HEP-Frame: A Software Engineered Framework to Aid the Development and Efficient Multicore Execution of Scientific Code

Andre Pereira
Department of Informatics
LIP and University of Minho
Braga, Portugal
Email: ampereira@di.uminho.pt

Antonio Onofre
Department of Physics
LIP and University of Minho
Braga, Portugal
Email: Antonio.Onofre@cern.ch

Alberto Proenca
Department of Informatics
University of Minho
Braga, Portugal
Email: aproenca@di.uminho.pt

***Abstract*—This communication presents an evolutionary software prototype of a user-centered Highly Efficient Pipelined Framework, HEP-Frame, to aid the development of sustainable parallel scientific code with a flexible pipeline structure. HEP-Frame is the result of a tight collaboration between computational scientists and software engineers: it aims to improve scientists coding productivity, ensuring an efficient parallel execution on a wide set of multicore systems, with both HPC and HTC techniques. Current prototype complies with the requirements of an actual scientific code, includes desirable sustainability features and supports at compile time additional plugin interfaces for other scientific fields. The porting and development productivity was assessed and preliminary efficiency results are promising.**

***Index Terms*—High Throughput Computing, Pipeline, Coding Environment, Execution Efficiency.**

I. Introduction

Computational sciences in the context of this work address the resolution of complex science and technology problems through the intensive use of computing resources. This intensive use may address one or both targets: to shorten the time to get a result, or to to obtain more results per time unit. The former is known as high performance computing (HPC) while the latter is commonly known as high throughput computing (HTC). Scientific applications often include both needs.

A current societal requirement is sustainable computing, which aims to minimize the cost of using intensive computing now and in future generations. This goal compels computational scientists to give higher priority to the efficiency of their scientific code on any computing platform, applying a merge of HPC and HTC techniques without compromising the accuracy and robustness of the software application. The design of sustainable scientific code in any multicore platform, either current or in the future, relies on a set of key features: (i) knowledge of its science/engineering domain, (ii) clear definition of the application requirements and (iii) adequate design of sustainable software, both the algorithms and the data structures, to be efficiently executed on a wide range of HPC/HTC computing environments.

Common complex scientific applications are related to modeling/simulation and quantitative data analysis and these often rely on a set of pipelined tasks, where tasks execution may depend on previous conditions and their execution order may vary. These flexible pipelined applications can be described through *propositions* in linear temporal logic with the structure displayed in figure 1, where a *proposition* is a simple or complex computational task applied to a dataset element and an optional verification of a given criteria to decide if that element is further processed or simply discarded.

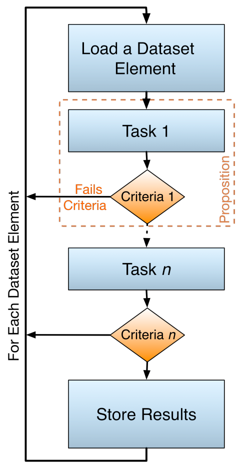

Fig. 1. Structure of a typical flexible pipelined application.

A computational scientist is the best expert in the domain of her/his scientific field and has a clear understanding of the software requirements under development. However, she/he often has a limited knowledge of the updated methods and techniques to develop sustainable parallel code, while a software engineer may be an expert on software design, clearly specifying the requirements of a sustainable software application, but often hardly grasp the end user domain and his view of the requirements may not match the scientist view. A successful approach to the development of sustainable scientific software requires an adequate bridge between the science/engineering domain and the underlining computing environments, a user-centered framework, developed through a tight cooperation between a (computational) scientist and a software engineer expert.

Some general purpose libraries and frameworks already aid the development of parallel code on multicore systems. Libraries, such as those in OpenMP [1] or TBB (Intel Threading Building Blocks) [2], address the workload distribution on multicore systems but require computing expertise to handle data consistency, avoid race conditions, and ensure the correctness of the application, features that often lack in scientists.

Frameworks, such as StarPU [3] or Legion [4], dynamically manage the workload distribution among computing units on heterogeneous platforms with both multicore CPU and GPU devices, but require applications to be rewritten according to their restrictive specifications. They also require a high level of expertise to take full advantage of the efficiency potential of these frameworks, and scientists do not feel comfortable with their learning curve.

A successful approach to the development of sustainable scientific software requires an adequate bridge between the science/engineering domain and the underlining computing environments, a user-centered framework, developed through a tight cooperation between a (computational) scientist and a software engineer expert.

Pipelined applications typically have the structure presented in figure 1. In the context of this work, a *proposition* is a computational task, which is applied to a dataset element, and an evaluation of a given dataset element characteristic that may restrict the execution of subsequent *propositions*. The tasks may be computationally simple or complex, and the criteria may discard different amounts of dataset elements. The work presented on this paper is also capable of addressing performance issues of pipelined applications that do not have the criteria component.

This communication address an evolutionary software prototype of an Highly Efficient Pipelined Framework, HEP-Frame, a user-centered framework to aid the development of sustainable parallel scientific code with a flexible pipeline structure. HEP-Frame aims to improve scientists coding productivity and robustness, while ensuring an efficient parallel execution of the resulting application on a wide set of multicore computing platforms. The design complies to the requirements of an actual scientific code (an event data analysis code in the search of the Higgs Boson, at LIP/CERN[1]), its implementation includes the desirable sustainability features, and it may be later refined. The framework supports at compile time additional plugin interfaces for other specific scientific fields, providing an easier and less error prone development environment. The impact of porting and development productivity is briefly assessed, and the HPC/HTC efficiency results of this prototype are presented and evaluated.

The communication is structured as follows: section II introduces the design features of HEP-Frame and the current prototype; section III presents an actual pipelined application to evaluate the HEP-Frame; section IV discusses development productivity issues complemented with performance portability analysis of the resulting multicore code; section V concludes the communication with a critical analysis and suggestions for future developments.

[1]Laboratório de Instrumentação e Física Experimental de Partículas, LIP, is an associated laboratory of the European Organization for Nuclear Research, CERN, which operates the Large Hadron Collider particle accelerator.

II. The HEP-Frame

Two main approaches to development frameworks are currently claiming to aid the design and deployment of scientific code: a resource-centered approach, closer to the computing platforms, stressing efficiency and performance portability, but forcing the scientists to rewrite the existing code to adapt it to their constraints, being the most relevance StarPU and Legion; an alternative user-centered approach that stresses the interface to domain experts to improve their productivity and code robustness, at the same time including the desirable sustainability features. The user-centered HEP-Frame aims to motivate scientists to adopt a user-friendly framework that dynamically addresses the efficiency concerns across different types of parallel computing platforms.

This section presents the design of a user-centered framework that automatically parallelizes scientific pipelined applications, bridging the gap between code execution efficiency and development productivity. It provides a user-friendly interface without requiring any specific tuning for each individual scientific code or computing platform. Optimization efforts focus on improving both the execution throughput of an input dataset and the execution time of each element in the dataset, in offline quantitative analyses. In its final development stage, HEP-Frame will also address performance issues of compute, memory and I/O bound applications, through techniques already present in its design. It can be later extended to process continuous streams of online data and produce results in real-time.

Subsection II-A presents the design of the HEP-Frame, in terms of data structuring, user interface, programming model, and optimization techniques. Subsection II-B discusses some efficiency features. Subsection II-C introduces the current evolutionary prototype of HEP-Frame with a set of already implemented functionalities.

A. The Design

The framework design focus on the development and efficient execution of compute and memory bound pipelined applications. It aids the code development, implementing several application-specific features with an user-friendly interface. It also automatically produces efficient code, tuned at runtime for any computing platform, taking advantage of the code structure and domain knowledge to optimize data processing throughput, in a transparent way to the user. The structure of the framework is presented in figure 2.

The framework operates at compile time, by the use of tools to provide several abstractions to the developer and translate those abstractions into working code, and at runtime, by implementing several parallelization and optimization mechanisms

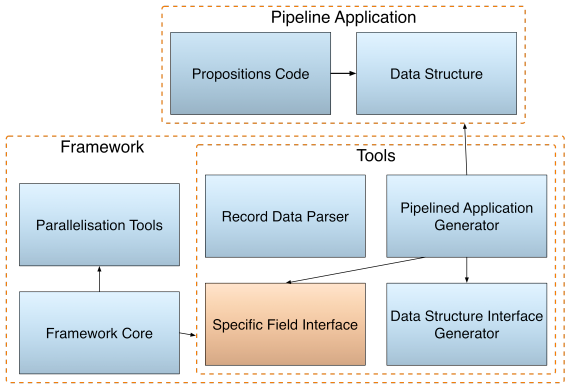

Fig. 2. HEP-Frame modular structure.

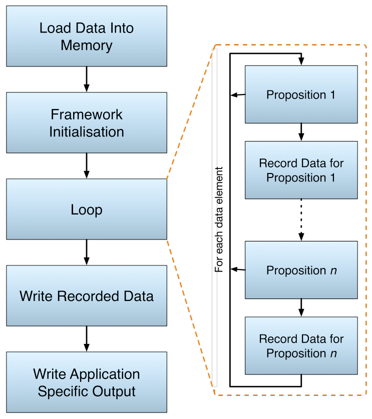

Fig. 3. HEP-Frame execution flow.

to dynamically deal with regular and irregular workloads. The user is able to create a new application code skeleton through the `Pipelined Application Generator`, with all the files required for the *propositions* code, configuration, and the data to be recorded per *proposition*. The `Record Data Parser` automatically uses the latter file to create the required code to store information for each dataset element specified by the user. The dataset structure, specified by the user, is parsed by the `Data Structure Interface Generator` to create an interface to hide the complexities of the data structure from the user, giving the illusion that the information of each dataset element is stored on global memory. The `Specific Field Interface` is a tool that can be integrated with the framework and can ease the scientists development environment by providing specific functionalities or by automatically creating the dataset structure.

The *Framework Core* encapsulates all functionalities required to develop the application and efficiently execute the code. At this stage, the user codes the *propositions* as functions, considering that they do not receive any inputs and return a boolean (true if the criteria is met and false otherwise), and submits them to the framework. When executing the code, the framework loads the data from the input file to memory, initialises timers, performs initial parallelism configurations, and applies the user coded *propositions* to the each dataset element. This execution flow is presented in figure 3.

B. Efficiency Features

To embed efficiency features in the final code that the framework generates, some key approaches must be followed, including (i) removal of algorithmic and data structures inefficiencies, (ii) taking advantage of the underlying parallel hardware and (iii) adequate ordering of the pipeline tasks (*propositions*). The former was already addressed on a discussion of an actual particle physics pipelined application [5] and the automatic removal of most reported inefficiencies will be later included in HEP-Frame.

Two main types of parallelization techniques can be applied in an application: process intra-dataset elements, which will favour HPC, or process inter-dataset elements, to favour HTC code. Intra-dataset element optimizations are oriented to complex processing of each dataset element, which can be identified and optimized, namely in multicore CPUs or offloaded to accelerator devices, such as the Intel Xeon Phi or NVidia GPUs. Inter-dataset element parallelism may be easier to implement; the absence of dependencies among dataset elements makes this an embarrassingly parallel problem and if the code has little dependencies on external libraries the dataset element processing can be offloaded to accelerator devices. When the required performance of an application requires both HPC and HTC optimizations, a balance between both approaches requires careful study, as discussed ahead.

A flexible pipeline is described in linear temporal logic as a formula $FGP_1 \wedge FGP_2 \wedge ... \wedge FGP_n$, where each *proposition* P_i is a task in the pipeline, F assumes that a *proposition* will eventually be true in the future, and G states that it will always hold its truth value. *Propositions* may have dependencies among them, declared by the user in the framework, which must be respected. If P_2 depends on P_1 the formula changes to $FGP_n \wedge ... \wedge FG(P_1 \wedge FGP_2)$. This formula states that the *propositions* can be executed in any order (the conjunction is commutative), as long as P_2 is executed after P_1. An element from the dataset D has to fulfil the formula $D_i \models FGP_{ni} \wedge ... \wedge FG(P_{1i} \wedge FGP_{2i})$ to pass all tasks in the pipeline.

In the pipeline flow, the propositions execution order might have a context to the user, but it may not be the most efficient from the computational point of view. Propositions with a high failure probability of the filtering criteria should be placed in an early stage, while propositions with high execution time might be best placed later in the pipeline flow. This reordering of the propositions should be performed dynamically during the application execution, as the filtering ratio and execution times are not known at compile time and might vary during the dataset processing.

The proposition reorder mechanism measures each proposition execution time and filtering ratio between specified checkpoints. An ordering weight is attributed to each proposition based on the filtering ratio, the execution time and the position in the pipeline flow, which is stored in an array. A simplified starting approach does not account the proposition place in the pipeline flow, hence the weights are stored in a matrix. The best proposition order is obtained by finding the path that passes through all propositions in the matrix, without returning to the initial position, known as the directional Hamiltonian path [6]: a path that visits each vertex of a graph exactly once, an NP-complete problem. The weights for each new checkpoint consider the best placement of propositions in the pipeline flow after being reordered, converging to a near optimal solution.

C. The Prototype

A preliminary prototype of the framework was developed, and it is currently being used by a research group on particle physics, which focused on implementing the features required to aid the development of pipeline code, by providing a user-friendly interface. Some code optimizations were also implemented and are discussed later in this section. From the framework modules presented in subsection II-A, only the integration with automatic parallelization frameworks for heterogeneous system was not performed.

The *Pipelined Application Generator* tool creates a sample code skeleton based on an application name provided by the scientist and an input file. This code skeleton contains the constructor of the main class, the basic input parameters reading (input event file and output file name for the variables recorded), a *proposition* function prototype, and the `main` function. It also creates a second file where the user will later define the dataset element variables to record per *proposition*. The *Record Variable Parser* parses this file to check if the variables exist in the dataset structure. The scientist only has to write the variable name inside the specified section of the file. The tool creates the required code to store the information, and supports scalars, array elements or entire arrays of any type. It is also possible to specify simple arithmetic operations, such as `a[0] * a[1] - b`, where the expression is computed and then stored in a data structure with the appropriate type to avoid losing numeric precision. An interface for high energy particle physics was developed, which provides a set of functionalities, interaction with specific libraries, and an automatic data structure creation based on a given input file for the application.

The *Framework Core* implements an execution flow where the data is loaded into a specific structure and processed. It is planned to later implement a stream-like flow, where the dataset elements are being loaded simultaneously to the processing of other events, and at that stage a more suitable data structure will be used. The *propositions* function pointers are stored into an array and are executed in their original order by the `loop` method. These details are hidden from the user.

Inter-dataset elements parallelization was adopted in a first stage, using OpenMP to manage the threads among the cores of the CPUs. Due to the irregular nature of this domain, a dynamic scheduler was used to perform the load balancing among threads. The data to be saved for each *proposition* is stored locally to each thread, and is merged after the processing of the entire dataset to minimize synchronizations. Intra-dataset elements can be performed by either parallelizing the execution of a complex *proposition* or executing multiple propositions simultaneously. The former requires data structures to be created dynamically to store intermediate results, thus applying the same proposition simultaneously to multiple dataset elements, and the latter may not provide performance improvements, as most *propositions* execution time is very small and the overhead would restrict the performance. This two alternatives will be addressed in later evolutions of the HEP-Frame prototype.

HEP-Frame implements a backtracking algorithm, whose flow is presented in figure 4, to obtain the best *proposition* order with the simplified design presented in subsection II-A. The paths where *proposition* dependencies are violated are interpreted with an infinite cost. The user has the option to save the best *proposition* order to be later loaded as an initial order for other executions of the application.

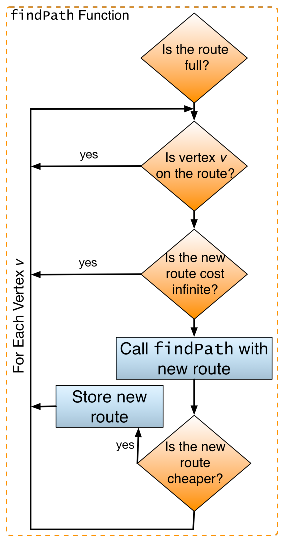
findPath Function
Is the route full?
Is vertex v on the route?
yes
Is the new route cost infinite?
yes
For Each Vertex v
Call findPath with new route
Store new route
yes
Is the new route cheaper?

Fig. 4. *Proposition* reorder backtracking algorithm.

III. AN ACTUAL CODE TO EVALUATE HEP-FRAME

An actual pipelined scientific code was selected to assess the HEP-Frame as a development aid and to evaluate its performance portability: a particle physics event data analysis after a proton beam collision at CERN, the `ttH_dilep` application [5]. Following the discovery of the Higgs boson at CERN, one of the searches conducted at the LHC is the study of the associated production of top quarks together with Higgs bosons ($t\bar{t}H$) [7]. This search has been carried on by both ATLAS [8] and CMS [9] at the LHC and it is of crucial importance to understand the couplings of the top quarks to the Higgs boson. Figure 5 represents the final state topology of a proton beam collision for the $t\bar{t}H$ production. The experiments can record the characteristics of the bottom quarks (detected as a jet of particles) and the two leptons (muon or electron), but not the neutrinos, since these do not interact with the detector. However, the top quark reconstruction requires the neutrinos, so their characteristics are analytically determined with the known information of the system, through a kinematical reconstruction.

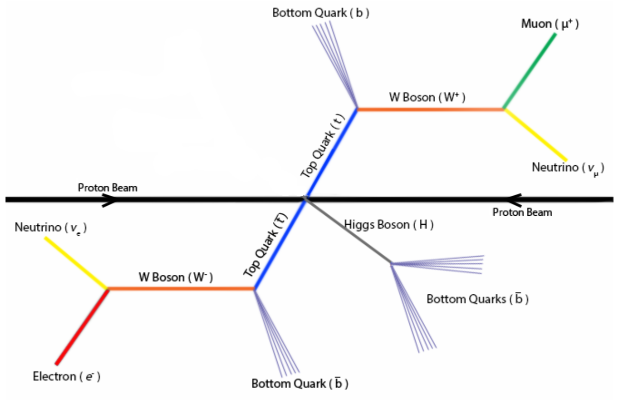

Fig. 5. Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

The `ttH_dilep` was developed to perform this analysis and has 18 stages to filter out the measured results that do not comply with the expectations (here referred to as *propositions*), with the kinematical reconstruction being the most computationally intensive. To further improve the accuracy of this reconstruction, the detector experimental resolution ($\pm 1\%$) was considered during the computation, through 1024 random variations of the measured data with its magnitude. From all variations within an event, only the one that best fits the theoretical model is chosen, improving the final analysis quality.

IV. Results and Discussion

The porting of a real case study (the `ttH_dilep` described in section III) into the proof of concept framework did not involve the computational scientist, but the prototype was reviewed and assessed by the scientist as an end-user, who ported his 4-year legacy code into the HEP-Frame in just 4 hours, after a 15 minute crash-course, without requiring substantial changes to the original code. The full functional framework will not require a learning curve longer than half an hour and provides relevant development aids, namely creation of data structures and their access, based on the input data file, automatic generation of domain specify functions, including file reading/writing and common statistical operations, and a guarantee of functional (and efficiency) portability of the supplied code across different computing platforms, in time and space. The code of a pipelined application can easily be modified at each computational task or at the filtering criteria at each *proposition*.

An optimized parallel implementation of the pipelined application used as case study, previously described in [5], combines both HPC and HTC approaches. The former improves the performance of a single event (in this context an event is the processing of a single dataset element), by parallelizing the execution of the heaviest *propositions*, while the latter improves the event throughput with multiple simultaneous execution of events. The results shown in figure 6 combine both HTC and HPC approaches, and show that it is more efficient to explore an HTC approach versus HPC, although the best efficiency results will need both approaches. It is also shown that hardware multithreading on CPU devices only improves the performance if both approaches are used simultaneously. Therefore, the initial HEP-Frame prototype mainly addresses the HTC approach, with promising performance results, and will soon be improved by implementing HPC features and other tested optimizations described in [5].

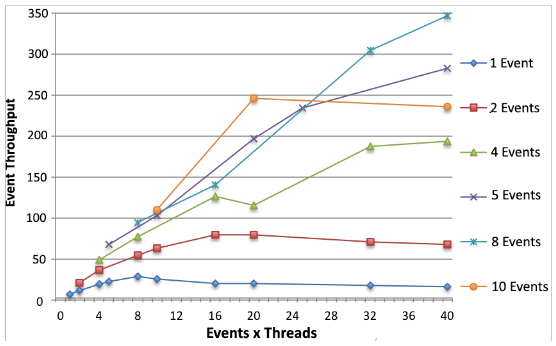

Fig. 6. Speedup for an hybrid multiprocess/multithread custom parallelization of `ttH_dilep` on a dual socket system with 10-core Intel Xeon E5-2670v2.

The performance portability of HEP-Frame was studied on different dual-socket computing nodes. The execution times of the original sequential `ttH_dilep` and the version ported to HEP-Frame are presented in table I. The *K*-best measurement scheme was adopted [10], with *K* set to 5 and a 5% tolerance, to ensure that only the best, but consistent, time measurements are considered. The number of threads is automatically set by the framework to the number of physical cores in both CPU devices, without any tuning by the user.

TABLE I
Execution times of the sequential and parallel `ttH_dilep` with HEP-Frame.

Intel Xeon	E5520	X5650	E5-2650v2	E5-2670v2	E5-2695v2
μArchitecture	Nehalem	Nehalem	Ivy Bridge	Ivy Bridge	Ivy Bridge
#Cores	2 x 4	2 x 6	2 x 8	2 x 10	2 x 12
Clock Freq.	2.27 GHz	2.67 GHz	2.6 GHz	2.5 GHz	2.4 GHz
Sequential Exec. Time (s)	215	196	175	180	183
Parallel Exec. Time (s)	45	30	23	22	23

The HTC parallelization approach in HEP-Frame improves the event throughput up to 8 times on a dual 10-core system (20 events), when compared to the sequential version (figure 7). This improvement still lies considerable short when compared with the version without HEP-Frame (figure 6, 50 times faster with 10 events with 2 threads/event). This is due to the sequential nature of the different implementations of the I/O operations (these are still sequential operations in current

TABLE II
CHARACTERIZATION OF THE 18 TTH_DILEP *propositions*.

Execution Time (ns)	Dataset Elements that Pass
1 : $\sim 10^8$	1 : $\sim 50\%$
1 : $\sim 10^6$	1 : $\sim 63\%$
1 : $\sim 10^4$	1 : $\sim 94\%$
2 : $\sim 5 * 10^3$	15 : $>99\%$
13 : $<10^3$	-

HEP-Frame prototype) and the merge of results at the end was not accounted in the version without HEP-Frame. The use of simultaneous multithread at each core may also be worth considering later. The use of hardware accelerators to improve the performance of specific *propositions* is being studied, with the adoption of specialized frameworks, such as Legion, to manage irregular load balancing. The goal is to identify at compile time *propositions* whose code is suitable to execute on such devices, and at runtime offload only the ones that require intensive computing.

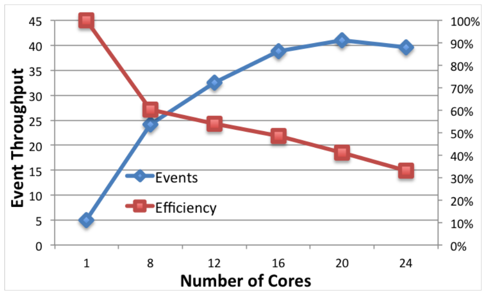

Fig. 7. Event throughput and CPU efficiency with HEP-Frame.

The properties of the *propositions* in the actual code used as case study does not show the potential of the *propositions* reorder mechanism: the range of the *propositions* execution time is too large (6 orders of magnitude) and they do not filter a substantial amount of dataset elements, as shown in table II. However, for datasets that require more than 100 reorders the performance already improves by more than 70%.

V. CONCLUSIONS

This communication describes HEP-Frame, a user-centered framework that aids scientists to improve coding productivity, while ensuring efficient parallel execution of the application on various computing platforms, in a way transparent to the end-user. A particle physics event data analysis was used as an actual case study, due to its suitable computational features. HEP-Frame enables scientific code to be easier maintained (write once, efficiently runs forever) while supporting the development of more complex algorithms to improve the analysis accuracy due with a better data processing throughput.

The coding productivity of HEP-Frame was assessed by measuring the time of porting an event analysis application to the framework by its physicist developer. The expected learning curve of the final version of the framework will take no longer than 30 minutes. The performance of the case study was improved by up to 8x in a dual socket system, without parallelization of heavy I/O functions. Results also sowed that HEP-Frame ports the efficiency of pipelined applications on different computing platforms. Parallelization inefficiencies were already identified on the original code, namely on the concurrent access to the data structure and at the merge of the final results of each thread, and will be addressed in future version of the framework. The feasibility of automatic offloading of computationally intensive propositions to hardware accelerators is currently being assessed.

The *proposition* reorder mechanism applied to an adverse case study showed a performance improvement of 70% over the original pipeline flow. The implemented mechanism converges to a near optimum solution and can be further improved.

The best performance of pipelined applications is achieved through hybrid HPC/HTC approaches, as preliminary results showed in [5]. Further research is required to evaluate how this two types of parallelization techniques interact and to search for an heuristic to be used in different pipelined applications. A preliminary automatic hybrid parallelization mechanism is expected in the next version of HEP-Frame, together with the inclusion of heterogeneous computing capabilities (with accelerators).

ACKNOWLEDGMENT

This work was supported by FCT (Fundação para a Ciência e Tecnologia) within Project Scope (UID/CEC/00319/2013), by LIP (Laboratório de Instrumentação e Física Experimental de Partículas) and by Project Search-ON2 (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework, through the European Regional Development Fund.

REFERENCES

[1] OpenMP Architecture Review Board, "Openmp Application Program Interface," OpenMP Architecture Review Board, Tech. Rep., 2013.
[2] Intel Corporation, "Threading Building Blocks (Intel TBB)." [Online]. Available: https://www.threadingbuildingblocks.org/
[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, February 2011.
[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.
[5] A. Pereira, A. Onofre, and A. Proença, "Removing Inefficiencies from Scientific Code: The Study of the Higgs Boson Couplings to Top Quarks," *Computational Science and Its Applications – ICCSA 2014*, 2014.
[6] E. W. Weisstein, "Hamiltonian Path," http://mathworld.wolfram.com/HamiltonianPath.html.
[7] ATLAS Collaboration, "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC," *Phys.Lett.*, 2012.
[8] T. A. Collaboration, "The atlas experiment at the cern large hadron collider," *Journal of Instrumentation*, 2008.
[9] The CMS Collaboration, "The CMS experiment at the CERN LHC," *Journal of Instrumentation*, 2008.
[10] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 1st ed. Prentice Hall, 2003.