Computer Physics Communications 263 (2021) 107844

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

HEP-Frame: Improving the efficiency of pipelined data transformation & filtering for scientific analyses☆,☆☆

André Pereira *, Alberto Proença

Algoritmi Center, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal

ARTICLE INFO

Article history:
Received 28 October 2019
Received in revised form 7 December 2020
Accepted 18 January 2021
Available online 28 January 2021

Keywords:
Computational physics
Data analysis
Pipeline reorder
Stream computing
Scientific computing
Heterogeneous computing
High performance computing

ABSTRACT

Software to analyse very large sets of experimental data often relies on a pipeline of irregular computational tasks with decisions to remove irrelevant data from further processing. A user-centred framework was designed and deployed, HEP-Frame, which aids domain experts to develop applications for scientific data analyses and to monitor and control their efficient execution. The key feature of HEP-Frame is the performance portability of the code across different heterogeneous platforms, due to a novel adaptive multi-layer scheduler, seamlessly integrated into the tool, an approach not available in competing frameworks.

The multi-layer scheduler transparently allocates parallel data/tasks across the available heterogeneous resources, dynamically balances threads among data input and computational tasks, adaptively reorders in run-time the parallel execution of the pipeline stages for each data stream, respecting data dependencies, and efficiently manages the execution of library functions in accelerators. Each layer implements a specific scheduling strategy: one balances the execution of the computational stages of the pipeline, distributing the execution of the stages of the same or different dataset elements among the available computing threads; another controls the order of the pipeline stages execution, so that most data is filtered out earlier and later stages execute the computationally heavy tasks; yet another adaptively balances the automatically created threads among data input and the computational tasks, taking into account the requirements of each application.

Simulated data analyses from sensors in the ATLAS Experiment at CERN evaluated the scheduler efficiency, on dual multicore Xeon servers with and without accelerators, and on servers with the many-core Intel KNL. Experimental results show significant improved performance of these data analyses due to HEP-Frame features and the codes scaled well on multiple servers. Results also show the improved HEP-Frame scheduler performance over the key competitor, the HEFT list scheduler.

The best overall performance improvement over a real fine tuned sequential data analysis was impressive in both homogeneous and heterogeneous multicore servers and in many-core servers: 81x faster in the homogeneous 24+24 core Skylake server, 86x faster in the heterogeneous 12+12 core Ivy Bridge server with the Kepler GPU, and 252x faster in the 64-core KNL server.

Program summary

Program Title: HEP-Frame
CPC Library link to program files: https://doi.org/10.17632/m2jwxshtfz.1
Licencing provisions: GPLv3
Programming language: C++.
Supplementary material: The current HEP-Frame public release available at https://bitbucket.org/ampereira/hep-frame/wiki/Home.
Nature of problem: Scientific data analysis applications are often developed to process large amounts of data obtained through experimental measurements or Monte Carlo simulations, aiming to identify patterns in the data or to test and/or validate theories. These large inputs are usually processed by a pipeline of computational tasks that may filter out irrelevant data (a task and its filter is addressed as a proposition in this communication), preventing it from being processed by subsequent tasks in the pipeline.

☆ The review of this paper was arranged by Prof. David W. Walker.

☆☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: ampereira@di.uminho.pt (A. Pereira).

https://doi.org/10.1016/j.cpc.2021.107844

This data filtering, coupled with the fact that propositions may have different computational intensities, contribute to the irregularity of the pipeline execution. This can lead to scientific data analyses I/O-, memory-, or compute-bound performance limitations, depending on the implemented algorithms and input data. To allow scientists to process more data with more accurate results their code and data structures should be optimized for the computing resources they can access. Since the main goal of most scientists is to obtain results relevant to their scientific fields, often within strict deadlines, optimizing the performance of their applications is very time consuming and is usually overlooked. Scientists require a software framework to aid the design and development of efficient applications and to control their parallel execution on distinct computing platforms.

Solution method: This work proposes HEP-Frame, a framework to aid the development and efficient execution of pipelined scientific analysis applications on homogeneous and heterogeneous servers. HEP-Frame is a user-centred framework to aid scientists to develop applications to analyse data from a large number of dataset elements, with a flexible pipeline of propositions. 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.

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; (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. There are also a set of frameworks that merge these two categories (Liu et al., 2015 [1]; Deelman et al., 2015 [2]) for scientific computing. While they do not have steep learning curves, concessions have to be made to their ease of use to allow for their broader scope of targeted applications. HEP-Frame attempts to merge this gap, placing itself between a fully user- or resource-centred framework, 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) is addressed by automatically generating code skeletons and transparently managing the data structure and automating repetitive tasks.

Additional comments: An early stage proof-of-concept was published in a conference proceedings (Pereira et al., 2015). However, the HEP-Frame version presented in this communication only shares a very small portion of the code related to the skeleton generation (less than 5% of the overall code), while the rest of the user interface, multi-layer scheduler, and parallelization strategies were completely redesigned and re-implemented.

1. Introduction

Scientists often develop software to analyse very large sets of experimental data, in a continuous stream of *n-tuples*, aiming to monitor, test and/or prove hypotheses and theories. Most scientific analyses apply a set of pipelined tasks (typically > 10) on independent datasets.

Pipeline stages in scientific analyses 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.

The development and deployment of efficient data analyses on different computing platforms is a highly demanding task for most scientists – due to the complexities of balanced parallel computing on homogeneous and heterogeneous servers – and may have a significant impact on the research productivity of domain experts, since most analyses take a long time to execute.

A new user-centred tool was designed to aid the development of these analyses and to provide an efficient execution engine, the Highly Efficient Pipeline Framework, HEP-Frame.

Previous work addressed three issues related to the design and construction of the HEP-Frame prototype: (i) the specification and development of the prototype and its validation by domain experts [3]; (ii) the performance tuning of pipelined scientific data analyses, where events can be discarded in the middle of the pipeline sequence when they do not comply to a given constraint; this performance tuning stressed the reordering of non-dependent pipeline tasks to apply first those tests that discard more data and to delay those tasks more computationally intensive [4]; and (iii) the use of a multi-buffer approach to speedup the generation of large sets of pseudo-random numbers (PRN), a feature required by most scientific data analyses, using a GPU accelerator [5].

This paper provides a detailed overview and evaluation of an upgraded and fully functional HEP-Frame tool[1] where the domain expert supplies the tasks code and their inter-dependencies, the datasets files and the type and size of the compute server(s); the execution engine automatically reads the data from input streams, builds an adequate data structure, to improve data locality, and runs its key components, a tasks library and the multi-layer scheduler that manages the code execution on a single or multiple compute servers.

The key scientific contribution of the work being reported here is the orchestration of a multi-layer scheduler that adaptively balances the workload of a large set of pipelined scientific analyses on homogeneous/heterogeneous multicore/manycore servers. This scheduler extended the performance improvements of the previous proof-of-concept prototype of HEP-Frame and led to a stable version of the framework, which became a self-contained, integrated, and fully functional tool. This tool was thoroughly validated by domain experts and is currently being used by several high energy physics researchers.

[1] Available at https://bitbucket.org/ampereira/hep-frame/wiki/Home.

Three variants of an actual case study were selected to test and evaluate HEP-Frame on single and multiple heterogeneous compute servers: one memory-bound and two compute-bound versions, one of them requiring a large amount of PRNs. The case study is an application from high energy physics, the $t\bar{t}H$ analysis, where simulated data collected from sensors at the ATLAS Experiment (at CERN) were analysed to prove a theory: the Higgs boson production.

Section 2 gives an overview of the HEP-Frame, from the general structure of pipelined data analyses to a comparative analysis with competing frameworks. Section 3 details the multi-layer scheduler and how it exploits the pipeline parallelism, discussing the related work. Section 4 describes the three variants of a real cased study to test and validate the framework. Section 5 evaluates the overall performance of the HEP-Frame scheduler on different heterogeneous compute servers, discussing the measured results. Section 6 presents a critical analysis of the developed work with suggestions to extend the framework features.

2. Highly efficient pipeline framework

HEP-Frame is a user-centred framework that aims high-performance portability across different types of parallel computing platforms. It provides automatically generated code skeletons, transparently manages the data structure, and manages the efficient execution of the code through a multi-layer scheduler. Often, non-computer scientists store the data element being processed in global memory, replacing it with the next element once the current processing is finished. HEP-Frame keeps this illusion – the user code can access the data element being processed as if it was in global memory – while the framework stores multiple data elements in adequate data structures, transparently to the user. This approach provides a familiar coding environment, ensures that existing code can be easily ported into HEP-Frame, and hides the unnecessary complexity of interacting with specialized data structures.

The extended scheduler version includes the following features:

- it balances 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;
- it adaptively reorders the pipeline execution flow and the tasks distribution across heterogeneous computing resources, exploiting parallel execution of independent pipeline stages (task parallelism) with multiple dataset elements/input streams (data parallelism);
- it balances the data and workloads distribution on a heterogeneous multicore server with accelerators, such as the manycore co-processor Intel Xeon Phi Knights Corner (KNC) or GPU devices, and on multiple multicore and/or manycore servers (with Xeon Phi Knights Landing, KNL).

Users have control of their code, which is not modified by the framework, and specify the dependencies among the pipeline stages, so that the HEP-Frame scheduler ensures the application correctness during its parallel execution.

Next subsections address the structure of a typical flexible pipelined scientific data analysis, the HEP-Frame usability (the skeleton generator and associated pipeline inter-dependencies, the data structure setup, plug-ins to store results), and a comparative analysis of the framework competitors.

2.1. Flexible pipelined data stream analyses

A scientific data analysis is a process 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.

HEP-Frame supports three modes to input data into the pipeline: batch or mini-batch from files, and streaming from external sources.

The batch input loads and pre-processes a pre-defined batch of data (usually an input data file) before this chunk is available to be processed by the application pipeline. This is the strategy most commonly used in scientific computing, where the data from a whole file is read and prepared before being processed, but it means that concurrent input reading, which often requires an initial data setup phase (input reading and setup are addressed as *DS*), and the processing of the data (data processing phase is addressed as *DP*) only makes sense with multiple batches.

The mini-batch input loads and pre-processes each data tuple, or a small set of tuples, individually from an input batch: data is available earlier to be processed by the application pipeline. By definition, a batch or mini-batch entails the loading of a set of information, which cannot be processed before its *DS* phase is finished. If an input file can be divided into mini-batches, it is possible to start processing data from a file, or multiple files, before the whole file is read, so *DS* and *DP* tasks can be concurrently processed for each data tuple.

Streaming continuously loads data tuples from a given input descriptor, similarly to mini-batch, until it is signalled to stop. The *DS* and *DP* tasks can be simultaneously executed, but this requires careful memory management to avoid exceeding the size of the available physical memory.

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 work 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. Fig. 1 shows the flowchart of a typical structure.

Scientific data analyses 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, if the code is more memory-bound than compute-bound. The HEP-Frame scheduler dynamically employs different techniques to get the best performance of both memory and compute bound irregular workloads.

The default order of the pipelined propositions, as defined by the domain expert when developing the application, is not guaranteed to be the most efficient, as 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. Possible execution orders of propositions, which respect their dependencies, can only be represented as a directed cyclic graph. A formalization of a method to structure the pipeline order execution for maximum efficiency is described in depth in [3].

Several scientific fields require pipelined applications structured as detailed above: particle physics data analyses, which mostly filter and process data measured from sensors; cosmology analyses, where it is often required to find objects with certain characteristics, discarding most of the gathered data; and optimizing queries on databases and data streams.

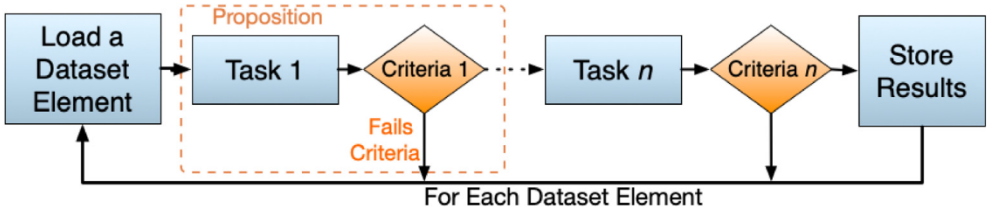

Fig. 1. Structure of a typical flexible pipelined scientific data analysis.

2.2. HEP-frame usability

The HEP-Frame run-time system handles all repetitive tasks usually performed during an analysis code execution, while requiring the user to provide code snippets to fill the remaining gaps, as shown in Fig. 2.

HEP-Frame provides a toolset with scripts for the development and compilation of pipelined scientific code. Four tools are available in the current version of HEP-Frame: `skeleton_generator`, `class_generator`, `record_parser` and `interface_generator`. The HEP-Frame can also include tools to automatically and transparently create the load/store code for a specific case study (the green boxes in Fig. 2). Additional tools can be coded and added by the domain expert, like plugins, so that other specific needs of their pipelined skeleton can be addressed.

The `skeleton_generator` tool creates a skeleton with function prototypes for the user to fill in with the required code to run the application, such as propositions and their interdependencies, dataset file loading, dataset element class structure, and result storage.

Domain experts code their analysis stages in the propositions, which HEP-Frame will manage as black boxes without modifying the code, to ensure that users trust its correctness and have total control of the code. This also allows their code to be easily updated and expanded, while working out-of-the-box with updated versions of HEP-Frame. Users can also organize their propositions and auxiliary functions in multiple source files, as HEP-Frame will automatically detect and compile them.

Users must add the propositions to the analysis in the `main` function in the skeleton file, by calling a method that receives the proposition function pointer and a 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 through a method that indicates that a given proposition depends on another one. The HEP-Frame scheduler balances proposition execution without compromising the correctness of the results, based on these user-defined dependencies. Partial and final results can also be stored: the user provides the code in a specific function on the skeleton file.

The `class_generator` automatically creates the dataset structure and the code to load the data from an input data file. This tool is automatically called when a domain expert creates a new analysis with HEP-Frame.

Users specify the dataset element variables to be stored, for the elements that pass each proposition or the whole pipeline, by indicating their name on a specific section of the skeleton file. The code to store these variables for each dataset element is transparently created by the `record_parser` tool when the user compiles the analysis.

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

A proposition receives only a counter parameter as input, managed by HEP-Frame, and returns a Boolean to indicate if the current dataset element passes to the next proposition or is filtered out. For instance,

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

int main (void) {
    // ...
    analysis.addProposition (prop1, "prop1");
    analysis.run();
    // ...
}
```

`val1` and `val2` are variables of a dataset element, which can be accessed as if they are declared in global memory. The `interface_generator` translates the access to these variables into accesses to the HEP-Frame data structure that holds all dataset elements. This proposition will be applied to all dataset elements, according to the scheduler assessment, similarly to kernels in CUDA.

2.3. Competing frameworks

HEP-Frame has a two-fold goal of (i) being a user-centred framework to aid the development of scientific code and (ii) managing the efficient execution of the code on multicore and manycore servers. Several popular and widespread frameworks aim to aid the development of parallel code, but few address the domain-experts usability requirements, and even fewer address the transparent performance portability across different servers. Flink [6], Storm [7], and Spark [8] loosely fit in this category, but they do not meet the expectations of domain experts, and do not allow for the level of control that HEP-Frame has over this specific type of scientific applications.

Very few frameworks address transparent performance portability and at the same time are oriented to domain experts, particularly non-computer scientists [1]. StarPU [9] and Legion [10] 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 programmers and lack support for flexible pipelines, namely those that may discard dataset elements in intermediate pipeline stages.

StarPU provides a refinement of this strategy by using task priority information to select which task to process among all tasks in a ready queue. However, since each task priority must be defined and updated by the user during the application execution, this requires that the user must have a comprehensive knowledge of the problem, to define an adequate heuristic for an efficient application code execution.

Legion focuses on proper structuring and efficient placement of the data on the complex memory hierarchies of heterogeneous

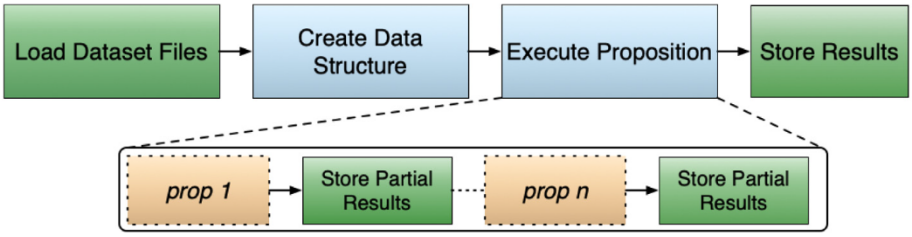

Fig. 2. Execution flow with the HEP-Frame: the user provides code for the darker boxes (orange and green) and the framework run-time system manages the blue boxes. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

servers. Users specify properties of data structures, such as partitioning strategies and coherence, which allow it to implicitly extract parallelism and manage data. Legion allows user defined mapping of both tasks and data to specific computing devices, while respecting the properties of the datasets. However, this tool uses generic schedulers for irregular tasks (that can be extended by the user), which may not be suited for every type of applications. Applications with simple data dependencies or heavily compute-bound are not the focus of Legion.

Other simpler tools are available to specific balance the workload of irregular pipelined code, but do not provide multiple scheduling options and other features that are provided by StarPU and Legion, and they also lack support for flexible pipelines, such as Pegasus [2]. OmpSs [11] and DAGuE [12] are the alternative tools that may be best suited for pipelined scientific code.

OmpSs is a *pragma* based programming model that extends OpenMP [13] to support asynchronous parallelism and offload to accelerator devices, which allows easy integration in existing codes by users with some experience in parallel programming. It is based on a *task* clause that defines the parallel region, in which data dependencies and transfers to and from accelerator devices are specified. It supports OpenCL and CUDA capable devices, as well as the Intel KNC co-processor. However, OmpSs is limited as its parallelization needs to be tailored for the user code and requires the tasks to be compatible with the accelerator devices, which is often not the case, 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 acyclic 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 work stealing strategy. However, flexible pipelined data analyses tasks can only be represented by directed cyclic graphs, as described in Section 4.

A data processing framework that aims to ease the development of physics scientific data analyses is ROOT [14]: it is used by a wide scientific community, namely at CERN, and it provides an extensive set of features for I/O, physics, statistical analysis tools, and even application skeleton generators. However, the performance of these features was not the priority of ROOT developers, which leads to significant bottlenecks in code execution [15].

The two types of competing frameworks presented, StarPU/Legion and ROOT, stand in opposite ends: the former is performance oriented with a steep learning curve for non-computer scientists, while the latter is feature oriented and easy to use, but computationally inefficient. Performance oriented frameworks usually have steep learning curves, while feature oriented frameworks are easier to use but often computationally inefficient.

While the original case studies were sequential and heavily based on ROOT, the performance improvements presented in this communication are speedups over improved and multithreaded versions of the case studies, which are also less dependent on ROOT (see [15] for a comparison of the original case studies).

To develop an application with StarPU is a hard task for domain experts: StarPU imposes several coding restrictions and offers a large set of configuration options that are not clear for non-computer scientists. All published papers that describe applications developed with StarPU were written by computer scientists. However, StarPU can generate efficient code through a variant of an adequate scheduling approach, which will be used in this paper as the reference scheduler to show the efficiency improvements of HEP-Frame. The case studies used to evaluate the HEP-Frame performance were not ported into StarPU by their original particle physicists developers, as they lacked the expertise and time to perform an adequate implementation.

In between these two types of competing frameworks lies HEP-Frame, user-centred to be easily adopted by non-computer scientists: it includes the relevant functionalities of ROOT and at the same time it efficiently uses the available computing resources.

3. The multi-layer scheduler

A proof-of-concept single-layer scheduler for compute-bound pipelined scientific data analyses was developed for multicore environments, to assess the feasibility of the proposed novel scheduling strategy [4].

This scheduler was extended with multiple layers, each specialized for a specific task. The top layer distributes the workload among the available servers in a distributed environment (cluster or grid/cloud), while the other two layers operate at the server level, with scheduling mechanisms to manage compute- and memory-bound codes on multicore and manycore servers with accelerators.

The key distinction between multicore and manycore servers in the context of this work is not only related to number of cores in the chip (or package of multiple dies), but also mainly due to the trade-offs the designers had to follow to fit in so many cores, namely by removing the shared L3 cache. As for the accelerators, these are usually optimized for a given set of operations rather than be an optimized version of a general purpose CPU, so in the context of this work HEP-Frame is highly effective to take advantage of these devices to speedup specific time consuming tasks.

The scheduler design and implementation are independent from HEP-Frame, so that it can be integrated into other tools or directly into the user code. The amount of propositions and their dependencies are fed to the scheduler, during its initialization. Computing threads only have to request the next task to process, which consists of a combination of a proposition and a dataset element, and report back to the scheduler if the proposition passed or failed before requesting the next task. The stand-alone multi-layer scheduler is also available for download.[2] The multi-layer scheduler will be presented through the next subsections within the context of the HEP-Frame tool, where the detailed scheduler features are self-contained and independent from any external code.

[2] https://bitbucket.org/ampereira/hep-frame/src/master/

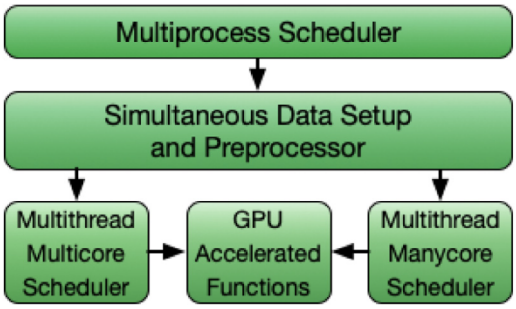

Fig. 3. Multi-layer structure of the HEP-Frame scheduler.

3.1. Structure of the scheduler layers

The HEP-Frame scheduler is currently structured in three layers, as shown in Fig. 3, to efficiently use the available computational resources in multiple multicore and manycore devices.

The top layer 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. Preliminary tests, using up to 6 multicore and manycore servers, showed that this approach leads to an efficient task balance with minimal overhead.

The main focus of the HEP-Frame scheduler is on desktop servers and mini-clusters, since 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 a larger cluster server.

The middle layer implements a multithreaded file/data-stream reading with the data structure creation and pre-processing, the data setup (*DS*), in parallel with the pipeline execution, the data processing (*DP*). This layer is responsible for the management of the amount of threads dedicated to *DS* and *DP* tasks, adapting each amount at runtime to the needs of the application. The number of parallel threads allocated to each component (*DS* and *DP*) is adjusted during the overall execution of the application, to adapt to memory or compute bound code.

The bottom layer implements parallel data processing for multicore or manycore environments, which will mostly improve the performance of compute bound code, where propositions of the same or different dataset elements are concurrently executed, prioritizing the execution of faster propositions that filter more data, while respecting dependencies among propositions in the pipeline. This pipeline order is periodically updated, as described in [4], according to changes in the computational characteristics of the propositions.

The execution time and the amount of data filtered out by each proposition may change at run-time due to (i) the values of the dataset elements that may specify different operations to perform, each with a different computational impact, and (ii) the relative position of the propositions in the pipeline, as some may filter out data at the beginning of the pipeline that will not be later processed by compute intensive propositions. An adaptive scheduling strategy is required in this layer since the execution time and filtering ratios of propositions may significantly vary during run-time.

This layer also manages the distribution of the workload in a Xeon compute server coupled with manycore KNC co-processors. A demand-driven strategy is used, where the KNC devices request from the Xeon PU data chunks to process.

HEP-Frame also provides a set of GPU accelerated functions to generate large sets of pseudo-random numbers (PRNs). These are usually required by most data analysis applications, where PRN generation may take a significant slice of the application execution time. Offloading this task to an accelerator allows faster generation of PRNs and frees up resources in the multicore/manycore devices, which HEP-Frame uses to compute other parts of the application. The user interacts with the HEP-Frame PRNG API as regular multicore-based code, while the efficient PRN generation and management is completely transparent and is supervised by HEP-Frame.

3.2. Dynamic tuning of data setup and processing

This HEP-Frame core scheduling layer manages the parallel execution of the data read and setup task (the *DS*) and the pipeline data processing task (the *DP*), ensuring that both compute- and memory-bound codes are efficiently executed (most frameworks do not support this capability). This balance is achieved when a *DS* task finishes close to a new *DP* task requesting more data to process, and it not only minimizes the overall execution time, but also the size of used memory, since input data is immediately consumed. For each supported mode to input data into the pipeline (batch/mini-batch from files, and streaming, detailed in Section 2.1) the user must provide the code to load a single data element from the chosen input type.

This scheduler layer creates one *DS* thread and one *DP* thread per physical multicore PU core, but at any time it only activates one thread. It periodically allocates, in run-time, the number of threads to *DS* and to *DP*, avoiding the performance penalty of creating and destroying threads at each core: it simply switches on the required *DS* or *DP* thread and puts asleep the other one. 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, when compared to a single thread that switches between both tasks.

The following heuristic leverages the amount of threads for the *DS* tasks and for the *DP* tasks:

- define the time between checkpoints to adjust the number of *DS* threads (*DSt*) and the number of *DP* threads (*DPt*) as the time to read and process a chunk of dataset elements in the file (default value: a pre-defined size);
- activate the same amount of threads for *DSt* and *DPt* as the initial default value;
- at each checkpoint dynamically tune the number of threads for each task;
- allocate all threads to *DP* once the *DS* is complete.

To tune the number of threads for each task at each checkpoint, the scheduler follows these steps:

- measure the execution times for *DS* and *DP* tasks (DSt_{time} and DPt_{time}) to process the pre-defined chunk of dataset elements in the file;
- compute the time impact of each thread, by dividing the chunk execution time by the amount of threads used, for both data setup (DSt_{time}) and processing (DPt_{time});
- when the *DP* task takes longer than the *DS* one, allocate more threads for the DP tasks, and vice-versa; if both *DS* and *DP* execution times are similar, set a balanced amount of threads to *DS* and *DP*;
- compute the number of threads (nt) that should be shifted from setup to processing and vice-versa, according to Eq. (1) (if $DS_t \leq DP_t$) and 2 (if $DP_t < DS_t$);

$$DS_t - nt * DSt_{time} = \frac{DS_t + DP_t}{2} \tag{1}$$

$$DP_t - nt * DPt_{time} = \frac{DS_t + DP_t}{2} \tag{2}$$

- double the time between checkpoints when *nt* is zero, which means no change is necessary, in two consecutive checkpoints, to reduce the scheduler overhead;
- double the checkpoint rate, when *nt* values are different from zero in two consecutive checkpoints, to deal with the dataset irregularity.

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 possible amount of *DP* threads to not wait for data to be loaded.

The limitations of memory-bound code are usually due to accessing data in RAM at a slower rate than its processing, which can be due to either the latency of the RAM devices or the memory channel bandwidth. However, if the required amount of data is larger than the RAM size or is not yet in RAM, then the data access bottleneck moves to the data transfer from an HDD/SSD into RAM, which becomes an I/O-bound code. HEP-Frame addresses this latter view on memory-bound code, since the scheduler layer balances the threads between *DS* (which contain both data reading and initial setup) and *DP* tasks. 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.

HEP-Frame may also improve the performance of I/O-bound code using an input data stream from an external source, provided that the limitation is not the stream bandwidth.

Preliminary experimental results showed that there are no significant benefits of using simultaneous multithreading (Intel Hyper-Threading) in both compute- and memory-bound codes. No results are shown to avoid littering this communication with less relevant data, since it is common knowledge among the computer science community that using Intel Hyper-Threading seldom leads to performance improvement.

Current memory monitoring system in HEP-Frame only frees the whole data structure in memory 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 input stream the *DP* threads are put asleep.

3.3. Pipeline ordering and parallel execution in a multicore server

The end user suggests an execution order of the pipeline propositions, but it may not be computationally efficient. Reordering the pipeline often leads to a faster analysis execution, while respecting their inter-dependencies. 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 computed by the heavier propositions, reducing the overall application execution time.

Propositions execution flow can only be defined by a directed cyclic graph, unlike most list scheduling algorithms: in the absence of dependencies, $prop_0$ can be executed before $prop_1$ and vice-versa, which is represented by a bidirectional edge allowing cycles to be present on the graph. Most list schedulers cannot be used due to this property, and more computational power is required to find the best list order than on acyclic graphs.

The HEP-Frame scheduler takes a simpler alternative approach since it does not need to follow a strict propositions order in a parallel environment: it creates a graph, where a direction edge between two nodes (propositions) represents a dependency, previously defined by the user. If $prop_0$ needs to be executed before $prop_1$ there is only a directional edge from $prop_0$ to $prop_1$. Then the Breadth-First Search (BFS) algorithm [16] is used to compute a list of all paths in the graph with directional edges, which corresponds to the list of dependencies among propositions. BFS has a complexity of $O(N^2)$, but preliminary tests showed that its execution time was negligible, as applications usually have a small amount of different tasks for it to be a limiting factor. For instance, the case studies presented in Section 4 only have 18 propositions and the graph only has 33 edges. This implementation is further detailed in [4].

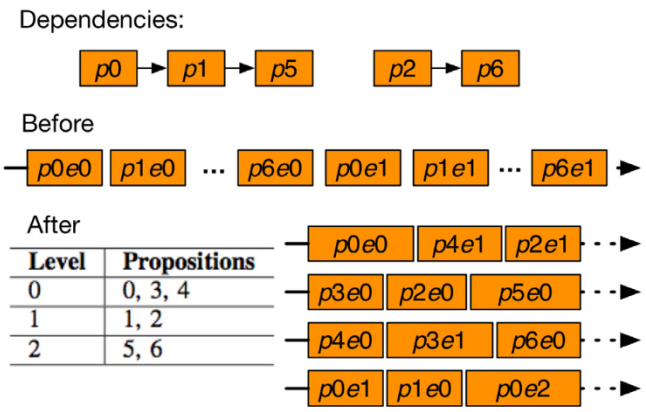

Fig. 4. Sample pipeline execution with the HEP-Frame scheduler.

A table is built with n levels, where n is the amount of propositions in the longest dependency chain. Fig. 4 illustrates the parallel execution of a pipeline on a 4-core server, with 7 propositions ($p0 \ldots p6$) for various dataset elements ($e0, e1 \ldots$), where the longer dependency chain has 3 propositions. The scheduler assigns propositions from a given table level to the threads to process, but does not assign a proposition of the next level until all propositions on the current level are processed.

Since dependent propositions are on different levels, this mechanism ensures that all dependencies are respected. If a proposition criteria evaluation fails, or if there are more threads than propositions within the current level, the scheduler starts assigning propositions of the next dataset element.

Propositions are ordered according to their weight within each level; those that weight less will be processed earlier. This weight is calculated based on the ratio of discarded dataset elements and its execution time. The default values are set to 70% for the former and 30% for the latter, which is a heuristic based on extensive testing with various case studies, but can be tuned by the users.

Propositions can also be moved between table levels: propositions with long execution times and that filter out few data are processed later in the pipeline. These propositions may not have dependencies, but moving them to another level introduces an artificial dependency, which acts as a barrier. This allows lighter propositions, which filter out a larger amount of dataset elements, to be processed earlier in the pipeline. It ensures that there is more data that is filtered out by these faster propositions, and fewer data reaches the heavier propositions. It means that originally the heavier propositions were being processed on data that was then filtered out, and now only data that passes the faster propositions are processed by these heavier propositions. Overall, these heavier propositions are processed less often than originally, reducing the execution time of the pipeline (detailed in [4]).

Fig. 5 compares a traditional list scheduler with the HEP-Frame scheduler for a case study with 4 independent propositions ($p0 \ldots p3$) and several dataset elements e, on a 4-core server.

In this illustrative case study it is assumed that proposition $p2$ filters out dataset element eN (red box) and proposition $p3$ takes significantly longer to execute than the remaining propositions. HEP-Frame automatically measured this information about these

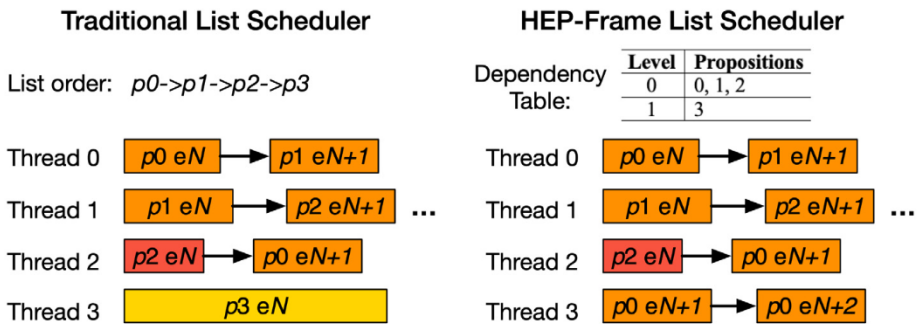

Fig. 5. Traditional list scheduler *vs.* HEP-Frame list scheduler. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

propositions during the execution of the first *N-1* dataset elements, since the programmer seldom has any knowledge about the proposition computationally characteristics.

In a traditional list scheduler a weight is assigned to each proposition, which can be similar to the HEP-Frame approach, causing propositions to be ordered in a list that is used to feed subsequent propositions to computing threads. This approach takes into account the filtering of data in the weight calculation, but not when executing the proposition. This characteristic, allied to the absence of barriers, means that propositions of a given event can only be executed sequentially as dependencies cannot be assured otherwise, allowing only for data-level parallelism. Moreover, the absence of barriers does not allow for a more strict reordering of propositions that HEP-Frame uses.

The use of barriers between levels of the table ensures that (i) selected groups of propositions can be simultaneously executed on the same or on different dataset elements and (ii) there is a higher probability of an element being filtered out before executing a heavy proposition, reducing the overall execution time of the pipeline.

A traditional list scheduler would schedule each proposition $p0$ 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 $p2$ failure.

The HEP-Frame scheduler introduced an artificial dependency based on $p3$ weight, which forced $p3$ to be executed after all other propositions, so $p0$ for the next dataset element was assigned to the last thread. This avoided unnecessary computations, as $p3$ was never executed for eN. This strategy may provide significant performance improvements for propositions that take considerably longer to execute than others (sometimes by a factor of 10^6, as are the case studies in Section 4, used to validate this work).

An improvement of the traditional list scheduler could consider a task as a combination of the proposition and the dataset element, so that it could execute the $p3$ of several e much later than the $p0$-2 of those e. However, separating their execution by a significant amount of time leads to a poorer use of the spatial and the temporal locality of the dataset, as an element would be partially processed by some propositions and then processed again some time later. For instance, when $p3$ is executed, this approach requires to reload into cache data that was already previously there, resulting in a higher cache miss rate than processing $p3$ as soon as the dataset element is not filtered out by the previous propositions. An efficient use of the cache is crucial to ensure the efficient use of the available computational resources, leading to a faster execution time of the pipeline over the improved list scheduler. The HEP-Frame scheduler takes advantage of the spatial and temporal locality of the data, as it schedules $p3$ for execution as soon as $p0$-2 finish.

Traditional list schedulers are extremely efficient at managing pipelines of tasks that do not filter out dataset elements, but are not designed to schedule propositions. As stated before, their lack of barriers, and thus task-level parallelism of a single or multiple dataset elements does not ensure the most efficient balance of data and propositions among threads. This may lead to the unnecessary execution of computationally intensive propositions, such as $p3$ in Fig. 5. This novel strategy of simultaneously processing propositions of the same and different dataset elements, while reordering and respecting proposition dependencies, reduces the computational load and leads to a faster adaptation to irregular workloads, thus reducing the overall execution time compared to alternative approaches.

3.4. Pipeline ordering and parallel execution in a manycore server

The scheduling algorithm was tweaked to be able to explore the KNL massive parallelism potential, since most manycore devices lack a cache level. Reordering pipelines with complex dependencies requires frequent scheduler synchronizations, which may limit performance when large amounts of threads are running (up to 256 on KNL), due to the missing L3 cache.

Each thread on the KNL computes the whole pipeline of a predefined data chunk size, 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 are still measured for each pipeline stage, as it will still contribute to the reordering of the pipeline. After processing a data chunk, an average of the normalized measured values of all threads for each proposition is computed, which attributes a global weight to each proposition.

A directed cyclic graph is built, where each node 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. This list scheduling approach is different than the one presented in Section 3.3, since the use of barriers has an adverse impact on KNL devices due to the large amount of threads being executed (usually around 250), slower clock rate, lack of L3 cache and costly communications among cores.

The best pipeline order is obtained by computing the shortest path that passes through all nodes (propositions) of the graph using a recursive backtracking algorithm. The shortest path, i.e., the path with the least weight, is computed for each node that can be used as the beginning of a path on the graph. From these paths, only the shortest is considered, which is used as the best pipeline order at the moment. The backtracking algorithm to find the shortest path for a given starting node is shown in Fig. 6 and is further detailed in [3].

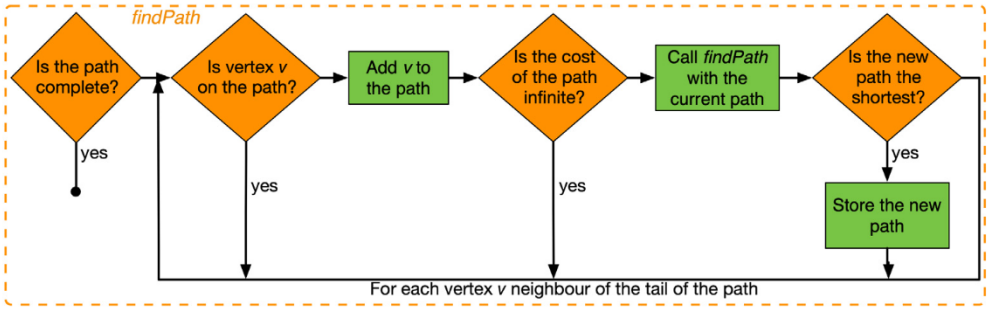

Fig. 6. Scheduler pipeline reordering backtracking algorithm for the KNL server.

Preliminary tests showed that the overhead of this algorithm is less than 1% of the case studies presented in Section 4, since it is only computed for a small set of starting nodes, and these graphs usually have a small number of nodes (18 for these case studies).

The pipeline is reordered at given checkpoints through the application execution, as shown in Fig. 7.

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 Fig. 7). 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.

3.5. Offload into accelerators

Accelerator devices act as co-processors to improve the performance of scientific code using one of two alternative approaches: (i) to offload a significant portion of the code, ideally processing it simultaneously with the multicore devices, or (ii) to offload specific sections of code more suitable for the device, which may account for a significant portion of the overall execution time. HEP-Frame can use both offload strategies: it is currently compatible with Intel KNC co-processor, Intel KNL manycore server and NVidia GPU accelerators.

The KNC device poses some limitations to the user:

- it requires to explicitly transfer the required data for the propositions from the multicore memory to the manycore memory, forcing the user to use simple data structures (the same applies to the use of GPUs); developing the code to transfer complex data structures, based on containers, classes, and pointers, requires an expertise that most domain experts lack; tools to automate the generation of the code to move data between multicore and KNC devices did not consistently provide working implementations;
- the libraries required by the propositions must be specifically compiled to the device, which is often not feasible due to compatibility issues.

An HEP-Frame proof-of-concept prototype was developed to assess the feasibility of using KNC devices using approach (i). Offloading propositions required the user to develop the code to transfer the required data and to modify the proposition code. The use of this accelerator only provided a 10% performance improvement in the best case (using the Ivy Bridge server described in Section 5.1, with further details of the case studies in Section 4), while limiting the performance of most compute- and memory-bound codes. Since most domain experts lack know-how to develop efficient code for this device and it only has the potential to provide a small performance improvement, this prototype was discarded in the final version of HEP-Frame.

GPU use a different programming model from regular multicore code, on top of the same limitations of the KNC device. Most libraries are not available in CUDA or OpenCL, and porting these into the accelerator can be unfeasible. Therefore, it may not be possible to use the offload approach (i), since most propositions use libraries that were not ported for GPUs.

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, PRN generator (PRNG) functions are supported, but more APIs can be added on user feedback.

HEP-Frame is able to run 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 scheduler in Section 3.3, which is presented in Section 3.4. 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 offloading the whole pipeline execution.

3.5.1. Offloading pseudo-random number generation to accelerators

The current version of HEP-Frame supports several types of PRNGs (available in MKL [17], ROOT [14] and PCG [18]), some of which offloaded to NVidia GPUs, using the CUDA cuRand library, and to the KNC, using the MKL library. The user can choose one out of several PRNGs, all providing sets of values with both uniform and Gaussian distributions:

- a simple PRNG, which uses the compiler default or the MKL if available;
- a multithreaded PRNG, for PRN intensive applications, which transparently fills a buffer available to the user;
- a PRNG on GPU, which is detailed in this subsection.

Scientific code requires a high quality PRNG, with a very large period and a very small correlation between PRNs. Mersenne Twister [19] is one of the most used PRNG, and is currently the PRNG that is offloaded to GPUs and KNCs on HEP-Frame. The cuRAND NVidia library provides an implementation that produces a single PRN, which is encapsulated by a kernel in HEP-Frame to produce a large set of values to efficiently use the available GPUs. The MKL provides an implementation that generates a batch of PRNs, which was integrated into a function that offloads it to KNCs.

A thread is created on the multicore device for each computing thread to manage a fixed size dual-buffer for PRNs. The first buffer is filled with PRNs generated by the accelerator when HEP-Frame is loading and creating the dataset structure, and every PRN requested by the code is returned from the buffer. When the

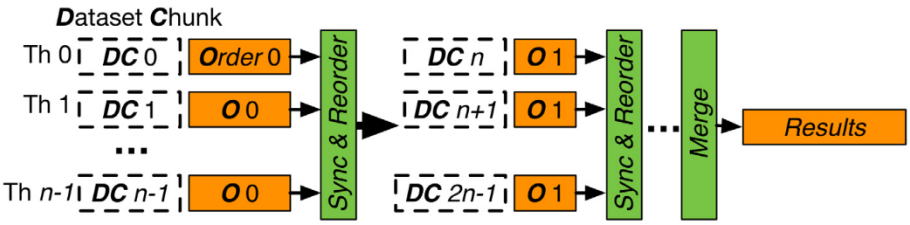

Fig. 7. Parallel execution of the pipeline in the KNL server for n threads (*Th*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

buffer is almost full, the managing thread instructs the accelerator to generate a new set of PRNs to fill the other buffer, while the first buffer is still being consumed. Buffers are swapped once one of them is depleted.

Using two buffers per compute thread hides the PCI-Express bottleneck and ensures that the code has always available PRNs and is not locked waiting for the accelerator to generate them. The buffer size can be tuned by the user if needed. Each managing thread launches the PRNG kernel on the GPU and transfers the required data through an individual CUDA stream, so that multiple managing threads can simultaneously issue PRNG requests. The KNC implementation uses a similar strategy, with individual memory transfer streams for each managing thread. The dual buffer approach provided speedups up to 71x on a dual-socket server, using the Pascal GPU. A detailed performance evaluation of this approach with multiple GPU, manycore, and multicore PRNG acceleration is available in [5].

3.6. Related scheduling work

Scheduling pipelined applications into available parallel resources has been addressed in list schedulers, being traditional list scheduling a well studied subject. The Heterogeneous Earliest Finish Time (HEFT) [20] is one of the earliest and most relevant heuristics to schedule tasks on heterogeneous processing units, where tasks are assigned based on their priority and interdependencies. The priority of a task is computed based on its execution time and data transfer costs. Some relevant scheduling works deserve an additional note and comment:

- in [21] the authors reorder the pipeline execution of database queries, where queries that produce relevant tuples are processed first; however, this approach does not address the efficiency of the pipeline execution;
- in [22] the authors present a scheduler for pipelined streams that reorders simple filters at run-time, which may seem similar to the *Multithreaded Multicore Process Scheduler* layer of the HEP-Frame scheduler; however, when reordering filters, it does not take into account that filters may be irregular and computationally complex nor the potential data dependencies among filters;
- in [23] the authors propose a scheduler that mixes streaming task and data parallelism, but do not consider reordering nor inter-task dependencies, assuming that all tasks are always executed, none is filtered out;
- in [24] the authors propose a *Predict Earliest Finish Time* algorithm for heterogeneous systems, but it only schedules tasks (it does not take into account the dataset) and assumes that all tasks are executed, disregarding the optimization potential of conditional task stages;
- in [25] the authors propose a programming model to schedule irregular workloads in stream applications; it supports pipelines represented as cyclic graphs but does not consider that they may filter out data and does not parallelizes nor reorganizes the pipeline tasks;
- in [26] the authors propose a set of algorithms for multicore list scheduling but assume that each individual task is already parallelized, which makes the users accountable for the efficiency of their parallel code;
- in [27] the authors present StreamBox, an out-of-order data parallel engine with parallel execution of pipelines on data streams; although it may seem very similar to the HEP-Frame scheduler, it has severe limitations, as described below.

The authors of StreamBox claim that its engine exploits both data and out-of-order pipeline parallelism. However, this tool displays yet a set of limitations that makes it unsuitable for the type of data analyses this paper 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 in parallel data inside data batches;
- StreamBox assumes that all tasks in a pipeline are executed and no data is filtered out, while HEP-Frame efficiently addresses the propositions workload by removing from the pipeline the tasks that failed one of the followup criteria;
- StreamBox focus on scheduling according to the characteristics of the input data, assuming that the time to process the pipeline is negligible; most scientific data analyses 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 synchronize, as well as data reordering based on time stamps, which is out of the context of the target applications for HEP-Frame.

HEP-Frame simultaneously exploits both data and out-of-order pipeline parallelism, where multiple data tuples are concurrently processed, as well as multiple propositions of a single data tuple, improving both latency and computation throughput.

HEP-Frame tackles applications with pipelines whose propositions cannot be defined by directed graphs, as their position in the pipeline may change during the application execution. However, most recent work focus on scheduling tasks on heterogeneous systems that can be represented as directed acyclic graphs, such as [24,26,28], which cannot be used for the applications that this work aims to improve, as shown in Section 3.3.

There is no approach in the literature that simultaneously deals with pipeline reordering and proposition parallelization for irregular workloads on heterogeneous servers, which is the key research topic that the HEP-Frame scheduler targets. The presented parallel list schedulers that reorder the pipeline apply it atomically to different data, while HEP-Frame implements reordering and parallel execution of propositions inside the pipeline, to the same or different dataset elements, as detailed in Section 3.3.

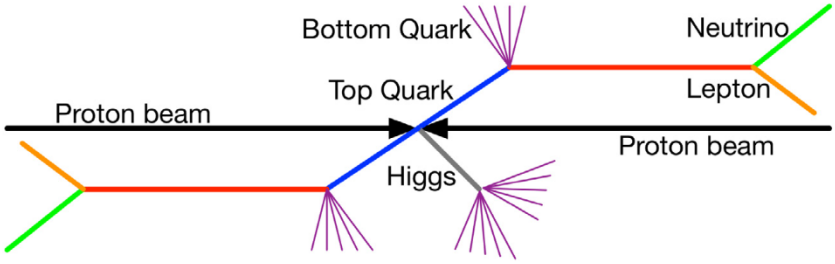

Fig. 8. Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

4. The case studies

To evaluate the performance of the HEP-Frame scheduler, the case studies should ideally have the following features:

- be representative of one or more classes of scientific code;
- be actual code developed by non-computer scientists;
- contain the key features of a pipelined data stream analysis as described earlier, namely a very large set of streaming *n-tuple* input data, over 10 processing pipeline stages, some with commutative tasks and/or heavy and irregular computational tasks, and that may filter out data;
- include variants of the same code: either more compute-bound or more I/O-bound; this will allow to assess the impact of the I/O access bottleneck.

The selection fell on a scientific data analysis code developed by high energy physics scientists at the ATLAS Experiment [29] at CERN, to study the production of top quarks with the Higgs boson [30], following head-on proton–proton collisions (aka events) at the Large Hadron Collider (LHC): the $t\bar{t}H$ analysis. Fig. 8 represents the final state topology of a proton beam collision.

The outcome of an event is recorded by the ATLAS particle detector, which measures the characteristics of the bottom quarks (detected as jets of particles due to a hadronization process) and leptons (both muons and electrons), but not the neutrinos, as they do not interact with the detector sensors. These are later analytically reconstructed by the $t\bar{t}H$ application, through a kinematic reconstruction.

The $t\bar{t}H$ pipeline has 18 stages with the code structure of Fig. 1, each coded as a proposition in HEP-Frame. Each stage has a variable duration computational task (from few microseconds to several milliseconds per *n-tuple*) and a test to filter out measured events that do not comply with the theoretical model.

$t\bar{t}H$ may sample the kinematic reconstruction process within the 99% confidence level, to reduce the relative measurement uncertainty of the sensors in the ATLAS Experiment detector, with a direct impact on the complexity of the performed computation per event.

The proposition dependencies, defined by the user in HEP-Frame, are represented in Fig. 9. Since the order that the propositions are executed may have a significant impact on the overall execution time, the scheduler reordering of the computational pipeline aims to improve the application performance, respecting the inter-proposition dependencies.

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

- `ttH_as` (*accurate sensors*): the data measured by the ATLAS detector is considered 100% accurate when reconstructing the event; this behaves as a latency-bound code[3] in most computing systems;
- `ttH_sci` (*sensors with a confidence interval*): the ATLAS detector has a measurement accuracy error 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.

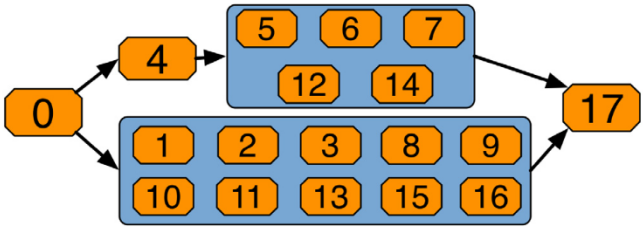

Fig. 9. Schematic representation of the proposition dependencies in $t\bar{t}H$ applications.

On a given compute server, 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 two of them are heavier with an execution time of 29 and 5 ms, respectively.

The new proposition 13 in `ttH_scinp` has a longer execution time than in `ttH_sci`, around 49 ms, and proposition 16 has now a passing ratio of 30%, versus 99% in `ttH_sci`.

The original sequential code of these three data analyses was parallelized with OpenMP [13], using a common approach in scientific code: independent events are processed in parallel by the pipeline. The OpenMP *dynamic* scheduler was used to adapt the workload distribution according to the irregularity of the pipeline execution. The input data read and setup and the output writing is performed sequentially. This implementation will be used to perform a thread-by-thread comparison with HEP-Frame.

The case studies propositions depend on ROOT [14], a complex library that cannot be properly ported to efficiently use accelerator devices, such as NVidia GPUs. The propositions also require

[3] Latency-bound code is limited by the memory latency, rather than its bandwitdth. Applications that access a small amount of data but multiple times in a inconsistent pattern may be susceptible to this behaviour.

major modifications to algorithms and data structures beyond ROOT to be executed on these devices, whose implementation requires know-how that most non-computer scientists lack.

5. Performance evaluation

This section resumes the key previous results and complements with new experimental measurements that evaluate the efficient execution of the improved HEP-Frame in heterogeneous environments. It starts by describing the testbed environment and the measuring methodology.

5.1. Testbed environment

Four different CentOS 6.3 compute servers performed the quantitative evaluation of the HEP-Frame with the multi-layer scheduler:

- a dual socket server with 12-core Intel Xeon E5-2695v2 Ivy Bridge devices @2.4 GHz nominal, 64 GiB RAM, with a NVidia Tesla K20 with 2496 CUDA cores and 5 GiB of GDDR5 memory, linked through PCI-Express;
- a dual socket server with 16-core Intel Xeon E5-2683v4 Broadwell devices @2.1 GHz nominal (1.7 GHz nominal with AVX2), 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), 192 GiB RAM;
- a 64-core Intel Xeon Phi 7210 (KNL) server @1.3 GHz (1.1 GHz nominal with AVX-512, 4-way simultaneous multithreading), 16 GiB of eRAM, 192 GiB of RAM; the KNL computing tiles were configured as SNC-4 clustering mode, with flat embedded RAM.

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 $\pm 6,000$ events (the dataset elements) and 250 different data variables per event, measured by the ATLAS detector. The code was compiled with the Intel 2018 compiler suite and the NVidia CUDA 8.0 toolkit. A k-best measurement heuristic was used to ensure that the results can be replicated, with $k = 5$ with a 5% tolerance and a minimum/maximum of 15/25 measurements.

5.2. Results and discussion

This subsection presents and discusses key performance measurements of the latest HEP-Frame version:

- efficiency of the dynamic tuning of *DS* and *DP* threads;
- the impact of HEP-Frame on the execution of the case studies;
- the use of a NVidia Kepler GPU as an accelerator;
- the performance of a KNL-based server with different configurations;
- performance of the best configuration of the KNL server *versus* multicore servers, with or without accelerators (including a GPU);
- the performance of a cluster with multiple KNL servers;
- a comparative evaluation of the HEP-Frame scheduler with the HEFT list scheduler.

More details on these and other experimental results can be found at the wiki website (link in the footnote at the Introduction).

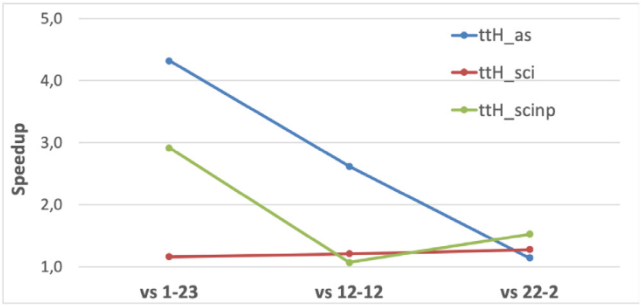

Fig. 10. Speedup of dynamic tuning *vs* static of *DS* and *DP* threads.

5.2.1. Dynamic tuning of DS and DP threads

The dynamic tuning of *DSt* and *DPt* on a dual 12-core Ivy Bridge server is compared against 3 fixed configurations of *DSt-DPt*: 1-23, 12-12 and 22-2 (Fig. 10). These fixed configurations attempt to illustrate what a user could manually choose as a fixed distribution of the amount of *DS* and *DP* threads without extensive profiling of the behaviour of the case studies during their run-time on a 24-core server.

The dynamic tuning outperforms all fixed configurations. A 4x speedup was achieved for `ttH_as` against using 1 *DSt*, since this code is limited by the *DS* phase, as most of the application execution time is spent on this phase. Having more threads concurrently loading data allows the remaining *DP* threads to process data without having to wait as long as the fixed configuration, for its setup.

The `ttH_as` converges to a stable *DS-DP* configuration after loading 5% of the dataset (converge to *22-2*), while the other two case studies converge after only $\pm 2.5\%$ of the dataset (to *2-22* and *6-18*, respectively). The convergence of this process depends on the irregularity of the computation, which is different for each case study, and the irregularity created by the information in the dataset, which is the same. This means that the `ttH_as` has a more dynamic behaviour, where initially the processing is more irregular than the other case studies that converge faster (as stated before the propositions execution time depends on different parameters of the dataset elements).

An analysis of the scheduler with the Intel VTune profiler showed that, for the `ttH_as` latency-bound code, the most challenging case study for the scheduler, the *DP* threads were waiting for data to be loaded for less than 10% of the overall data setup time, resulting in a 90% utilization of the available computing resources in the server during this case study execution. Comparatively, for a configuration with only one *DS* thread the *DP* threads were waiting for data 95% of the overall execution time. `ttH_sci` and `ttH_scinp` presented much higher utilization, as they are inherently compute-bound.

5.2.2. Multithreading with and without HEP-frame

The performance of the three $t\bar{t}H$ analyses using HEP-Frame was compared against their original implementations without HEP-Frame but with a simple dataset-level parallelization of the pipeline with OpenMP,[4] using one and two Xeon devices of the Ivy Bridge, Broadwell, and Skylake micro-architectures (speedup outcomes in Fig. 11). HEP-Frame significantly improved the performance of all multithreaded codes:

[4] In this implementation multiple threads process the whole pipeline with different dataset elements, where the data reading and setup are sequential, as in most analyses.

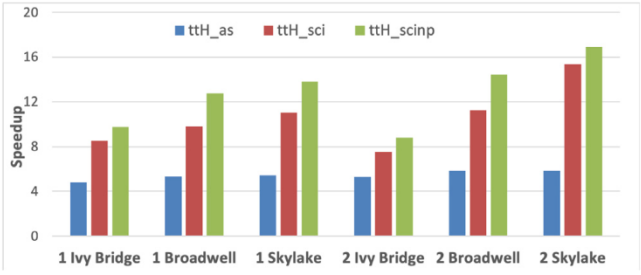

Fig. 11. Speedup of the parallel $t\bar{t}H$ analyses with HEP-Frame *vs* a standard OpenMP parallelization for the same number of threads.

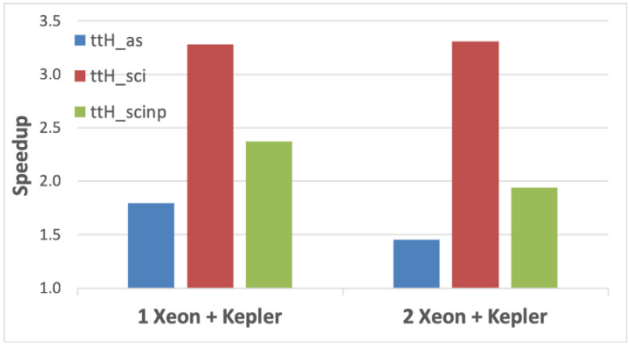

Fig. 12. Speedup of the case studies in HEP-Frame on servers with accelerators.

- `ttH_as`: up to 6x due to simultaneous *DSt* and *DPt*;
- `ttH_sci` and `ttH_scinp`: respectively 11.5x and 15x speedups, mostly due to the pipeline run-time ordering and workload scheduler;
- `ttH_scinp`: up to 17x mostly due to a worse initial pipeline order.

A second multicore device in the same server did not always lead to a linear performance improvement due to the NUMA architecture. However, the user could allocate one process per multicore device to eventually achieve higher speedups. This approach was not considered in these tests since it would require user configuration.

The Skylake server had the best speedup mostly due to its higher core count.

5.2.3. GPU accelerators in the servers

The performance of two configurations of the heterogeneous system was compared against homogeneous multicore configurations, shown in Fig. 12: 1 or 2 Ivy Bridge devices with a Kepler GPU *vs* 1 or 2 Ivy Bridge devices.

The dual-buffer PRNG offload to the GPU device led to an higher PRN throughput, freeing multicore time to run other parts of the case studies. This is specially evident in `ttH_sci`, as it is the most compute intensive case study.

5.2.4. The KNL-based server

The KNL package has 32 mesh-interconnected compute tiles, each a dual core PU sharing L2 cache, with on-package configurable embedded RAM (eRAM). The mesh structure and the memory organization are configured in boot time. KNL configurations are detailed in [31].

The tiles mesh can be configured as all-to-all (generally the worst performing), hemisphere/quadrant or sub-NUMA clustering (SNC-2/SNC-4), providing different levels of address affinity and impact overall performance. The high bandwidth 16 GiB of eRAM (with similar latency as external RAM) can be configured as last level cache, as flat addressable RAM (mapped in the overall address space), or as an hybrid of both.

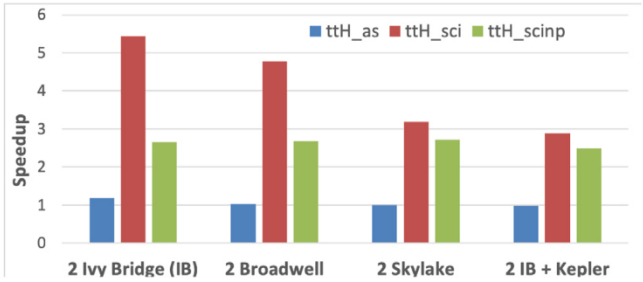

Fig. 13. Speedup on the KNL server *vs* multicore dual socket servers.

Experimental tests to evaluate the performance of the HEP-Frame scheduler on the KNL server used 4 processes with a total of 64, 128 and 256 threads. Results were compared against the original multicore scheduler with 24 threads on the dual 12-core Ivy Bridge server, showing that:

- the best configuration for the computing tiles is the SNC-4 mode, and eRAM as flat addressable RAM;
- eRAM as cache decreases performance in all case studies by 10%–30%, since this code reuses little data (independent processing of each dataset element);
- the peak speedup of 5.5x over the multicore server for the `ttH_sci`, with 128/256 threads, is mostly due to the vectorization capabilities, larger overall L2 cache and 4 independent processes with its own *DS*;
- the all-to-all clustering mode was $\pm 2x$ slower than the SNC-4.

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).

Fig. 13 compares the performance of the KNL server with three multicore servers without accelerators and the Ivy Bridge server with one Kepler GPU.

The `ttH_sci` application running with 128/256 threads on the KNL outperforms the multicore-only 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.

5.2.5. Multiple KNL servers

Fig. 14 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.

Fig. 14. Speedup of the case studies for 2, 4, and 6 KNL servers *vs* a single KNL server.

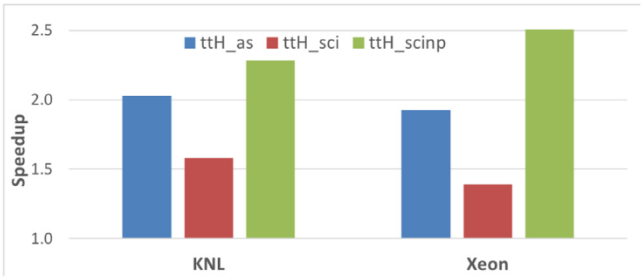

Fig. 15. Speedup of the HEP-Frame scheduler *vs* the StarPU using the `dm` scheduler.

`ttH_sci`, the most compute intensive analysis, 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.

5.2.6. HEP-frame scheduler vs starpu HEFT

Fig. 15 compares the performance of the HEP-Frame scheduler with the more efficient StarPU HEFT scheduler, the double-queue model (`dm`), on a single dual socket Xeon server and a KNL server: on both servers the HEP-Frame scheduler outperformed StarPU. The port of the case studies to StarPU was performed by the authors, as the particle physicists that implemented them on HEP-Frame did not have the required expertise to adapt to the steep learning curve of StarPU. The code could not be adapted to use GPUs, as it depends on ROOT functionalities that could not be ported.

The `ttH_sci` performance only improved ±50% on HEP-Frame since this code benefits less from reordering and behaves more as a regular compute-bound application. The most compute intensive proposition is at the end of the pipeline (taking 70% of the analysis execution time), while the most filtering propositions are at the beginning by default. This is the case study that best fits the characteristics of StarPU, as it focuses on compute-bound code.

The speedup goes to 2.5x for the `ttH_scinp` analysis, as it benefits the most from the proposition reordering in HEP-Frame, since the default order is not as good as in `ttH_sci`. The StarPU HEFT scheduling is less efficient with memory-bound code, as shown by the speedup of using HEP-Frame on the `ttH_as` analysis, which is achieved by adapting the amount of *DS* threads accordingly.

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

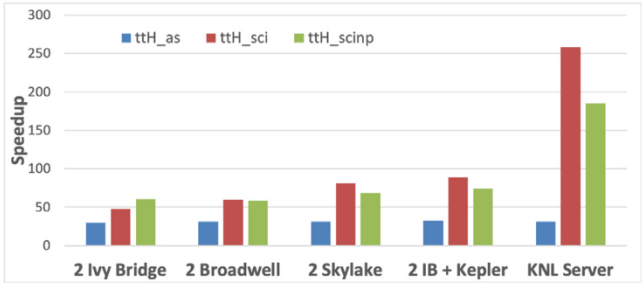

Fig. 16. Overall speedup on HEP-Frame *vs* their original sequential implementation.

One limitation of StarPU HEFT scheduler common to the three case studies is that explicit proposition dependencies cause a proposition to await execution if it depends on the one being executed, therefore diminishing processor usage. HEP-Frame schedules propositions from other dataset elements to mitigate this problem and improve processor usage, leading to a performance advantage over StarPU.

5.2.7. Overall performance improvement

Fig. 16 shows the speedup obtained with HEP-Frame *vs* the original sequential implementation of the three case studies on several servers.

HEP-Frame provides a significant performance improvement for every case study, which is consistent across multiple platforms with significant architectural differences. It adapts to irregular compute-bound code, with overall speedups on the KNL server up to 252x and 185x for the `ttH_sci` and `ttH_scinp` applications. It also efficiently handles memory-bound code, with a speedup of 30x of `ttH_as` application for every server.

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 synchronizations and data consistency overheads. The gap could be reduced if the user had defined a similar multiprocess approach to the dual socket servers.

6. Conclusions and future work

This paper presented and discussed the key features of a multi-layer scheduler for heterogeneous systems, best suited for pipelined scientific data analyses. The scheduler is the main component of a framework that aids the development and execution of pipelined applications in heterogeneous systems, HEP-Frame, where large raw experimental data is converted into useful information through complex computational tasks. It provides several abstractions of the computational complexities of heterogeneous systems, so that scientists can develop efficient code in an easy and intuitive environment.

The HEP-Frame scheduler was evaluated with three versions of an actual case study: 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. Each scheduler layer performs a different action:

- the top layer balances data and workloads among servers in a heterogeneous cluster environment; it achieved good scalability with both memory- and compute-bound codes, in either multiple homogeneous or multiple heterogeneous servers;
- 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 KNL manycore server provided speedups up to 5.5x, 4.8x and 3.2x for the `ttH_sci` application, when compared to the dual socket multicore servers, based respectively on Ivy Bridge, Broadwell and Skylake devices. The KNL server also outperformed the Ivy Bridge server with a 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 I/O-bound code, the `ttH_as` application on the KNL server ran with the same performance.

The framework also provides library functions, such as efficient parallel implementations of different PRN generators for both CPU and GPU devices, hidden behind an easy to use API. The GPU PRNG achieved almost 2x performance improvement over the homogeneous Xeon server for the PRN intensive `ttH_sci` application; for the other `ttH` versions the speedup was marginal as they do not rely as much on PRNs. The KNL server is still faster than this heterogeneous server, with a 2.9x performance improvement.

The HEFT list scheduler, integrated into HEP-Frame to be compared with the framework scheduler, was outperformed by the HEP-Frame scheduler in all tested cases. Most performance gains were due to the balance between data setup and processing and the reorganization of the propositions execution based on their filtering rate.

HEP-Frame may not be as versatile as other frameworks, since its scheduler is specially tuned for pipelined scientific code, but it provides a user friendly environment to develop efficient and portable applications. This ensures that scientists trust their code and can easily maintain and improve it, which is not possible when their original code is heavily modified to extract efficient parallelization by someone else, on a single or multiple heterogeneous servers.

Overall, the KNL server provided better performance than any other tested server. The best global performance improvement over the fine tuned original `ttH_sci` sequential code was: 81x faster in the homogeneous dual 24-core Skylake server, 86x faster in the heterogeneous dual 12-core Ivy Bridge server with the Kepler GPU, and 252x faster in the 64-core KNL server.

6.1. Future work

Extensions to the HEP-Frame are currently being considered. One is to add to the scheduler support for nested pipelining, to process a sequence (pipeline) of batches, where each batch-proposition is a pipelined data stream. Nested pipeline reordering may be required to process complex applications, where scheduling of a single pipeline is an over-simplification of the workload that may not fully exploit the potential improvements of multiple pipeline reordering.

Another path is to extend the scheduler to support the ordering of multiple *DS* blocks, each reading sets of values for distinct datasets. These *DS* tasks may mix batch *DS* with continuous data streaming, often required by queries on large databases.

An improvement of the top layer of the HEP-Frame scheduler is under evaluation. Currently, it focuses on managing workloads among a small amount of servers with a small overhead, but a more complex strategy may be required to handle a large amount of servers, which is common in most computing clusters.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data statement

The scientific data analyses and full input data used in this communication cannot be provided. Even though these analyses were developed and published by particle physicists at Laboratório de Instrumentação e Física Experimental de Partículas (www.lip.pt), in close cooperation with the ATLAS Experiment at CERN, the code should remain private as it is currently being used in a production environment.

A simple scientific data analysis is provided with the purpose of illustrating the HEP-Frame main functionalities. An input data file is also provided, which is a subset of the input data used to obtain the performance measurements presented in this communication. It contains simulated particle collision data created by particle physicists within the ATLAS Experiment.

Funding

This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

References

[1] J. Liu, E. Pacitti, P. Valduriez, M. Mattoso, J. Grid Comput. 13 (4) (2015) 457–493.
[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Future Gener. Comput. Syst. 46 (2015) 17–35.
[3] A. Pereira, A. Onofre, A. Proenca, Proceedings of the 2015 International Conference on Computational Science and Computational Intelligence, IEEE, 2015, pp. 615–620.
[4] A. Pereira, A. Onofre, A. Proenca, Proceedings of the International Conference on High Performance Computing Simulation, HPCS, IEEE, 2016, pp. 751–758.
[5] A. Pereira, A. Proenca, Proceedings of the International Conference on Mathematical Applications, Institute of Knowledge and Development, 2018, pp. 7–12.
[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. 36 (4) (2015).
[7] M.H. Iqbal, T.R. Soomro, Int. J. Comput. Trends Technol. 19 (1) (2015) 9–14.
[8] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al., Commun. ACM 59 (11) (2016) 56–65.
[9] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Concurrency Comput.: Pract. Exp. 23 (2) (2011) 187–198.
[10] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, pp. 66:1–66:11.
[11] A. Duran, E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Parallel Process. Lett. 21 (02) (2011) 173–193.
[12] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, Parallel Comput. 38 (1–2) (2012) 37–51.

[13] L. Dagum, R. Menon, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[14] F. Rademakers, Comput. Phys. Comm. 180 (12) (2009) 2499–2512.
[15] A. Pereira, A. Onofre, A. Proenca, Proceedings of the 14th International Conference on Computational Science and its Applications, Springer International Publishing, 2014, pp. 576–591.
[16] S. Skiena, The Algorithm Design Manual, second ed., Springer-Verlag London, 2008.
[17] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, High-Performance Computing on the Intel Xeon Phi, Springer, 2014, pp. 167–188.
[18] M.E. O'Neill, PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation, Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 2014.
[19] M. Matsumoto, T. Nishimura, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30.
[20] H. Topcuoglu, S. Hariri, M.-Y. Wu, Proceedings of the 8th Heterogeneous Computing Workshop, IEEE, 1999, pp. 3–14.
[21] T. Urhan, M.J. Franklin, Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 2001, pp. 501–510.
[22] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, 2004, pp. 407–418.
[23] M.I. Gordon, W. Thies, S. Amarasinghe, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, 2006, pp. 151–162.
[24] H. Arabnejad, J.G. Barbosa, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 682–694.
[25] C. Min, Y.I. Eom, IEEE Trans. Parallel Distrib. Syst. 26 (6) (2015) 1594–1607.
[26] Y. Liu, L. Meng, I. Taniguchi, H. Tomiyama, Int. J. Netw. Comput. 4 (2) (2014) 279–290.
[27] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K.S. McKinley, F.X. Lin, 2017 USENIX Annual Technical Conference, 2017, pp. 617–629.
[28] N. Zhou, D. Qi, X. Wang, Z. Zheng, W. Lin, Concurrency Comput.: Pract. Exp. 29 (5) (2017).
[29] ATLAS Collaboration, J. Instrum. 3 (08) (2008) S08003.
[30] ATLAS Collaboration, Phys. Lett. B 716 (1) (2012) 1–29.
[31] A. Sodani, R. Gramunt, J. Corbal, H.S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y.C. Liu, IEEE Micro 36 (2) (2016) 34–46.