results). In the initial stages of this analysis, the full output data format, which uses information directly from the output detector reconstruction (known as *xAOD* in ATLAS), comprises very large sets of data with an accumulated size of the order of the petabytes. From these large datasets, an intermediate sized data format is produced (of the order of the terabytes), from which physicists can obtain the final n-tuples (using the ROOT file format, with the `.root` extension) that is of the order of the megabytes to few gigabytes. Most scientific data analysis are expected to use a filtered *xAOD* file, usually stored in a ROOT format, but users can have access to the original *xAOD* files if necessary. ATLAS has collected more than 170 petabytes of data.

The selection of an adequate case study fell on a scientific data analysis code developed by high energy physics scientists working at the ATLAS Experiment that operates on this last stage: the $t\bar{t}H$ analysis. The goal of this code is to study the production of top quarks associated to a Higgs boson [78], in the dileptonic channel. Figure 4.1 represents the final state topology of a proton beam collision for the $t\bar{t}H$ production with an associated Higgs boson. One top and anti-top quarks and one Higgs boson are produced following a proton-proton collision. The top quarks are expected to decay through the main decay channel, i.e., $t(\bar{t}) \rightarrow bw^+(\bar{b}w^-)$, and the Higgs boson to $H \rightarrow b\bar{b}$. The b quarks are detected as jets of particles due to a physics process known as hadronization. The $w^+(w^-)$ bosons are expected to decay to leptons, i.e., $w^+(w^-) \rightarrow l^+\nu_l(l^-\bar{\nu}_l)$. The outcome of an event is then recorded by the ATLAS detector, which measures the properties of b quarks and leptons (both muons and electrons). Neutrinos are not recorded since they do not interact with the detector, but can be later analytically reconstructed by the $t\bar{t}H$ analysis, through a process addressed as kinematic reconstruction.

4.1.1 The $t\bar{t}H$ Analysis Code

The $t\bar{t}H$ analysis code is written in C++ and has 18 propositions that are organised in a sequential pipeline. Each proposition has a variable duration for its computational task (from few microseconds to several milliseconds per event) and a test to filter out measured events that do not comply with the theoretical $t\bar{t}H$ model. The kinematic fit aims to reconstruct the undetected neutrinos by assuming that originate from the w boson decays, which in turn decay from top quarks. This is achieved by imposing that the energy/momentum of the neu-

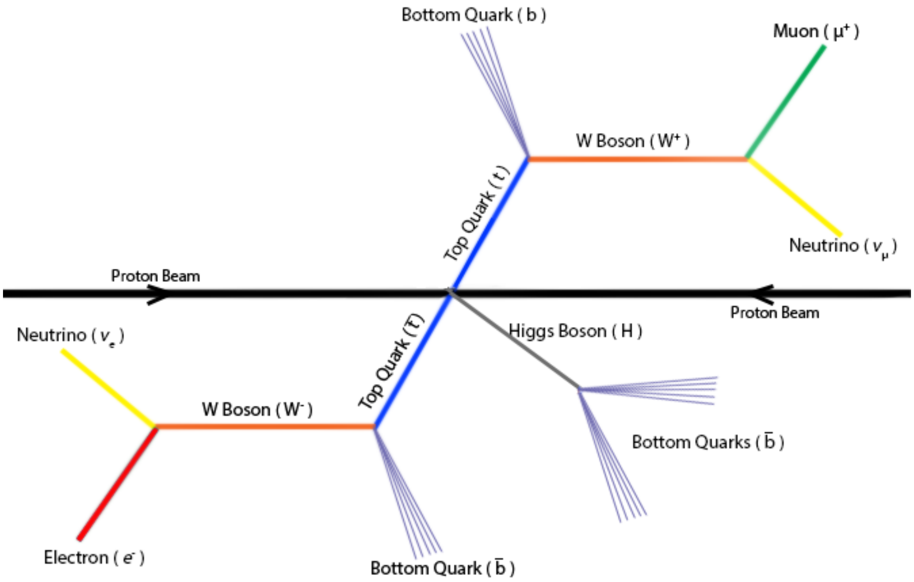


Figure 4.1: Schematic representation of the $t\bar{t}$ production with an associated Higgs boson, in the dileptonic physics channel.

trinos cannot exceed, together with the charged leptons, the energy associated with their respective w boson. In turn, the two w bosons are used to reconstruct the energy/momentum of top quarks, assuming the mass of the reconstructed top quarks to be 172,5 GeV. As all measurements are subject to resolution effects from the detectors used to identify the particles, it is very likely that the solution obtained directly from the measurements is not accurate due to uncorrected energy loss. Testing several solutions to allow the energy momentum to change within a specific range is crucial to achieve an accurate reconstruction of the event.

The last proposition implements the kinematical reconstruction process, which can be sampled multiple times per event, where the measured data is varied within an ±1% limit on the energy/momentum of the neutrino according to the Gaussian distribution. This reduces the relative measurement uncertainty of the detector to improve the precision and accuracy of the neutrino reconstructions. Only the solution that best fits the data is considered for each event that reaches this proposition. This process has a direct impact on the amount of computations performed per event.

The default organisation of the pipelines in the following analyses was setup by the scientist that developed the code and it was already optimised under his point of view: the heavier proposition is the last pipeline stage, while previous stages filter out a significant percentage of events. The dependencies among propositions in the $t\bar{t}H$ analysis are represented in Figure 4.2. The propositions inside the blue boxes do not have dependencies among them but depend, as a group, on other propositions.

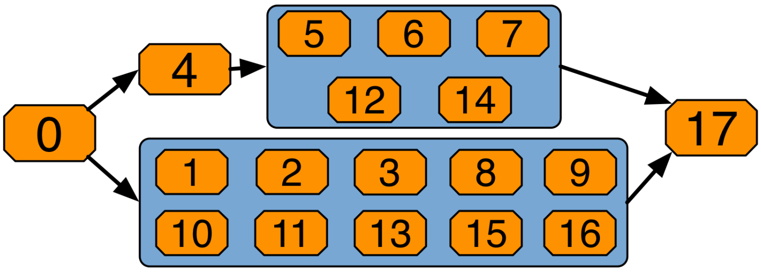


Figure 4.2: Schematic representation of the proposition dependencies in $t\bar{t}H$ analysis.

Three versions of the $t\bar{t}H$ analysis were considered as representative case studies:

`ttH_as` (***accurate detector system***)**:** the data measured by the ATLAS detector is considered 100% accurate when reconstructing the event. This behaves as a latency-bound code in most computing systems.

`ttH_sci` (***detector system with a confidence interval***)**:** considers an error in the accuracy of the ATLAS detector measurements up to ±1% and performs an extensive sampling within the 99% confidence interval in the kinematic reconstruction, where only the best reconstruction is considered. This version performs 1024 samples, where each requires the generation of 30 different PRNs, to a total of 30 Ki numbers per event, leading to a compute-bound code.

`ttH_scinp` (`sci` ***with a new pipeline***)**:** two propositions were replaced to perform different operations on the data element, maintaining the same overall proposition dependencies and only 128 samples within the same confidence interval of `ttH_sci`. This version is also compute-bound, but is less compute intensive than `ttH_sci`.

A preliminary test of the $t\bar{t}H$ analyses was performed on a dual-socket Ivy Bridge server with the dataset used for the performance evaluation to measure the execution time and filtering ratios of the propositions (the server is later described in Section 4.2). These results are detailed in Tables 4.1 and 4.2, respectively. The 18 `ttH_as` propositions have execution times always shorter than 13 microseconds, of which 16 pass more than 90% of the events. Two propositions have a passing ratio of 63% and 50%, respectively. The `ttH_sci` propositions have the same filtering ratios, since they share the same pipeline flow as `ttH_as`, but propositions 17 and 13 are heavier with an execution time of 29 and 5 milliseconds, respectively. The new proposition 13 in `ttH_scinp` has a longer execution time than in `ttH_sci`, around 49 milliseconds, and proposition 16 has now a passing ratio of 30%, versus 99% in `ttH_sci`.

Table 4.1: Execution time of the 18 propositions in the $t\bar{t}H$ analyses.

	Execution Time (nanoseconds)			
	$]0, 10^2]$	$]10^2, 10^4]$	$]10^4, 10^6]$	$]10^6, 10^8]$
`ttH_as`	2	15	1	0
`ttH_sci`	2	13	1	2
`ttH_scinp`	2	13	1	2

Table 4.2: Filtering ratios of the 18 propositions in the $t\bar{t}H$ analyses.

	Passing ratios (% of passing dataset elements)				
	[0%, 20%]	]20%, 40%]	]40%, 60%]	]60%, 80%]	]80%, 100%]
`ttH_as`	0	0	1	1	16
`ttH_sci`	0	0	1	1	16
`ttH_scinp`	1	0	1	1	15

These analyses use a wide set of features from the ROOT framework [72], which is a tool developed at CERN and widely used by the high energy physics community. It provides library functions for I/O of ROOT files, physics, statistical analysis tools, pseudo-random number generators and even application skeleton generators. Since most propositions of the $t\bar{t}H$ analyses depend on ROOT functionalities that are not implemented in any CUDA library, and cannot be easily ported, the pipeline cannot be executed in GPU devices.

4.1.2 Simple Parallelisation

The three original $t\bar{t}H$ analyses are implemented sequentially. The two parallelisation approaches already described in Section 2.3.3, based on multiple threads and multiple processes, were implemented into these analyses so that *HEP-Frame* can be compared with with these analyses using the same amount of computing resources. Both parallelisations should be designed to receive a large set of ROOT files, as these analyses are used to process several files usually not larger than 2 GiB each. A set of event information per proposition and all the data of the events that pass all propositions in the pipeline should be saved into two separate files.

The multithreaded parallelisation requires the initial data setup (input file reading, data decompression and data structure creation on memory) and output file writing to be performed sequentially, as ROOT file reading is not thread-safe. The pipeline processing can be performed simultaneously by several threads on different events, where each thread runs a sequential version of the full pipeline. Threads store event information per proposition on thread private data structures, which are merged before being written to an output file. This parallelisation is implemented using OpenMP [40]. The OpenMP *dynamic* scheduler was used to adapt the workload distribution according to the irregularity of the pipeline execution. This whole process is repeated for each input file.

The multiprocess parallelisation is used by non-computer scientists, where multiple instances of the analyses are executed with different sets of input files. Each instance of the analyses outputs two files, as previously described, which are later merged with the outputs of the remaining instances. An alternative approach is to launch a single instance of an analysis, which then creates a set of processes and divides the workload accordingly using a Message Passing Interface (MPI). The path to each input file is stored in a pool that is accessed by each process, which sequentially performs the data setup and processing, while partial and final event information is merged at the end of the processes execution. This approach was implemented using OpenMPI [20], but preliminary tests did not show a significant difference in performance over the multi-instance parallelisation and was discarded.

While the multiprocess approach ensures that there is simultaneous data setup and processing of the events, the multithreading can only perform parallel processing of different events.

4.1.3 Porting $t\bar{t}H$ Analyses into *HEP-Frame* and StarPU

The $t\bar{t}H$ analyses were ported into *HEP-Frame* by responsible scientist in just four hours, after a 30 minute crash course on the basic interaction with the framework, without requiring substantial changes to the original code. A code skeleton and data structures were automatically created by the *HEP-Frame* plugins based on an input ROOT file. If no such plugin was available, the user would also have to declare the variables of an event in a C++ class file, and provide the code to read the input files. The original analyses each proposition accesses the event variables stored in global memory. Since *HEP-Frame* provides an abstraction to access the dataset information in its data structure as if it is on global memory (as seen in Section 3.1), the propositions did not require any modifications. The code of all propositions was originally in a single function and had to be separated into individual functions.

The implementation of the $t\bar{t}H$ analyses in StarPU required major modifications to the original code to fit the framework requirements, which had to be performed by an experienced computer scientist. A major restriction that StarPU poses is the use of the `smartPtr` as the container of the event data structure, which required a major reorganisation of the original event structuring. This implementation uses the same multicore MKL PRNG as provided by *HEP-Frame*. The standard Heterogeneous Earliest Finish Time strategy [79] (HEFT, *dm* scheduler in StarPU) was used to balance the workload among the available workers. It schedules the propositions based on their execution time history, which causes tasks to be reordered according to their computational performance. The *dmdas* scheduler is a variant of HEFT that requires the user to implement a strategy to properly assign weights to the tasks in StarPU, so that the scheduler can reorder them. However, it was not used as the user would have to measure injected code in every proposition to measure its filtering ratio and execution time and create a function to assign a weight to each proposition, which is out of the expertise of most non-computer scientists.

Most propositions in the $t\bar{t}H$ analyses depend on ROOT functions that cannot be properly ported to efficiently use both the Intel Knights Corner coprocessor and NVidia GPU accelerators. The propositions would also require major modifications to their algorithms and data structures to be executed on GPUs, which would be unfeasible for most non-computer scientists. Due to these limitations, the use of StarPU will be restricted to multicore devices.

4.1.4 Key Characteristics of the $t\bar{t}H$ Analyses

Scientific data analyses usually have very specific sets of properties and can be used as case studies to evaluate the *HEP-Frame* performance. They can:

- Represent multiple types of scientific workloads (I/O, memory- and compute-bound).
- Be developed by non-computer scientists and actively used in their research.
- Be structured and contain a set of computing characteristics as defined in Section 2.3.
- Ideally, some propositions should be commutative, filter out dataset elements and/or perform heavy and irregular computational tasks.

The $t\bar{t}H$ analysis code, a set of three applications used by high energy physicists at CERN, was chosen as case study representative of a wide set of pipelined data stream applications.

As the original code of $t\bar{t}H$ is sequential, two distinct parallel implementations were developed to provide a fair comparison with *HEP-Frame*. These implementations followed the two common parallelisation approaches in scientific computing, already described in Section 2.3.3. These analyses were also ported into StarPU, a direct *HEP-Frame* competitor, in order to evaluate the differences between the schedulers available in these specialised frameworks.

4.2 Testbed and Methodology

The three configurations of a $t\bar{t}H$ analysis (`ttH_as`, `ttH_sci` and `ttH_scinp`) were tested with 128 input data files, each with $6,000$ events (the dataset elements), with around 250 different data variables per event, corresponding to measurements associated to reconstructed particles (electrons, muons, jets, etc) from ATLAS.

Four different types of compute servers were selected for the quantitative evaluation of the *HEP-Frame* performance features:

- A dual-socket server with 12-core Intel Xeon E5-2695v2 Ivy Bridge devices (IB) @2.4 GHz nominal, with 64 GiB RAM, coupled with a NVidia Tesla K20 with 2496 CUDA cores and 5 GiB of GDDR5 memory and a 61-core Intel Xeon Phi 7120 @1.2 GHz (KNC, 4-way simultaneous multithreading), linked through PCI-Express.

- A single-socket server with 10-core Intel Xeon E5-2630v4 Broadwell device @2.2 GHz nominal (1.8 GHz nominal with AVX2), with 64 GiB RAM, coupled with one NVidia GTX 1070 with 1920 CUDA cores and 8 GiB of GDDR5 memory (Pascal architecture).

- A dual-socket server with 16-core Intel Xeon E5-2683v4 Broadwell devices (BW) @2.1 GHz nominal (1.7 GHz nominal with AVX2), with 256 GiB RAM.

- A dual-socket server with 24-core Intel Xeon Platinum 8160 Skylake devices @2.1 GHz nominal (1.4 GHz nominal with AVX-512), with 192 GiB RAM.

- A single-socket 64-core Intel Xeon Phi 7210 server @1.3 GHz nominal (1.1 GHz nominal with AVX-512, KNL architecture with 4-way simultaneous multithreading), with 16 GiB of eRAM, 192 GiB of RAM.

The code was compiled with the Intel 2018 compiler suite, the NVidia CUDA 8.0 toolkit and ROOT 5.34.34. All servers used the CentOS 6.3 operating system. A k-best measurement heuristic was used to ensure that the results can be replicated, with $k = 5$ with a 5% tolerance, a minimum/maximum of 15/25 measurements.

4.3 Results and Discussion

The key quantitative evaluations performed to assess *HEP-Frame* performance related features include:

- The dynamic tuning of *DS* and *DP* threads.

- The impact of *HEP-Frame* multiprocess and multithreaded scheduler on multicore servers.
- The use of accelerators: 1 or 2 KNC devices, 1 GPU Kepler.
- The performance of a KNL-based server with different configurations.
- A performance comparison of the KNL server *versus* multicore servers, with or without accelerators (including a GPU).
- The scalability in a cluster of KNL servers.
- The scalability across heterogeneous servers (mix of multicore and manycore).
- A comparative evaluation of the *HEP-Frame* with StarPU.

4.3.1 Dynamic Tuning of *DS* and *DP* Threads

The dynamic tuning of *DSt* and *DPt* on a 12+12 core Ivy Bridge server is compared against 3 fixed configurations of *DSt*-*DPt*: 1-23, 12-12 and 22-2 (Figure 4.3).

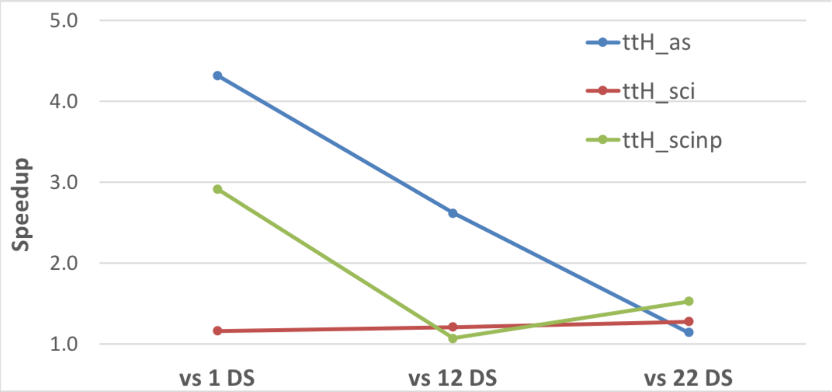


Figure 4.3: Speedup of dynamic *vs* static tuning of *DS* and *DP* threads on a dual-socket server.

The dynamic tuning outperforms all fixed configurations, with over 4x speedup for `ttH_as` with 1 *DSt*, as the efficiency of this code is limited by the *DSt*. The impact on `ttH_sci` is not so significant, since the computation is so complex that even with 1 *DS* thread the data setup is completed while only 20% of the dataset was processed.

The `ttH_as` converges to a stable *DS-DP* configuration after loading 5% of the dataset (converge to *22-2*), while the other two converge after only 2.5% of the dataset (to *2-22* and *6-18*, respectively). An analysis of the scheduler with the Intel VTune profiler [80] showed that, for the `ttH_as` latency-bound code, the *DP* threads were waiting for data to be loaded for less than 10% of the overall data setup time.

4.3.2 Multithreading with and Without *HEP-Frame*

To assess the adequate amount of threads to be used by *HEP-Frame* in a multi-socket server, a scalability evaluation of the framework was performed and is presented in Figure 4.4 using the $t\bar{t}H$ analyses, which indicative of applications with different computational requirements, in a 12+12 core Ivy Bridge server. *HEP-Frame* was tested using a single thread per physical core up to 24 cores, and 2 threads per core to use Intel 2-way simultaneous multithreading (Intel Hyper-Threading). Both compute-bound applications, `ttH_sci` and `ttH_scinp`, scale up to 24 cores with speedups up to 17.1x and 12.9x, respectively, while the memory-bound `ttH_as` reaches a peak speedup of 3x for 8 cores and 2.7x for 24 cores. As expected, the performance of the compute-bound applications increases almost linearly up to 8 cores, with a slight decrease in this rate of improvement from 12 to 24 cores, when a second multicore device is used and NUMA penalties apply.

The three $t\bar{t}H$ analyses did not benefit from simultaneous multithreading, with significant performance drops when using 48 threads over their highest speedup configuration. By default, *HEP-Frame* uses one thread per physical core of the available multicore devices, as it provided the best performance for `ttH_sci` and `ttH_scinp`, while having a small impact on the performance of `ttH_as` when compared to its best speedup (2.7x on 24 cores *vs* 3x on 8 cores). The maximum amount of threads can be changed by the user to fit the computational requirements of any specific application.

The performance of the multithreaded $t\bar{t}H$ analyses implemented with OpenMP was compared against their implementations in *HEP-Frame*, using one and two Xeon devices

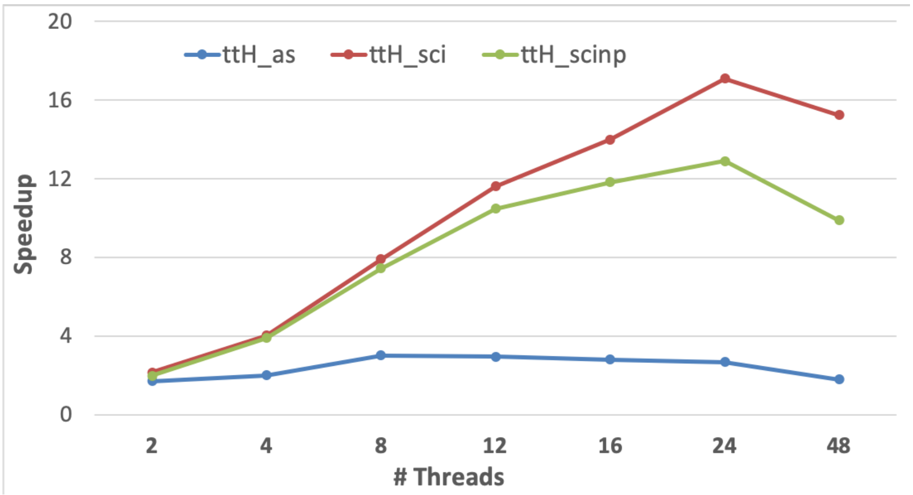


Figure 4.4: Scalability of the $t\bar{t}H$ analyses with *HEP-Frame* on a dual-socket Ivy Bridge server.

of the Ivy Bridge, Broadwell and Skylake micro-architectures (Figure 4.5). Both parallelisations use a single thread per physical core on the server, as preliminary tests showed that using Hyper-Threading did not provide significant performance improvements. *HEP-Frame* significantly improved the performance of all multithreaded implementations:

`ttH_as`**:** up to 6x, mostly due to simultaneous *DSt* and *DPt* management.

`ttH_sci` **and** `ttH_scinp`**:** 15x and 17x speedups, respectively, mostly due to the pipeline reordering and workload scheduler. *DSt* and *DPt* tuning does not have a big impact on performance, as both analyses require few threads for the *DS* tasks, which is similar to the OpenMP approach.

`ttH_scinp`**:** performance improvements mostly due to a worse initial pipeline order than `ttH_sci`.

The Skylake server has the best performance of all these servers mostly due to its higher core count, which is offset by its lower clock rate when compared to the Ivy Bridge server. Generally, the performance of using a second multicore device in the same server did not lead to a linear improvement due to the NUMA memory architecture. The data structures are

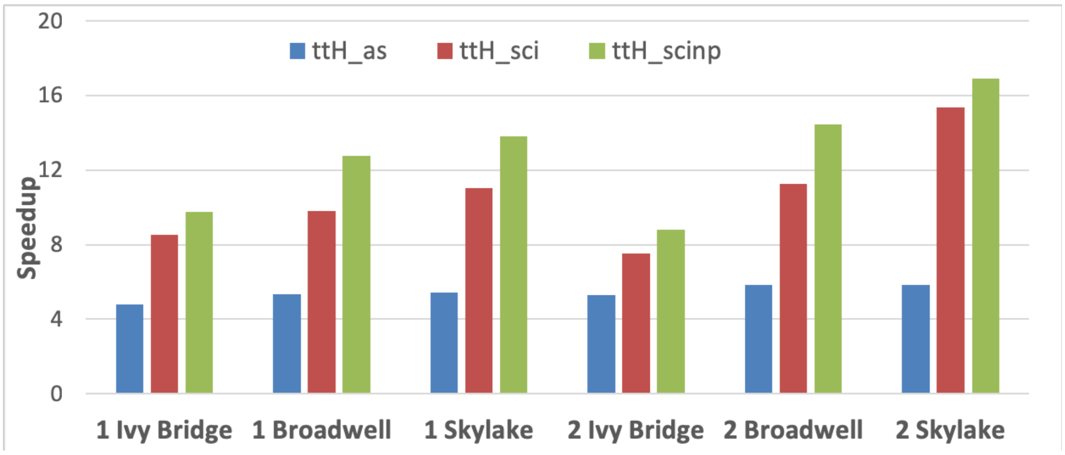


Figure 4.5: Speedup of the parallel $t\bar{t}H$ analyses with *HEP-Frame vs* a standard OpenMP parallelisation for the same number of threads on a server with single or dual multicore devices.

allocated in the memory bank of a single multicore device, which consistently causes higher memory access latencies for threads on the other multicore devices. This limitation could be overcome by using one process per multicore device, which would provide higher speedups as there would be independent data structures in each memory bank. This approach is later explored in Subsection 4.3.3.

The performance gap between *HEP-Frame* and OpenMP increases proportionally to the number of cores in the server, as shown by the improved speedup when using dual Broadwell and Skylake devices over a single device. This proves that the OpenMP `dynamic` scheduler, which uses a task pool strategy, is not as efficient as *HEP-Frame* when dealing with a high amount of threads, and that this improvement is not related to the pipeline reordering. Using the OpenMP `guided` scheduler provided a similar behaviour.

The default order of the pipeline on the three $t\bar{t}H$ analyses, as defined by the scientist that developed them, already has most propositions that filter out most dataset elements in the beginning and the heavier propositions in the final stages. Pipelined data stream applications with worse default pipeline orders would benefit more from the *HEP-Frame* pipeline reordering scheduling layer. The aggregated overhead of all functions related to the *HEP-Frame* multi-layer scheduler for `ttH_as` is 10%, and tends to decrease significantly for compute-bound code, as it is less than 5% for both ttH_sci and ttH_scinp.

HEP-Frame uses an array-of-structures type data structure that is not as efficient as a structure-of-arrays for memory-bound code, which is the case of ttH_as. However, this trade-off in performance is acceptable as it allows the framework to work with any type of dataset elements provided by the user, requiring only a class-like representation of the data.

4.3.3 Multiprocess on Multi-Socket Servers

Multi-socket servers display an additional challenge to performance tuning due to the NUMA architecture: each multicore device share its memory controller with the neighbouring device and may impose a potential bottleneck when there is no match between the core executing the code and the memory device where data is placed (known as core affinity). One way to avoid this situation is to allocate one multithreaded process to each device.

On the other hand, if a given application code contains a significant sequential part that can not be parallelised, the multithreaded approach does not use all cores during the sequential part of the code. In this case, allocating a fully autonomous sequential process to each core leads to a better overall performance. In between we need also to consider those applications that are mostly parallel but that their memory requirements may create conflicts among the core requests for data.

In pipelined data applications, such as those described in the presented case studies, HEP-Frame can efficiently explore all available cores using a multithreaded approach, but most likely the NUMA architecture may degrade the overall performance.

To test and validate these assumptions some experiments were devised: to compare a fully multithreaded approach with a multiprocess approach on one and on two devices in a server. Figure 4.6 shows the impact of allocating 2 multithreaded processes to a single device and 1 multithreaded process to each device in a dual-socket server.

As expected, the performance slightly degraded when more processes were allocated to a single device (down to 83%). When allocating a full process to each socket the data structures are no longer shared among threads in different devices, avoiding the memory penalties of NUMA, and the performance of the multiprocess implementation went up to 57% (ttH_-

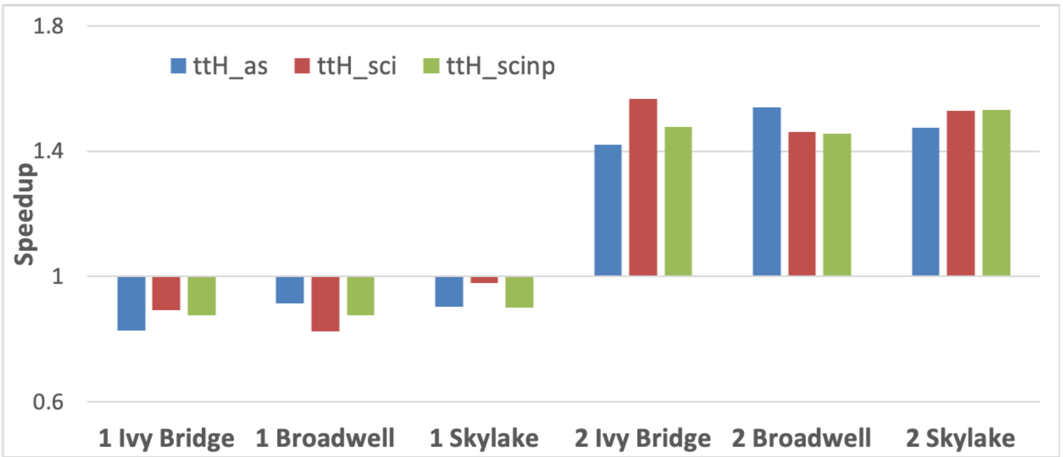


Figure 4.6: Comparative performance of a dual process *vs.* a single process implementation on single and dual socket servers.

`scinp` in dual Ivy Bridge).

To assess the assumption that *HEP-Frame* adequately uses the available computational resources in a single multicore device in a multithreaded environment, and that it suffers a small performance penalty when using multiple processes in a single multicore device an experiment was devised: compare the performance of *HEP-Frame* using 1 process against using 4 processes for single- and dual-socket servers. Figure 4.7 presents the results of this experiment using the case studies on 3 single-socket and 3 dual-socket multicore servers. Using 4 processes on single-socket servers greatly restricts the performance of all $t\bar{t}H$ analyses, as expected from the results of using 2 processes. The OpenMPI library used passes messages between processes of the same node with very low communication overhead through shared memory, but that is not enough to offset the costs of merging the results from 4 processes, and tends to increase with the number of processes.

`ttH_as`, `ttH_sci` and `ttH_scinp` applications improve up to 17%, 24% and 16% over a single process on the dual-socket servers, respectively. This improvement is lower than with 2 process, and is less consistent among different servers. Preliminary results showed that using more processes further degraded the performance on both single- and multiple-socket servers.

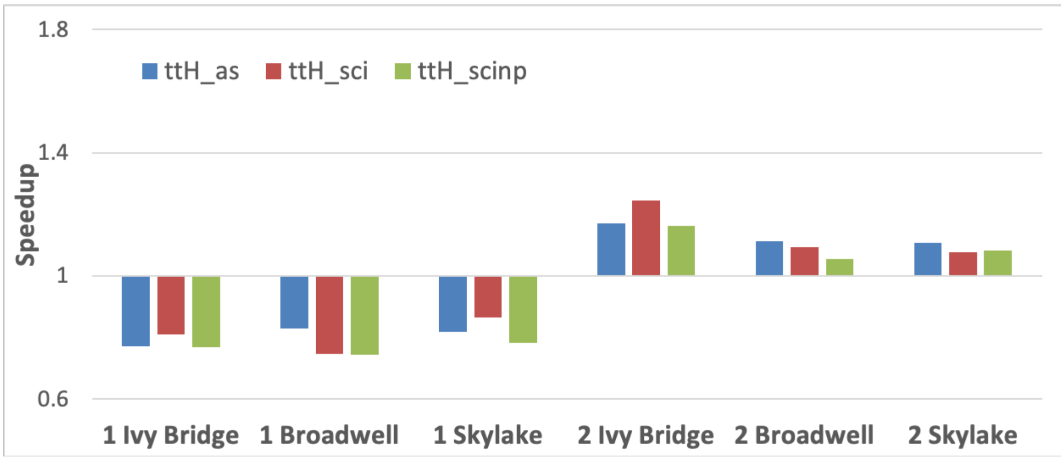


Figure 4.7: Comparative performance of a 4 process *vs.* a single process implementation on single and dual socket servers.

The overhead of balancing the input data files among the various threads is less than 5% of the overall application execution time for 4 processes. A similar preliminary evaluation was performed with other pipelined data stream applications from high energy physics, but the performance improvement of using 2 processes in dual-socket servers was not consistent. Due to this reason, automatic creation and management of multiple processes in a single server is yet to be implemented in *HEP-Frame*.

4.3.4 Proposition Offload to Knights Corner Accelerators

The Xeon Phi KNC coprocessor can be used as a computing accelerator, namely for parallel number crunching, due to the large number of cores and wider vector unit. However, the lack of L3 cache suggest that this device may have its performance degraded in memory-bound applications, such as `ttH_as`. To validate these assumptions, an experimental test compared the performance of a server with one or two KNC accelerators against a server without any accelerator. Figure 4.8 displays the experimental results, which confirmed the assumptions.

The KNC improved the performance up to 33% and 20% for the `ttH_sci` and `ttH_scinp` applications, respectively. The overall best performance was obtained with 2 Xeon device and

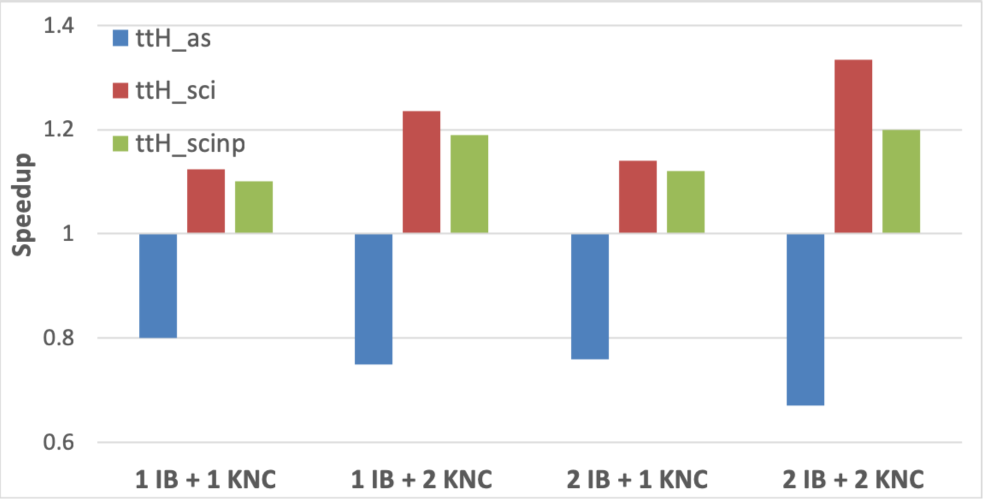


Figure 4.8: Speedup of the case studies in a *HEP-Frame* prototype on a server with one or two Ivy Bridge (IB) devices with KNC accelerators *vs* the same server without accelerators.

2 KNC, 38% faster than 1 Xeon with 2 KNC, as the increased amount of multicore threads can better utilise the resources of both accelerators. Using a single multicore device with 2 KNC devices distributes the overhead of memory management and scheduling of the accelerators through a smaller amount of multicore threads, which are not enough to hide these latencies. However, the KNC did not improve the performance of `ttH_as`, as expected, as this is a latency-bound code that does not have enough computation to use the KNC resources.

However, as already explained in Section 3.3.1, this implementation required modifying data structures to be offloaded to the KNC, which must be developed specifically for each different scientific code. At the moment, the *HEP-Frame* is not capable of generating this code automatically based on the scientists proposition code, as is a too complex process to be performed to be used only in a niche situation for marginal performance gains and is out of the scope of this work. A detailed analysis of this implementation is available in [71].

4.3.5 Efficient Generation of PRN Batches

Pseudo-Random Number Generators (PRNGs) are a compute intensive task that may have a significant impact on the performance of pipelined data stream applications. The use of an efficient implementation of a statistically sound PRNG, such as the Mersenne Twister, may improve the performance of applications, specially, but not exclusively, when they require large amount of Pseudo-Random Numbers (PRNs). An adequate management of the PRNs may also have an impact on the performance of an application. Using a dual buffer to store PRNs (referred to as *DB* in this subsection, whose implementation was previously described in Subsection 3.3.2) it is expected to be more efficient than using a single buffer (*SB*), and much faster than calling the PRNG whenever a PRN is required.

To assess these assumptions several implementations of the Mersenne Twister PRNG were tested on the `ttH_as`, `ttH_scinp` and `ttH_sci` case studies in *HEP-Frame*, which require a small, moderate and a large amount of PRNs, respectively, on multiple multicore servers with and without accelerators. The PCG RXS-M-XS PRNG (a linear congruential generator) was also tested since the authors claim a statistical quality similar to the Mersenne Twister with a better computational performance [63]. The original code of these case studies used the TRandom3 PRNG available in ROOT, which may not be adequate to generate large amounts of PRNs.

Figure 4.9 shows the speedup of the two 24-threaded versions of the $t\bar{t}H$ analyses, using the *HEP-Frame* scheduler with a single process in the server, with the selected PRNG algorithms (one per physical core of the Xeon devices) and the different approaches for PRNG concurrent execution, compared to the ROOT single number PRNG.

The efficient use of PRNGs provides larger performance improvements for the multithreaded `ttH_sci`, compared to its sequential execution. The use of a dual buffer *vs* single PRN generation provides an overall improvement across all PRNGs, specially with a speedup improvement from 42x to 48x for the PCG PRNG. The `ttH_scinp` behaves similarly to `ttH_sci` but with smaller improvements, up to 11x and 20x using using the MKL and PCG PRNGs, respectively. The performance of the `ttH_as` was not degraded by the use of PRN buffers, contrary to its sequential execution, with speedups up to 14% for the dual buffer PCG. Overall, the efficient use of PRNGs may provide a significant performance improvement on sequential and multicore applications. The usage of a dual buffer approach out-performs a single buffer and may lead to significant performance improvements on parallel code.

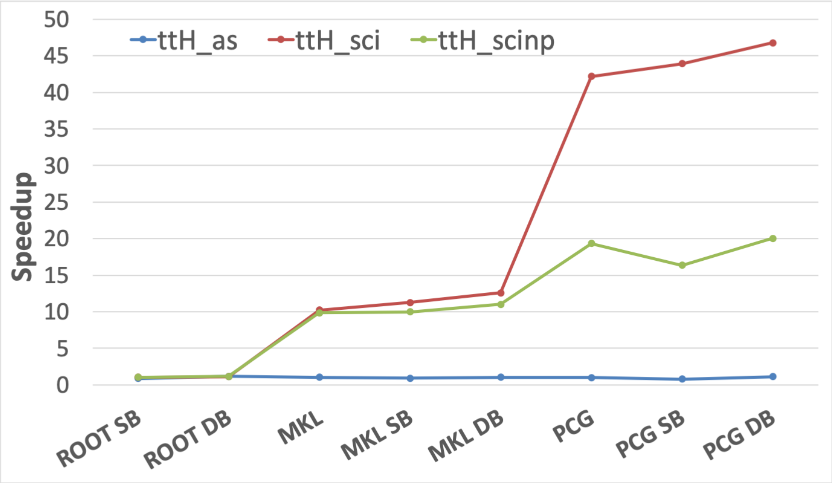


Figure 4.9: Speedup of the parallel $t\bar{t}H$ analyses with different PRNG algorithms and approaches *vs* the original ROOT single number PRNG on the Ivy Bridge server.

Vectorization may have a significant impact on the performance of PRNGs, as the generation of a large amount of PRNs has the necessary characteristics to adequately explore this optimisation. To assess the impact of vectorization on PRNGs, as well as the efficiency of these algorithms on newer vector extensions, the performance of the $t\bar{t}H$ analyses should be tested on different multicore architectures. While the IB architecture only has AVX, the BW devices use AVX2, which extends AVX by providing 256-bit wide vector operations on integers and implements fused multiply addition.

Figure 4.10 shows the performance improvement of using the dual-socket BW server over the IB server for the different on-device PRNGs using all available cores. The BW architecture favours the `ttH_as` application, as the code takes advantage of the extra cores to parallelise the reading of input data files, which is its main bottleneck. The single buffer approach is favoured for all PRNGs as one buffer is enough to hold all PRNs needed by this application. Further profiling showed that both `ttH_sci` and `ttH_scinp` improvements are not mainly caused by the additional cores but the usage of an improved vectorization instruction set (AVX on IB *vs* AVX2 on BW), specially by the MKL PRNG. ROOT PRNGs also improve, but are still far behind PCG and MKL in their overall performance. AVX2 improves the efficiency of

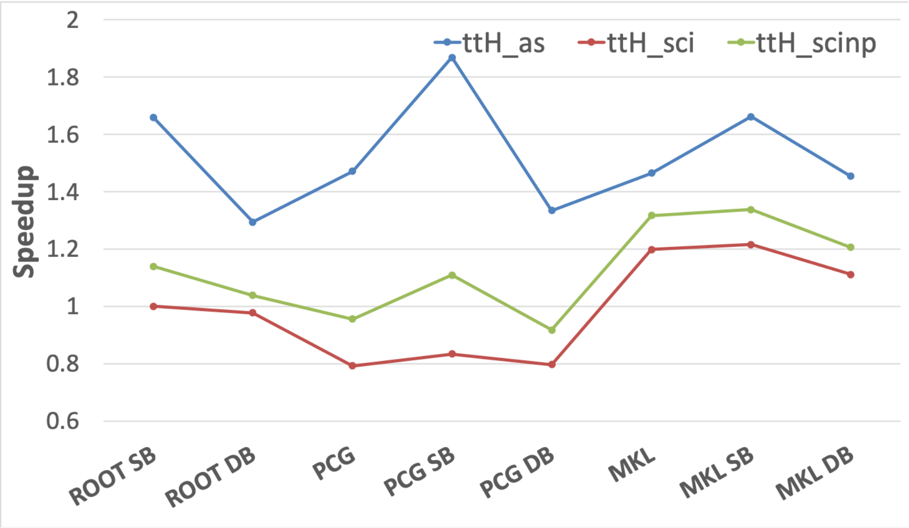


Figure 4.10: Performance comparison of the Broadwell server *vs* the Ivy Bridge server.

PRNGs over AVX, which is significant considering that the BW clock is up to 700 MHz lower than the IB server when executing vector instructions.

Hardware accelerators are often designed to improve the performance of highly parallel code that operates on independent data. There are several implementations available of the Mersenne Twister PRNG specially tuned for the hardware characteristics of GPU and KNC devices. A Kepler and Pascal GPUs, and a KNC coprocessor were used to assess the efficiency of these devices to generate large amounts of PRNs on the case studies (Figure 4.11), using efficient implementations of the Mersenne Twister in cuRAND and MKL libraries. Offloading PRNG to these accelerators frees computational resources on the multicore devices to be used by other computations of the $t\bar{t}H$ analyses. Additional KNL and IB servers were also used to offload the PRNG, to ensure that the performance improvements obtained using the accelerators were not only due to the increase in computational resources allocated to the $t\bar{t}H$ analyses. The speedups of the 10-core Pascal server were extrapolated to 24-cores to get an insight in how the Pascal and Kepler devices compare. It was not possible to mount the desktop Pascal GPU device on the IB server since it is a node in a computing cluster that requires GPUs with specific cooling solutions. Both Pascal and Kepler GPUs provide performance improvements up to 70x and 12x for the `ttH_sci` and `ttH_scinp` applications, respectively. A similar behaviour is observed with the KNC and KNL devices, with speedups of 47x and 51x for the `ttH_sci` and `ttH_scinp` applications, respectively. Using an extra IB server provides an improvement of 65x and 11x for these applications. The performance of the `ttH_as` application was decreased by $2x$ as its lack of heavy computations and need for PRNs is not enough to hide the costs of the memory transfers over PCI-Express and ethernet.

The use of a dual buffer *vs* a single PRN/buffer is crucial to minimise the impact of the PRN transfer from the devices to the host server. The GPU devices and the IB server over ethernet spent > 90% of the PRNG time on transferring the PRNs to the host device, while manycore devices spent 80% of their execution time. The overall higher speedups, compared to using only the host server, are due to the higher availability of the Xeon cores to perform application specific computations, since they were freed from generating PRNs.

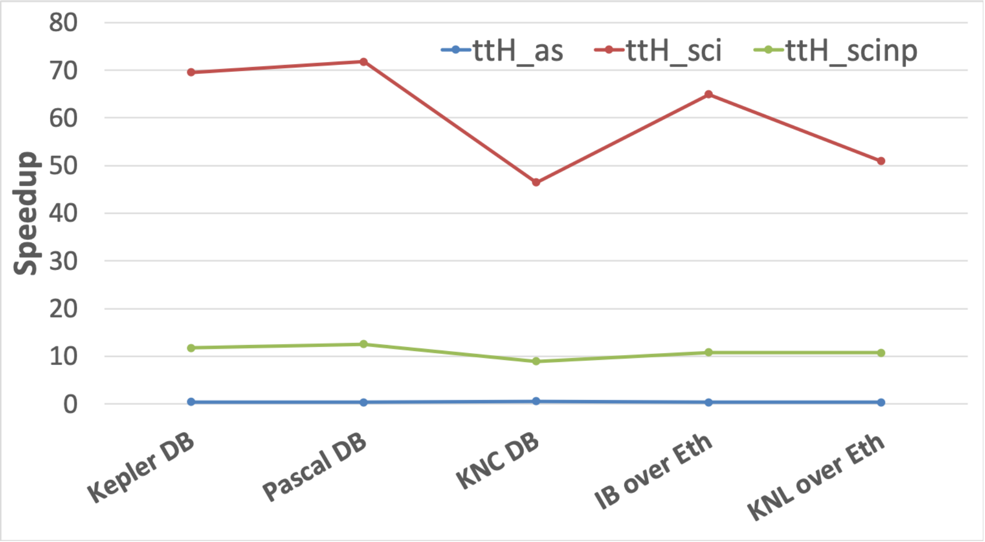


Figure 4.11: Speedup of the parallel applications with different PRNG algorithms using external computing devices *vs* the original ROOT single number PRNG.

The PRNG throughput using the Mersenne Twister of various multicore, manycore and accelerator devices is compared in Figure 4.12: dual buffer MKL for the dual-socket IB, dual-socket BW, KNC and KNL servers; dual buffer cuRAND for Pascal and Kepler Nvidia GPUs.

This comparison assesses the performance of each device, regardless of the application that the PRNG is integrated in, while measuring the impact that PCI-Express and ethernet have when accounted for the PRN throughput. The red bar considers each device PRN throughput, while the blue takes into account data transfer times over their respective interconnection. The single-socket BW server was not considered since the dual-socket BW server provides better performance for the same CPU device architecture.

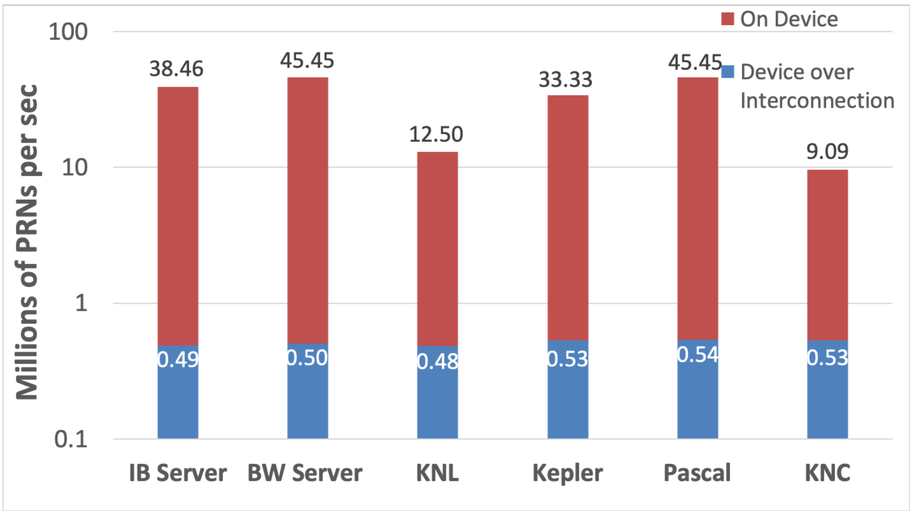


Figure 4.12: Throughput of the best PRNG for each different server and accelerator device.

The BW server is 18% faster than the IB server mostly due to the better vectorization operations available in its AVX2 instruction set. However, the performance does not scale linearly due to the lower clock rate of the BW server and the amount of PRNs requested is not enough to take advantage of the extra cores. The performance of both IB and BW servers is greatly degraded when the PRNs have to be transferred through ethernet. The Pascal and Kepler devices provide a throughput around 46M PRNs per second (justifying their similar speedup for the `ttH_sci` application), but their limited to only 0.54M PRNs per second due to the bottleneck of the PCI-Express interface. A similar bottleneck is observed with the KNL and KNC manycore devices over their respective interconnections, but their on device PRN throughput is the lowest of all tested devices, as they have the lower clock speeds and MKL cannot fully utilise the available vector processing units.

Accelerator devices provide great PRN throughput while freeing the host CPU devices to perform other calculations, which is a crucial factor to improving the performance of the case studies previously presented. The desktop Pascal GPU provided similar performance to the server-grade Kepler device, as the improvements in the architecture and clock rate overcome the lack of CUDA cores, while being significantly cheaper. Offloading PRNGs to KNL, IB and BW servers should also be considered if a lower latency interconnection is used, such as Myrinet and InfiniBand.

These tests prove that data stream applications that require a huge amount of PRNs can greatly benefit by efficiently using PRNGs, regardless of the server architecture and configuration in which they will execute. It is crucial that frameworks, such as *HEP-Frame*, provide efficient implementations for commonly used functions as is the case of PRNGs. The PCG PRNG was the best performing when using only multicore devices to process both the application code and PRNG. However, the PCG suite uses a more computationally efficient algorithm than the Mersenne Twister, which may not be a fair comparison. It is the responsibility of the end user to assess if this PRNG should be used over other traditional PRNGs, which are well accepted and extensively tested by the mathematics's community.

An in-depth analysis of all the details related to different PRNGs and their efficiency in various computing devices can be found in [13]. The best PRNG for each server configuration will be used in the tests of the next subsections, so that the evaluation of *HEP-Frame* is not biased by the inefficient ROOT PRNG.

4.3.6 *HEP-Frame* in a Manycore KNL Server

The mesh structure that interconnects the compute tiles and the memory organisation of the KNL package is configured in boot time. Each tile is a dual core PU, sharing cache L2; the KNL package also includes 8 chips of configurable embedded RAM (eRAM). KNL configurations were detailed in Subsection 2.1.2.

Experimental tests evaluated the performance of the $t\bar{t}H$ scheduler on the KNL server with 4 processes and a total of 64, 128 and 256 threads. Figure 4.13 show the results compared against the original multicore scheduler with 24 threads on the dual 12-core IB server.

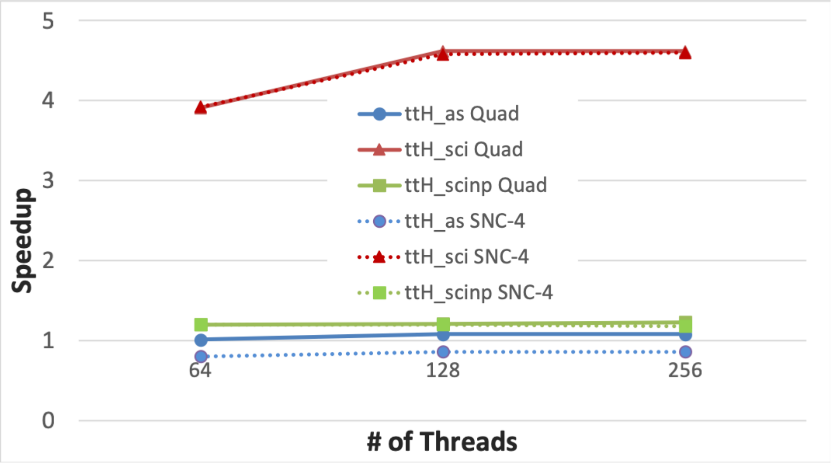


Figure 4.13: Speedup of KNL server configurations *vs* the multicore dual-socket IB server.

Results show that:

- The best configuration for the computing tiles is the quadrant/SNC mode.
- eRAM as flat addressable RAM was the best across all cluster configurations. eRAM as cache decreases performance (10-30% in all case studies), as this type of code reuses little data (independent processing of each dataset element).
- The peak speedup of 4.6x over the multicore server for the `ttH_sci`, with 128/256 threads, are mostly due to the vectorization capabilities, larger overall L2 cache of this device and 4 independent processes with its own *DS*.
- The all-to-all clustering mode was 2*x* slower than the quadrant/SNC-4 configurations.

The multicore scheduler assigns a combination of a proposition and a dataset element to each thread at a time, reducing the use of vectorizable code on this server. However, the simplified reordering approach on the KNL server did not take advantage of the inefficient `ttH_scinp` pipeline, as the complex scheduler on the multicore does: the speedups are not as high as in the `ttH_sci` (1.2x).

Figure 4.14 compares the performance of the KNL server with three different multicore servers without accelerators (with Ivy Bridge, Broadwell and Skylake devices), and the Ivy Bridge server with one Kepler GPU.

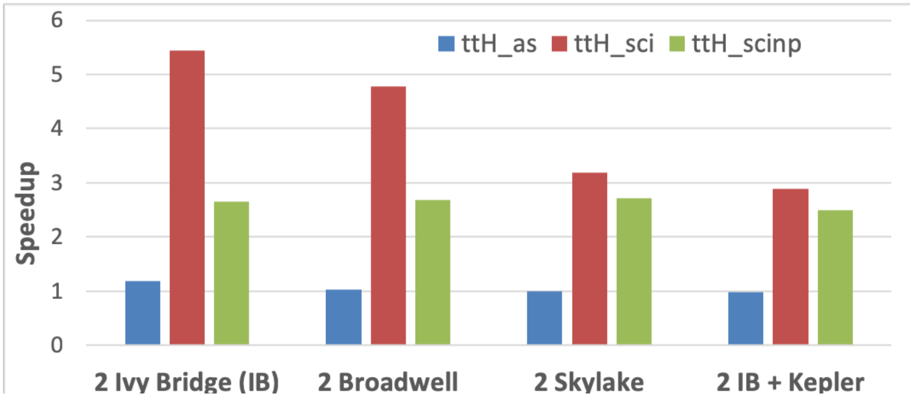


Figure 4.14: Speedup of the KNL server *vs* 3 multicore dual-socket servers and a 4th with a Kepler GPU.

The `ttH_sci` application running with 128/256 threads on the KNL outperforms the multicore servers with speedups up to 5.5x. However, it only improved by 3x compared to the server with a Kepler GPU, as a significant part of the execution time of this application (PRNG) is accelerated by this device. The memory-bound `ttH_as` does not improve as the KNL is designed for highly parallel and vectorizable compute-bound code.

Multiple Servers

Figure 4.15 shows the scalability of the case studies on *HEP-Frame* for 2, 4 and 6 KNL servers, when compared to a single server.

As expected, the memory-bound `ttH_as` analysis scales less with the increase in processing power, as the performance improvements are provided by the increase in memory and I/O bandwidth for the file reading and data structure creation and pre-processing. `ttH_scinp` scales better than `ttH_as`, with an almost linear speedup up to 4 servers. Beyond this it does not scale as well, as more *DS* threads are required by the increased computational throughput, leaving less room for *DP* threads. `ttH_sci`, the most compute intensive applica-

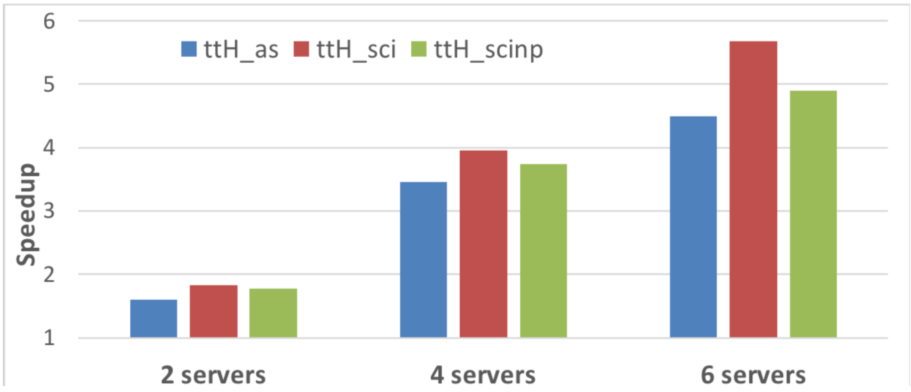


Figure 4.15: Speedup of the case studies for 2, 4 and 6 KNL servers *vs* a single KNL server.

tion, scales better than the other case studies with a speedup of 1.9x, 3.9x and 5.6x for 2, 4 and 6 servers, respectively. The scalability of using 2, 4 and 6 multicore Xeon servers is similar to the presented results.

4.3.7 *HEP-Frame vs.* StarPU

Figure 4.16 compares the performance of the *HEP-Frame* scheduler with the HEFT *dm* scheduler in StarPU, the most common list scheduler for these applications, on both a KNL server and a single dual-socket Xeon server. The case studies depend on the ROOT framework, which prevented the port of the code to NVidia GPUs, as was discussed in Section 4.1. However, the HEFT scheduling efficiency is the same when dealing with multiple multicore/-manycore devices on a server with or without accelerators.

On both Xeon and KNL servers the *HEP-Frame* scheduler outperformed StarPU. `ttH_sci` had only a 50% improvement on *HEP-Frame* as this code benefited less from reordering (the most compute intensive proposition is at the end of the pipeline, while the most filtering are at the beginning by default) and behaved more as a regular compute-bound application, where the last proposition took 70% of the analysis execution time.

The speedup went to 2.5x for the `ttH_scinp` analysis, as it benefited the most from the proposition reordering in *HEP-Frame*, since the default order was not as good as in `ttH_sci`.

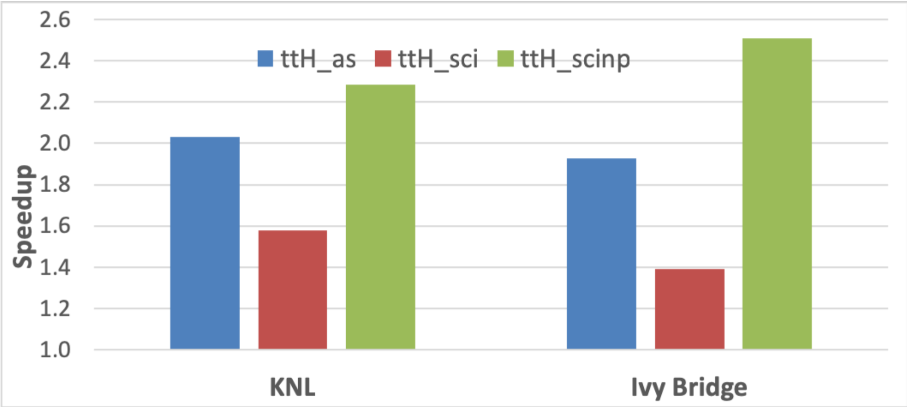


Figure 4.16: Speedup of the *HEP-Frame* scheduler *vs* the HEFT scheduler in StarPU on the KNL manycore server and a dual-socket IB server.

The StarPU HEFT scheduler also executed the propositions out-of-order, but only based on the throughput of each computing device. StarPU was less efficient with memory-bound code, as shown by the speedup of using *HEP-Frame* for `ttH_as`, which was achieved by adapting the amount of *DS* threads accordingly. This behaviour was expected since memory-bound code is not the target application type of StarPU.

The dynamic tuning of *DS* and *DP* threads accounted for up to 2x speedup of *HEP-Frame* over StarPU. Both *HEP-Frame* and StarPU behaved similarly on KNL and the Xeon servers, with only a minor advantage of *HEP-Frame* on the KNL for `ttH_sci` over the Xeon (58% *vs* 39% better than StarPU, respectively).

4.3.8 Overall Performance *vs.* the Original Case Studies

Figure 4.17 compares the performance of the three case studies implemented on *HEP-Frame* with their original sequential implementation on the best multicore servers (with dual Broadwell and Skylake devices), Ivy Bridge server with a Kepler GPU, and the KNL server.

HEP-Frame was set to use only one process per server, its default configuration, but, as shown in Subsection 4.3.4, using multiple processes in a multi-socket server may provide even larger overall speedups. *HEP-Frame* provided a significant performance improvement

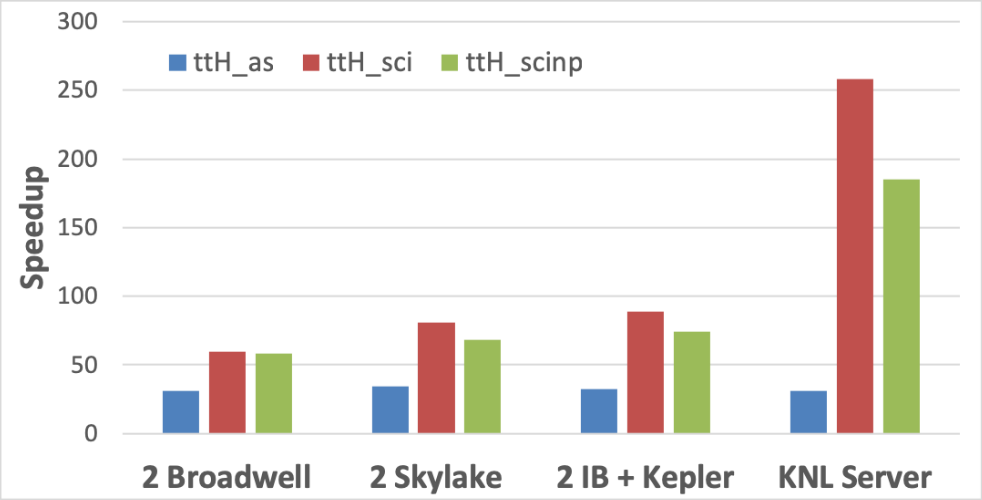


Figure 4.17: Overall speedup of the case studies on *HEP-Frame vs* their original sequential implementations.

for every case study, confirming it is portable across multiple platforms with crucial architectural differences. It adapted well to irregular compute-bound code, with speedups up to 252x and 185x for the `ttH_sci` and `ttH_scinp` applications on the KNL server. It also efficiently handled the memory-bound `ttH_as` application, with a speedup 30x for every server, due to its dynamic tuning of *DS* and *DP* threads. Note that for this final evaluation, the original, very inefficient, ROOT PRNG was replaced by the single number MKL PRNG to ensure a fair comparison with *HEP-Frame*.

The KNL server outperformed every other server mainly due to its core count and greater vectorization capabilities: two AVX-512 vector units per core, while Skylake has only one AVX-512 vector unit and Broadwell and Skylake have AVX units to operate on 256 bits. These two also suffered significant down clock frequency due to the AVX instructions, which is less severe on the KNL.

Another contribution for the performance gap between KNL and the multicore servers was the KNL configuration as SNC-4, which forced HEP-Frame to schedule 4 processes to the KNL device, reducing thread synchronisations and data consistency overheads. The gap could be reduced if the user had defined a similar multiprocess approach to the dual-socket servers.

4.4 Summary

The $t\bar{t}H$ analysis code is a set of three applications used by high energy physicists at CERN, and it was chosen a case study representative of a wide set of pipelined data stream applications. The three versions of the $t\bar{t}H$ analyses, `ttH_as`, `ttH_sci` and `ttH_scinp`, have pipelines with 18 propositions in a good default order as defined by the developers. The former analysis is latency-bound, as very little processing is performed by the pipeline, while the latter two are compute-bound, where `ttH_scinp` displays 2 different propositions and a worse default pipeline order. These analyses depend on functions of the ROOT framework, which limits the portability of propositions to GPU devices. The $t\bar{t}H$ analyses are originally sequential, but a multithreaded map-reduce parallelisation with OpenMP, common in the scientific community, and with StarPU was performed to provide a thread-by-thread comparison with *HEP-Frame*.

The dynamic tuning of the amount of threads for data setup and data processing provided significant speedups over the conventional strategy of loading data and then processing it, with an improvement up to 4.3x and 2.9x for the `ttH_as` and `ttH_scinp` applications. The most compute intensive application, `ttH_sci`, did not benefit as much as the data setup accounts for a small percentage of the overall execution time.

The pipeline parallelisation and reordering provided significant performance improvements for all $t\bar{t}H$ analyses over an OpenMP parallelisation using the same amount of threads. `ttH_as`, `ttH_sci` and `ttH_scinp` improved by up to 6x, 15x and 17x, respectively, for the dual 24-core Skylake server. `ttH_as` had the smallest improvements as it is latency-bound, while `ttH_scinp` had the biggest improvement due to its worse default pipeline order. The performance improvements could be greater for pipelined data stream applications with worse default pipeline orders. The performance improvement of these analyses can improve by 50% if two processes are used in the dual-socket servers, as this approach avoids the memory latency penalties of NUMA.

HEP-Frame can take advantage of KNC and GPU accelerators: the first for proposition offload and PRNG acceleration, while the second only for PRNG acceleration. The KNC only provided speedups up to 33% and 22% for the `ttH_sci` and `ttH_scinp` applications, respectively, while degrading the `ttH_as` performance by 27%. The use of Kepler and Pascal GPUs improved the performance of the `ttH_sci` and `ttH_scinp` by 70x and 12x, respectively, whose improvement is proportional to the amount of PRNs required by each of these applications. *HEP-Frame* can also offload the PRNG to KNC and multicore/manycore servers, but providing a smaller improvement than using GPUs.

The performance of `ttH_sci` and `ttH_scinp` is improved by 2.9x and 2.5x, respectively, on the KNL manycore server when compared to the best heterogeneous server, the dual-socket Ivy Bridge server with a Kepler GPU. The performance of the latency-bound `ttH_as` is similar in both servers. *HEP-Frame* is faster than StarPU, its closest competitor, for all $t\bar{t}H$ analyses on both KNL and Ivy Bridge servers, with speedups between 1.4x and 2.5x.

HEP-Frame improves the performance of the `ttH_as`, `ttH_sci` and `ttH_scinp` applications on the KNL server by 30x, 252x and 185x, respectively, over their original sequential implementation. It also improves the performance of these applications by 32x, 89x and 74x for an Ivy Bridge server with a Kepler GPU. *HEP-Frame* ensures efficient execution of both memory- and compute-bound pipelined data stream applications portable across various homogeneous and heterogeneous servers with accelerators, without requiring any modification of the code, configuration by the user, or prior knowledge of the server characteristics.

Chapter 5

Conclusions and Future Work

This chapter shares the final thoughts and conclusions on the work presented in this document. The research work presented in the previous chapters is revisited and summarised, with a look into the most relevant results.

Finally, a view on several possible research paths to further improve this work is presented.

This key component of this thesis work is *HEP-Frame*, a framework to aid the development and efficient execution of pipelined data stream applications in homogeneous and heterogeneous servers. Pipelined data streaming is commonly used in the non-computer scientific community, where researchers develop I/O-, memory-, or compute-bound applications to analyse large amounts of data. Performance is key for this type of applications, but researchers often lack the expertise to efficiently parallelise their code, specially for heterogeneous servers, as the programming paradigm and available frameworks have steep learning curves. The goal of this framework was to provide an user-centred development interface, through the use of code skeletons, automatic code generation, and automation of the compilation process, while transparently managing the efficient parallel execution of the code on multicore, manycore, and accelerator devices.

HEP-Frame was evaluated with three versions of a real world application: the $t\bar{t}H$ particle physics event data analysis, developed and used by CERN researchers, ported into *HEP-Frame* by the scientists who designed the code. The main component of *HEP-Frame* is its multi-layer scheduler, each layer performing a different action:

- The top layer balances data and workloads among servers in a heterogeneous cluster environment. It scaled for both memory- and compute-bound codes, in either multiple homogeneous or multiple heterogeneous servers. The compute-bound `ttH_sci` application improved by 1.9x, 3.9x, and 5.6x when using 2, 4, or 6 servers, respectively.

- The middle layer dynamically tunes the number of threads assigned to the parallel data read and setup (*DS*), including the creation of adequate data structures, and the pipeline execution (*DP*). It provided speedups up to 4x, when compared to a fixed configuration of *DS* and *DP* threads, for the same amount of total threads.

- The bottom layer addresses the scheduling at the server level, managing the parallel execution of the dataset workload among the available computing resources in a server. This layer includes the reordering of the pipeline propositions of the same dataset element and the parallel execution of multiple dataset elements, ensuring that pipeline propositions that filter out most elements are executed as early as possible. The performance of ttH_sci and ttH_scinp was improved by 15x and 17x, respectively, over an OpenMP paralellisation, which was mostly due to the pipeline reordering and parallel proposition execution in this layer.

The *HEP-Frame* PRNG management was tested by offloading the generation to additional multicore and manycore servers, as well as Kepler and Pascal NVidia GPUs and the KNC co-processor. For ttH_sci, the most PRN intensive case study, he use of a proper PRNG and management strategy provided speedups up to 70x, using Kepler and Pascal GPUs, over the inefficient ROOT PRNG on the original application code. The ROOT PRNG was not used for any of the remaining tests. The Kepler GPU PRNG achieved almost 2x performance improvement over the most efficient multicore PRNG on an Ivy Bridge server for the same case study; for the other $t\bar{t}H$ versions the speedup was marginal as they do not rely as much on PRNs.

The *HEP-Frame* scheduler for the KNL manycore server provided speedups up to 5.5x, 4.8x and 3.2x compared to a homogeneous multicore server, respectively two Ivy Bridge, two Broadwell and two Skylake devices, for the ttH_sci application. The KNL server also outperformed the Ivy Bridge server with a Kepler GPU accelerator, by 3x and 2.5x for ttH_sci and ttH_scinp applications, respectively. Although the KNL architecture was not designed to efficiently handle latency-bound code, the *HEP-Frame* scheduler ensured that the ttH_as application on the KNL server ran with the same performance as the multicore servers.

HEP-Frame outperforms StarPU, a competitive framework that targets a broader range of applications, on both multicore and manycore servers. ttH_as and ttH_scinp improved up to 1.9x and 2.5x, respectively, due to the middle and bottom layers of *HEP-Frame* scheduler on the Ivy Bridge server. ttH_sci only improved by 1.4x, as it benefits less from the pipeline reordering, which results on a scheduling strategy similar to StarPU.

Overall, *HEP-Frame* improved the performance of the ttH_sci and ttH_scinp applications 252x and 185x on the manycore server, while ttH_as improved by 30x across all servers, over the original sequential code using the MKL single number PRNG. Comparing *HEP-Frame* with the original $t\bar{t}H$ versions using the ROOT PRNG would provide bigger speedups but would not be indicative of the benefits of using the multi-layer scheduler. Pipelined data stream applications with worse pipeline orders by default would see even greater speedups.

Given the performance of the *HEP-Frame* in the $t\bar{t}H$ analysis, its use was extended to 6 analyses related to:

- $t\bar{t}$ production at the LHC, in both the semileptonic and dileptonic final states.

- Single top quark production through the t-channel and the Wt-channel in either semileptonic and dileptonic final states.

HEP-Frame allowed, for the first time, to use a common framework across several physics groups in ATLAS (Single Top Group and Top Properties Group, as well as the Higgs Group), which otherwise would not be possible, as each group tends to use specific file formats and code structures. This increase in performance largely covers the expected increase of computational resources required to explore data analysis during the third and fourth run of the high luminosity phase at the LHC, as previously described in Section 1.1.

5.1 Future Work

This dissertation focused on providing a tool to aid the development of pipelined data stream applications, specially for non-computer scientists, while ensuring efficient and portable execution of the code across various homogeneous and heterogeneous servers. However, there are still improvements that can be made to *HEP-Frame* in both fronts.

There are two key features that should be implemented to streamline the development of applications in *HEP-Frame*. The use of external tools (addressed as plugins throughout this document) at compile time has to be integrated into the *HEP-Frame* compiling process by adding them into specific sections of key Makefiles. A more intuitive and elegant solution would be to create an interface, such as a XML configuration file, where the user would specify which tool would be used in a given stage of the compilation process. A XML configuration would allow the definition of inputs and expected outputs of the plugins, which would be managed by a *HEP-Frame* compiling assistant. These configuration files would be distributed with the plugins to ensure plug-and-play installation and portability.

In the current version of *HEP-Frame*, users have to manually identify and indicate data dependencies to the framework to ensure the code correctness when reordering the pipeline. However, from our experience working with particle physicists, non-computer scientists often lack the expertise to clearly identify every data dependency in complex propositions. A plugin that parses the code files with the pipeline propositions and automatically infers data dependencies could be integrated into *HEP-Frame*, which would later be used by the multi-layer scheduler without any user interaction. This would free users from a tedious and error-prone task.

The results in Subsection 4.3.3 indicate that the performance of some pipelined data stream applications could benefit from using multiprocess parallelisation in multi-socket servers. This approach has to be tested with more case studies to find and implement into *HEP-Frame* an adequate heuristic to create and manage a suitable amount of processes for memory- and compute-bound code.

Some pipeline data stream applications present propositions that, depending on their result, can lead to the execution of different subsequent propositions. These propositions can be defined in nested pipelines, where the result of a proposition can lead to the execution of different sub-pipelines, in addition to the filtering out of dataset elements. The execution flow of these nested pipelines can be defined as *k-ary* trees, whose scheduling for parallel execution on homogeneous and heterogeneous servers is not widely researched by the HPC community. Allowing the coding and efficient execution of nested pipelines in *HEP-Frame* is crucial to ensure that the framework can be used by a wider scientific community. Support for nested pipelines is currently being developed in *HEP-Frame*.

HEP-Frame currently supports data streaming into a single data structure, whose elements are defined by a C++ class provided by the user. However, some applications use multiple input streams of data with different structures, where the execution of certain propositions requires that one of the data structures is complete while another is streamed in. Such is the case of several matrix-matrix operations present in query execution in linear algebra based systems. Support for this type of I/O would allow *HEP-Frame* to be used by a wider computer science community.

References

[1] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210," NVidia, Santa Clara, California, USA, Tech. Rep., 2014. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[2] C. Lomont, "Introduction to Intel Advanced Vector Extensions," *Intel White Paper*, pp. 1–21, 2011.

[3] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! Technology: Architecture and Implementations," *IEEE Micro*, vol. 19, no. 2, pp. 37–48, 1999.

[4] Z. Merali, "Computational Science: ...Error," *Nature International Weekly Journal of Science*, vol. 467, pp. 775–777, October 2010.

[5] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do Scientists Develop and Use Scientific Software?" in *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, May 2009, pp. 1–8.

[6] P. Prabhu, H. Kim, T. Oh, T. B. Jablin, N. P. Johnson, M. Zoufaly, A. Raman, F. Liu, D. Walker, Y. Zhang, S. Ghosh, D. I. August, J. Huang, and S. Beard, "A Survey of the Practice of Computational Science," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–12.

[7] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, “Software Development Environments for Scientific and Engineering Software: A Series of Case Studies,” in *29th International Conference on Software Engineering (ICSE’07)*, May 2007, pp. 550–559.

[8] J. Catmore, J. Cranshaw, T. Gillam, E. Gramstad, P. Laycock, N. Ozturk, and G. A. Stewart, “A new petabyte-scale data derivation framework for ATLAS,” *Journal of Physics: Conference Series*, vol. 664, no. 7, p. 072007, dec 2015.

[9] A. Di Girolamo, “The ATLAS Distributed Computing Project for LHC Run-2 and Beyond,” *PoS*, vol. EPS-HEP2015, p. 260, 2015.

[10] F. H. Barreiro Megino, “The Future of Distributed Computing Systems in ATLAS: Boldly Venturing Beyond Grids,” Tech. Rep., Jun 2018. [Online]. Available: https://cds.cern.ch/record/2627546

[11] T. F. Pfleiger, A. E. Kimball, and A. A. Desai, “Distributed Query Engine Pipeline Method and System,” Dec. 30 2008, uS Patent 7,472,112.

[12] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, “A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[13] A. Pereira and A. Proenca, “Efficient Use of Parallel PRNGs on Heterogeneous Servers,” in *Proceedings of the International Conference on Mathematical Applications*. Institute of Knowledge and Development, 2018, pp. 7–12.

[14] A. Pereira, A. Onofre, and A. Proenca, “Tuning Pipelined Scientific Data Analyses for Efficient Multicore Execution,” in *Proceedings of the International Conference on High Performance Computing Simulation (HPCS)*. IEEE, 2016, pp. 751–758.

[15] ——, “HEP-Frame: A Software Engineered Framework to Aid the Development and Efficient Multicore Execution of Scientific Code,” in *Proceedings of the 2015 International Conference on Computational Science and Computational Intelligence*. IEEE, 2015, pp. 615–620.

[16] ——, “Removing Inefficiencies from Scientific Code: The Study of the Higgs Boson Couplings to Top Quarks,” in *Proceedings of the 14th International Conference on Computational Science and Its Applications.* Springer International Publishing, 2014, pp. 576–591.

[17] “An Introduction to the Intel QuickPath Interconnect,” Intel, Santa Clara, California, USA, Tech. Rep., 2009. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf

[18] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger, S. FELLOW *et al.*, “The Next Generation AMD Enterprise Server Product Architecture,” *IEEE Hot Chips*, vol. 29, 2017.

[19] R. R. Schaller, “Moore’s Law: Past, Present and Future,” *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[20] T. Boku, M. Sato, M. Matsubara, and D. Takahashi, “OpenMPI-OpenMP Like Tool for Easy Programming in MPI,” in *Sixth European Workshop on OpenMP*, 2004, pp. 83–88.

[21] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, “Test-Driving Intel Xeon Phi,” in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering.* ACM, 2014, pp. 137–148.

[22] J. Ajanovic, “PCI Express (pcie) 3.0 Accelerator Features,” *Intel Corporation*, vol. 10, 2008.

[23] B. Thompto, “POWER9: Processor for the Cognitive Era,” in *Hot Chips 28 Symposium (HCS), 2016 IEEE.* IEEE, 2016, pp. 1–19.

[24] TOP500.org, “TOP500 List - November 2018.” [Online]. Available: https://www.top500.org/list/2018/11/

[25] ——, “GREEN500 List - November 2018.” [Online]. Available: https://www.top500.org/green500/list/2018/11/

[26] T. Instruments, “Digital Signal Processors.” [Online]. Available: http://www.ti.com/processors/digital-signal-processors/overview.html

[27] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *CoRR*, vol. abs/1803.06185, 2018.

[28] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks," *Neural Computing and Applications*, 09 2018.

[29] D. P. Kaz Sato, Cliff Young, "An In-Depth Look at Google's First Tensor Processing Unit (TPU)." [Online]. Available: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[30] D. S. Modha, "Introducing a Brain-Inspired Computer." [Online]. Available: http://www.research.ibm.com/articles/brain-chip.shtml?utm_source=Abundance%20Insider&utm_medium=email&utm_term=Computing&utm_content=Computing&utm_campaign=8%2F27

[31] G. Synek, "Intel Nervana is a Neural Network Processor to Accelerate AI." [Online]. Available: https://www.techspot.com/news/71453-intel-nervana-neural-network-processor-accelerate-ai.html

[32] "CERN Annual report 2017," CERN, Geneva, Tech. Rep., 2018. [Online]. Available: https://cds.cern.ch/record/2624296

[33] S. Andringa *et al.*, "Current Status and Future Prospects of the SNO+ Experiment," *Adv. High Energy Phys.*, vol. 2016, p. 6194250, 2016.

[34] D. Akerib, X. Bai, S. Bedikian, A. Bernstein, A. Bolozdynya, A. Bradley, S. Cahn, D. Carr, J. Chapman, K. Clark *et al.*, "The LUX Prototype Detector: Heat Exchanger Development," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 709, pp. 29–36, 2013.

[35] The Pierre Auger Collaboration, "Search for High-energy Neutrinos from Binary Neutron Star Merger GW170817 with ANTARES, IceCube, and the Pierre Auger

Observatory," *The Astrophysical Journal Letters*, vol. 850, no. 2, p. L35, 2017. [Online]. Available: http://stacks.iop.org/2041-8205/850/i=2/a=L35

[36] "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002.

[37] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel Math Kernel Library," in *High-Performance Computing on the Intel Xeon Phi.* Springer, 2014, pp. 167–188.

[38] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: a Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, Oct 1992, pp. 120–127.

[39] H. Jasak, A. Jemcov, Z. Tukovic *et al.*, "OpenFOAM: A C++ Library for Complex Physics Simulations," in *International workshop on coupled methods in numerical dynamics*, vol. 1000. IUC Dubrovnik, Croatia, 2007, pp. 1–20.

[40] L. Dagum and R. Menon, "OpenMP: an Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[41] NVidia, "Cublas Library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[42] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse Library," in *GPU Technology Conference*, 2010.

[43] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.

[44] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC — First Experiences With Real-World Applications," in *European Conference on Parallel Processing.* Springer, 2012, pp. 859–870.

[45] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-Core Parallel Programming Environment," in *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, vol. 28, 2007.

[46] T. Urhan and M. J. Franklin, "Dynamic Pipeline Scheduling for Improving Interactive Query Performance," in *Proceedings of the 27th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., 2001, pp. 501–510.

[47] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive Ordering of Pipelined Stream Filters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, 2004, pp. 407–418.

[48] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2006, pp. 151–162.

[49] H. Arabnejad and J. G. Barbosa, "List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, March 2014.

[50] C. Min and Y. I. Eom, "Dynamic Scheduling of Irregular Stream Programs toward Many-Core Scalability," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1594–1607, 2015.

[51] Y. Liu, L. Meng, I. Taniguchi, and H. Tomiyama, "Novel List Scheduling Strategies for Data Parallelism Task Graphs," *International Journal of Networking and Computing*, vol. 4, no. 2, pp. 279–290, 2014.

[52] J. C. Beard, P. Li, and R. D. Chamberlain, "RaftLib: A C++ Template Library for High Performance Stream Parallel Processing," *The International Journal of High Performance Computing Applications*, vol. 31, no. 5, pp. 391–404, 2017.

[53] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox: Modern Stream Processing on a Multicore Machine," in *2017 USENIX Annual Technical Conference*. USENIX, 2017, pp. 617–629.

[54] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. [Online]. Available: https://hal.inria.fr/inria-00550877

[55] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, pp. 66:1–66:11.

[56] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[57] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[58] G. Marsaglia, "The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness," *http://www.stat.fsu.edu/pub/diehard/*, 2008.

[59] P. L'Ecuyer and R. Simard, "TestU01: AC Library for Empirical Testing of Random Number Generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[60] B. Gough, *GNU Scientific Library Reference Manual*. Network Theory Ltd., 2009.

[61] N. A. Group and N. A. G. L. (Oxford), *Fortran Library Manual*. Numerical Algorithms Group, 1988, vol. 3.

[62] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[63] M. E. O'Neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, 2014.

[64] G. E. P. Box and M. E. Muller, "A Note on the Generation of Random Normal Deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 06 1958.

[65] L. Devroye, "Sample-Based Non-Uniform Random Variate Generation," in *Proceedings of the 18th conference on Winter simulation.* ACM, 1986, pp. 260–265.

[66] G. Marsaglia, W. W. Tsang *et al.*, "The Ziggurat Method For Generating Random Variables," *Journal of Statistical Software*, vol. 5, no. 8, pp. 1–7, 2000.

[67] NVidia, "CURAND Library," 2010.

[68] S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer-Verlag London, 2008.

[69] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[70] D. G. Corneil, "Lexicographic Breadth First Search - A Survey," in *International Workshop on Graph-Theoretic Concepts in Computer Science.* Springer, 2004, pp. 1–19.

[71] J. Maia, "Porting Heterogeneous Features into HEP-Frame," Master's thesis, University of Minho, Portugal, 2016.

[72] F. Rademakers, "ROOT — A C++ Framework for Petabyte Data Storage, Statistical Analysis and Visualization," *Computer Physics Communications*, vol. 180, no. 12, pp. 2499 – 2512, 2009.

[73] D. R. Hill, C. Mazel, J. Passerat-Palmbach, and M. K. Traore, “Distribution of Random Streams for Simulation Practitioners,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, pp. 1427–1442, 2013.

[74] M. Saito and M. Matsumoto, “A Deviation of CURAND: Standard Pseudorandom Number Generator in CUDA for GPGPU,” in *Proceedings of 10th International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, 2012.

[75] T. Bradley, J. du Toit, R. Tong, M. Giles, and P. Woodhams, “Parallelization Techniques for Random Number Generators,” in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 231–246.

[76] K. Claessen and M. H. Pałka, “Splittable Pseudorandom Number Generators Using Cryptographic Hashing,” in *ACM SIGPLAN Notices*, vol. 48, no. 12. ACM, 2013, pp. 47–58.

[77] ATLAS Collaboration, “The ATLAS Experiment at the CERN Large Hadron Collider,” *Journal of Instrumentation*, vol. 3, no. 08, p. S08003, 2008.

[78] ——, “Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC,” *Physics Letters B*, vol. 716, no. 1, pp. 1 – 29, 2012.

[79] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Task Scheduling Algorithms for Heterogeneous Processors,” in *Proceedings of the 8th Heterogeneous Computing Workshop*. IEEE, 1999, pp. 3–14.

[80] J. Reinders, *VTune Performance Analyzer Essentials*. Intel Press, 2005.

Appendix A

Installing and Creating an Application With *HEP-Frame*

This appendix describes the steps required for user to download and install *HEP-Frame*, as well as how to create the first pipelined data stream application. The process of compiling and executing an application is also described in detail.

A.1 Installing *HEP-Frame*

HEP-Frame requires that the *BOOST* library, and *ROOT* framework (at least v5.34.34) for compatibility with the high energy physics tools, is installed. Other optional dependencies can be set only when compiling an application in *HEP-Frame*, such as the Intel Math Kernel Library (MKL) and NVidia CUDA toolkit and libraries.

MKL provides several computationally efficient numerical algorithms and functions that can be useful to code propositions in a pipelined data stream application. *HEP-Frame* may internally use MKL to provide the user with efficient Pseudo-Random Number Generators (PRNGs) if the library is available, which is a compute intensive task in most scientific data stream applications (see appendix B). NVidia CUDA is also used to improve the computational efficiency of the *HEP-Frame* PRNG functions, by offloading this intensive task to GPUs through the *cuRAND* library. Use the latest CUDA Toolkit and an adequate CUDA capable

GPU.

HEP-Frame can be installed using Clang, Intel, and GNU compilers, with any version that supports the C++ standard 11. Other compilers may work but were not tested with the framework. The compiler to be used to install *HEP-Frame* and compile an application is set by executing `export HEPF_COMPILER=INTEL/CLANG` in the bash session before the compilation. GNU compiler is used by default if `HEPF_COMPILER` is not explicitly set.

To install *HEP-Frame*, and all standard and field specific tools in the `tools` directory (see subsection 3.2.1 for more details), the user has to execute the `install.sh` script, inside the `scripts` directory, which receives the directory in which *BOOST* is installed as a parameter. If no directory is specified, the installation process will assume that *BOOST* is installed in the system default library and include directories. A sample execution of the installation script is `./install.sh /home/user/boost/directory/`. This links the library core and tools with *BOOST*, compile, and install all tools in the `tools` directory.

The user can update *HEP-Frame* by executing the `update.sh` script. This script requires an internet connection to download the latest version of *HEP-Frame*, and then backups all data stream applications, replaces the old *HEP-Frame* core with the updated version, and automatically repeats the installation process.

A.2 Creating a Pipelined Data Stream Application With *HEP-Frame*

The user should execute the `newAnalysis.sh` script in the `scripts` folder with the following arguments to create a new pipelined data stream application:

`ApplicationName` : the name that the user defines for the application;

`/dir/to/the/rootfile.root` : if *HEP-Frame* is compiled with high energy physics tools, which is used by the `class_generator` tool to create the specification of the class of the dataset element and the code required for file I/O.

This creates a folder named `ApplicationName` in the `Analysis` directory. The created folder holds the makefiles specific to the compilation of this application and the automatically generated data structures and code skeletons in the `src` folder. `src` contains:

`ApplicationName.cxx` : the main skeleton file, containing an `initialise` and `finalise` functions, which can be used for the user to code specific setup and finalisation features to the application, a sample proposition, and the `main` function, where propositions are added to the application and dependencies defined; this file contains a large amount of comments to guide the user step by step;

`ApplicationName.h` : the main header file where the user should declare variables and auxiliary function global to the application;

`ApplicationName_Event.cxx` : contains the initialisation and I/O functions that interact with an input file and build load the data to a dataset element; the user can either write the code of these functions or it can be automatically created by a tool such as the `class_generator`;

`ApplicationName_Event.h` : contains the specification of the dataset element class, i.e., its variable names and types;

`ApplicationName_cfg.cxx` : contains the specification of the variables that must be stored per proposition; will contain the code required to store them in memory at *HEP-Frame* run-time, and the I/O code to write them into `.csv` or `.root` files, which is automatically generated when compiling the application.

It is possible to create auxiliary functions to better organise the code, in the `ApplicationName.cxx` or any other file created by the user. The compilation process will take into account every `.cxx` and `.h` files that are created in the `src` directory. If these functions directly interact with the event information the `unsigned this_event_counter` must be passed as an argument, along with any other arguments that may be required by the user.

A very simple pipelined data stream application in *HEP-Frame* is available at `https://bitbucket.org/ampereira/sampleapplication`.

Compiling the code

The compile process of the *HEP-Frame* library core is separated from the compilation of the pipelined data stream applications, as shown in figure A.1. The user must execute the `make` command on a bash session in the application folder, which compiles the application based on the instructions automatically generated in the `app_Makefile` file. The *HEP-Frame* core is only compiled the first time a given application is compiled, as it is a very time consuming task. The *HEP-Frame* `record_parser` tool parses the dataset element variables that the user indicated to store for each proposition, and generates the required code before compiling the application. This creates `.bak` backup files of the application files in the `src` folder for the user to use as recovery in case something goes wrong.

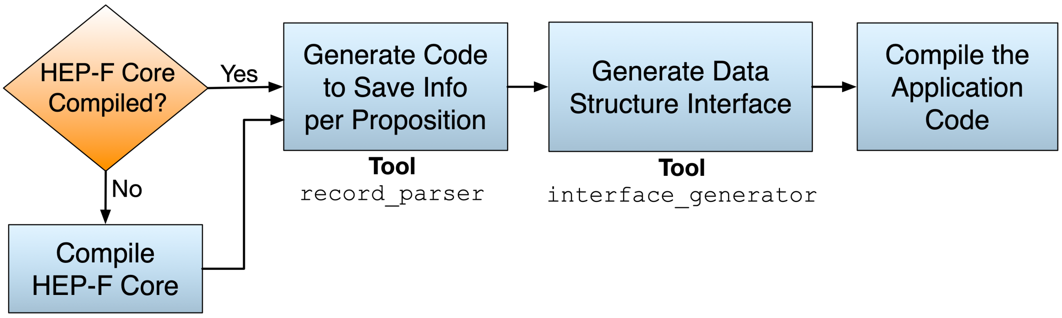

HEP-F Core Compiled?
Yes
No
Generate Code to Save Info per Proposition
Tool
record_parser
Generate Data Structure Interface
Tool
interface_generator
Compile the Application Code
Compile HEP-F Core

Figure A.1: The automatic process of compiling a pipelined data stream application in *HEP-Frame*.

The `interface_generator` tool parses the specification of the class of the dataset elements and creates an defined based interface to abstract the user from interacting with the *HEP-Frame* data structure when coding a proposition. For instance, if a proposition updates the value of the `var` variable of a dataset element, by `var = 2;`, the interface will replace the interaction with this variable by `data_structure[this_event_counter].var = 2;` at compile time. This does not change the original user code.

Other tools can be integrated into the compilation process by editing the `app_Makefile` file in the application directory, under the "Add your tools here" comment.

Specific functionalities of the framework can be configured by bash environment variables. The application and *HEP-Frame* core compilation can be configured by executing the `export VAR=yes` command in the bash session before the compilation, where `VAR` may be:

`HEPF_INTEL` : uses the Intel compiler suite, which often produces better performing code for Intel multicore and manycore devices; also enables the usage of the Intel MKL library.

`HEPF_MPI` : compiles both the application and *HEP-Frame* core to work in a multiprocess environment using MPI (see section 3.2.2 for more information).

`HEPF_KNC` : enables the offload of the `MKLKNC` PRNG to the Intel Knights Corner manycore device; this PRNG cannot be used in the API without this option enabled; use in conjunction with `HEP_INTEL` for the best computational performance and compatibility.

`HEPF_GPU` : enables the offload of the `CURAND` PRNG to a NVidia GPU device; this PRNG cannot be used in the API without this option enabled;

`HEPF_DEBUG` : enables the `-ggdb3` compiler option to provide extra debugging information.

The following options are designed for advanced users with expertise in parallel computing:

`HEPF_THREAD_BALANCE` : enables simultaneous parallel loading and processing of events; automatically adapts the number of threads assigned for input file loading and processing (see subsection 3.2.3 for more information).

`HEPF_SCHEDULER` : enables the advanced parallel scheduler; it does not allow storing information of the dataset elements per proposition; incomplete specification of each proposition dependencies may lead to incorrect results (see subsection 3.2.4 for more information).

`HEPF_AFFINITY` : adequately binds each parallel process and/or thread to the available computing cores; may lead to better computational performance, specially for systems with a non-unified memory architecture.

A.2.1 Pipelined Data Stream Application Execution and Output

The application executable binary is placed inside the `bin` folder of its directory. The application execution can be configured by `export` commands, similarly to the compilation process, and by passing arguments to the executable to override any default configuration. `export`-based configuration must be set before executing the binary using the `export VAR=VALUE` command, where `VAR` may be:

`HEPF_NUM_THREADS` : sets the maximum number of simultaneous threads used during the application execution; by default, this is set to the number of physical cores in the computing system; it is advised to set this to 1 (executing the application sequentially) when debugging the code.

`HEPF_NUM_KNL_THREADS` : sets the maximum number of simultaneous threads used during the application execution in the Intel Knights Landing manycore compute server; this should only be used in a multiprocess environment, where KNL servers are used in conjunction with multicore servers, and the user wants to set different amounts of threads for each device; the number of threads should be set by the `HEPF_NUM_THREADS` variable when only using KNL servers; by default, this is set to the number of physical cores in the computing system.

`HEPF_NUM_PROCESSERS` : sets a fixed number of threads assigned to the dataset processing when used in conjunction with `HEPF_THREAD_BALANCE`; only advised for advanced users.

`HEPF_NUM_READERS` : sets a fixed number of threads assigned to input file reading when used in conjunction with `HEPF_THREAD_BALANCE`; only advised for advanced users.

The application can be executed as `./application arguments`, where `arguments` is a list composed by a pair in the `-option value` format. These options are:

`help` : prints the description of the available options.

`file` : sets the relative or absolute path to the input file to be processed by the application; the usage of this option is mutually exclusive with `directory`.

`directory` : sets the relative or absolute path to a directory with files of the same format to be processed by the application; *HEP-Frame* automatically loads and processes every file in the set directory; the usage of this option is mutually exclusive with `file`.

`record` : sets the name of the output file where the event information stored for each proposition is saved.

`output` : sets the name of the output file and enables saving the dataset elements that pass all propositions.

An application could be executed by the `./analysis -f ../file.root -r rec_vars` command.

Applications compiled with the `HEPF_MPI` option for multiprocess environments must be executed using the `mpirun` wrapper, where the number of processes must be set by the user. However, multiprocess execution can only be used when multiple input files are passed to the application using the `directory` argument. The number of files should be higher than the number of processes used.

The total amount of dataset elements processed and the execution time are displayed when event analyses finish execution. A sample report would be:

```
6327 elements analysed


Number of elements that passed each proposition:


Prop 1:    6001
Prop 2:    5972
Prop 3:    3144
Prop 4:    2718
Prop 5:    2109
```

```
=> The application spent 0.29 secs on input
reading, 24.39 secs on computation and 4.51
secs on element storage
```

Appendix B

HEP-Frame API

HEP-Frame provides a set of features to aid the development of pipelined data stream applications, while ensuring that computationally efficient code is produced. The user can access *BOOST* functionalities by default, and *ROOT* if the high energy physics version of *HEP-Frame* was downloaded, as these libraries are automatically linked with the user code.

B.1 Pseudo-Random Number Generation

A common requirement of scientific data analyses, a subset of pipelined data stream applications, is the need for very large amounts of Pseudo Random Numbers (PRNs). The use of computationally inefficient PRN Generators (PRNGs) may cause significant impact on the computational performance of the applications. *HEP-Frame* offers several PRNGs with either uniform or Gaussian distribution, whose implementations and run-time execution and management are transparent to the user:

TRandom : uses the Mersenne Twister implementation available on the *ROOT* library, which generates a single PRN.

PCG : uses the set of PRNG available in the PCG family of PRNGs. It provides single value and array generation, as implemented in the PCG library.

MKL : uses the Mersenne Twister implementation available at MKL to generate a single value. If a Gaussian distribution is required this implementation uses the Box-Muller algorithm. MKL transparently provides multiple PRN streams to be used in a parallel multithreaded environment.

MKLA1 : uses the Mersenne Twister implementation available at MKL to generate an array of PRNs. HEP-Frame internally manages a dual-buffer for PRNs, where one buffer is simultaneously consumed, PRN by PRN, by multiple processing threads, while the other buffer is being filled. If a Gaussian distribution is required this implementation uses the Box-Muller algorithm.

MKLA2 : similar to *MKLA1*, but each consuming thread has allocated a private subset of the PRN buffer.

MKLA3 : uses the same algorithms as *MKLA1* but implements a single PRN lockfree queue, where the producer pushes batches of PRNs when the amount of PRNs in the queue is lower than a given threshold, and the consumers simultaneously pop PRNs whenever necessary.

CURAND : uses the Mersenne Twister algorithm available in the cuRAND CUDA library, coupled with the Box-Muller algorithm for Gaussian distributions. It uses a similar dual-buffer approach, where a buffer of PRNs is filled in the GPU while the consumer threads use the buffer on the CPU memory. The user needs to execute the command `export HEPF_GPU=yes` on the bash session before compiling the application to use this specific generator.

MKLKNC : uses the same approach as in *GPU*, resorting to the MKL implementation of the PRNG, but it is oriented to the Intel co-processor Xeon Phi KNC.

The PRNG is selected by changing `GENERATOR` to the correct PRNG identifier (as in the list above) in the `rnd.setGenerator(GENERATOR)` instruction, in the main function of the application skeleton file. Each provided generator is best suited for specific PRN intensive applications, so it is advisable to test which PRNG performs better for each individual data stream application on a small set of input data.

The user calls the PRNG by simply typing `rnd.uniform()` or `rnd.gauss()`, which will return an uniformly distributed `double` between 0 and 1 or a `double` in an user defined Gaussian distribution.

HEP-Frame provides the declaration and initialisation code of a PRNG as the `rnd` variable in the user application skeleton files. However, the user can declare more PRNGs by declaring and initialising it in the application, similarly to `rnd`. This may be useful if the user wants to generate PRNs following two different Gaussian distributions, where a PRNG should be declared for each individual distribution.

B.2 Event Loading and Output Storage

The event information is completely managed by the user, which may be stored in any file type. The user must supply the code required to read the information form the file into the *HEP-Frame* data structure. The `class_generator` tool is capable of creating the class specification of the data structure and the code to load the input files for the `.root` format common in high energy physics.

HEP-Frame is capable of analysing and loading any input `.root` file, or a batch of files, without any user interaction. When creating an analysis, the user needs to provide a sample `.root` file with a set of events which will be processed. HEP-Frame automatically analyses the structure of this file to create the required code to read the file, or batch of files, and build a data structure (available to the user) to simultaneously hold in memory the information of all events in the file, when an application is executed.

It is often required in data stream applications that dataset elements that pass all cuts are stored in output files. The code to perform this output must either be provided by the user or automatically generated by `class_generator` for `.root` files. When executing an application the user may specify the `-output` option, detailed in appendix A, and the dataset elements are automatically stored by *HEP-Frame*.

HEP-Frame allows the user to easily store specific dataset element information per proposition. For an application named `AnalysisName`, the name of the dataset element variables has to be inserted in the `AnalysisName_cfg.cxx` file, after the `#ifdef RecordVariables` statement, in the `src` folder of the application directory. The `record_parser` tool generates the required code to store these variables at compile time, for the dataset elements that pass each individual proposition, in an output `.csv` or `.root` file. It is also possible to store variables that were added to the data structure class but are not included in the input file.

Consider a dataset element containing `int var`, `float arr[5]`, `Class cl`, and `vector <double> vec` variables. They can be automatically stored by writing the following code:

`var` : stores the values of this variable.

`arr` : stores the value of every position of this array.

`arr[0]` : stores the value of the position with index 0 of this array; if the index is out of bounds of the array of a given dataset element no value will be stored.

`cl.getValue()` : stores the result of the `getValue()` method; these methods must not require any input parameters.

`vec[1]` : stores the value of the position with index 1 of this vector; it is necessary to always specify an index when storing vectors; if the index is out of bounds of the array of a given event no value will be stored.

It is also possible to store the result of simple equations based on the dataset element variables. The supported type of expressions are:

`var + var` : stores the result of this expression as an `int`.

`var + arr[2] - arr[1]` : stores the result of this expression as a `float` to avoid losing the fractional part of the result, as `arr` type is `float`.

`vec[0] / cl.getValue()` : stores the result of this expression as a `double` to avoid losing precision in the result, as `vec` type is `double`.

The +, −, ∗, /, and % operands are supported. The user must ensure that every index of the arrays and vectors used in these expressions are available, otherwise the application may not store the information properly. The variables in each expression are also individually stored.

B.3 Interaction with External Libraries

HEP-Frame depends on the *BOOST* library, which is automatically linked to each application code. The user can access these library features without manually configuring the compilation process.

Intel Math Kernel Library provides math routines useful for scientific computing specifically optimised for multicore and manycore Intel devices. It includes BLAS, LAPACK, ScaLAPACK, sparse solvers, FFTs, and vector math functions widely used in many scientific computing frameworks. Users can use MKL functions in their application code, if it is already installed on the system, by including the necessary headers in the application `.h` header files. The user should to execute the `export HEPF_MKL=yes` command in the bash session before compiling the application so that *HEP-Frame* automatically links with MKL.

Advanced users can use any external C++ library with *HEP-Frame*. They should include the necessary headers in the application code and edit the `app_Makefile` in the application folder to append both the compiler library linking and library directory flags to the `LIBS` and `INCLUDE` variables.