Removing Inefficiencies from Scientific Code: the Study of the Higgs Boson Couplings to Top Quarks

André Pereira[1,2], António Onofre[1,2], and Alberto Proença[1]

[1] Universidade do Minho, Portugal
[2] LIP-Minho, Portugal
{ampereira, aproenca}@di.uminho.pt, antonio.onofre@cern.ch

Abstract. This paper presents a set of methods and techniques to remove inefficiencies in a data analysis application in the ATLAS experiment. Profiling scientific code helped to pin point design and runtime inefficiencies, the former due to coding and data structure design. The data analysis code in the ATLAS experiment contributed to clearly identify some of these inefficiencies and to give suggestions on how to prevent and overcome those common situations in scientific code to improve the efficient use of available computational resources in a parallel and homogeneous platforms.

Keywords: High Performance Computing, Efficiency Removal, Homogeneous Systems

1 Introduction

The European Organization for Nuclear Research (CERN) is a consortium of 20 European countries aiming to operate the largest High Energy Physics (HEP) experiments in the world. The instrumentation used in nuclear and particle physics research is essentially formed by particle accelerators and detectors. The Large Hadron Collider (LHC) particle accelerator speeds up groups of particles close to the speed of light, in opposite directions, inducing a controlled collision the detectors core. The detectors record various characteristics of the resultant particles of each collision (an event), such as energy and momentum, which originate from complex decay processes of the collided protons. The purpose of these experiments is to test the HEP theories, such as the Standard Model, by confirming or discovering new particles and physics.

The ATLAS experiment is one of the seven particle detectors at CERN whose main goals are to confirm the existence of the top quarks and the Higgs boson in the Standard model. Approximately 600 million collisions occur every second at the LHC. Particles produced in head-on proton collisions interact with the detectors, generating massive amounts of raw data. It is estimated that all the combined detectors produce 25 petabytes of data per year, and it is expected to

grow after the ongoing LHC upgrade. This data then passes a set of processing and refinement stages until it is ready to be used to reconstruct the events by specific data analysis code, which may vary according to the HEP theory being researched. Several research groups work in event reconstruction in the same experiment, enforcing positive competition to produce quality results in a fast and consistent way.

These factors enforce the need to process more data, more accurately, in less time. Research groups often opt to invest on larger computing clusters to improve the quality of their research results. However, most scientific code was not designed and/or developed for an efficient use of the available computational resources. If these applications were adequately tuned (or redesigned), the event analysis throughput could be massively increased. An efficient parallel application can significantly improve its performance at a much lower cost.

This paper addresses inefficiencies in two stages of the data analysis application: the code development and application runtime. In the former, inefficiencies in the algorithm coding and data structuring are pin pointed and several solutions are suggested, based on a quantitative analysis of the bottlenecks. In the latter, inefficiencies in the shared memory parallelization and allocation of computational resources were identified, and a set of optimizations were proposed.

This paper is organized as follows: section 2 briefly presents the top quark and Higgs boson decay process and introduces a short characterization of the data analysis application used as case study; in section 3 the code inefficiencies are identified, analysed, and removed, with a final shared memory parallelization proposal; in section 4, runtime inefficiencies of the parallelization are identified and possible alternatives suggested, concluding with an assessment of the core affinity impact; finally, section 5 concludes the paper and proposes some future work.

2 Top Quark and Higgs Boson Decay

In the LHC, two proton beams are accelerated close to the speed of light in opposite directions, set to collide inside a specific particle detector. This head-on collision triggers a chain reaction of decaying particles, and most of the final particles react with the detector, recording relevant data. One of the experiments being conducted at the ATLAS detector aims to study the top quark and Higgs boson properties. Figure 1 presents the schematic representation of the top quark decay (addressed as the $t\bar{t}$ system).

The ATLAS detector can record the characteristics of the bottom quarks, detected as a jet of particles, and leptons (muon and electron). Neutrinos do not interact with the detector, so, their characteristics are not recorded. Since the top quark reconstruction requires the neutrinos, their characteristics are analytically determined with the known information of the system, through a kinematical reconstruction. However, the $t\bar{t}$ system may not have a possible reconstruction: the reconstruction has a degree of certainty associated, which determines its quality.

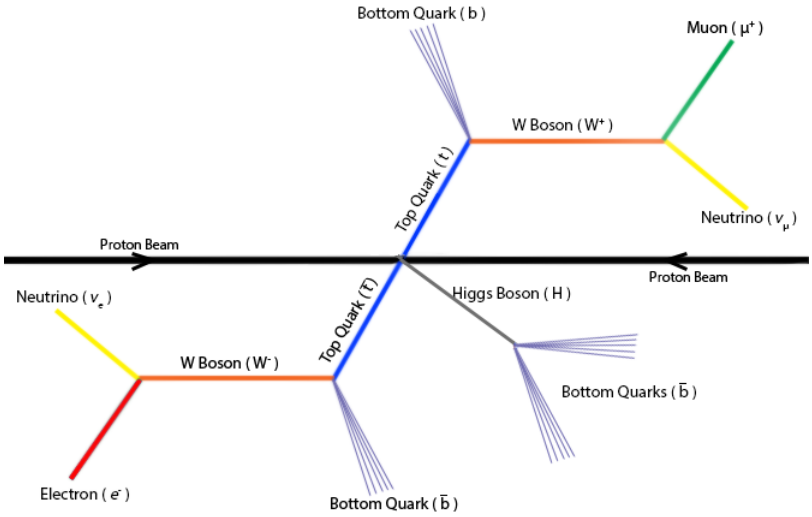

Fig. 1: Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

The amount of bottom quark jets and leptons detected may vary among events, due to other reactions occurring alongside the top quark decay. As represented in figure 1, two jets and two leptons are needed to reconstruct the $t\bar{t}$ system, but the captured data for an event may contain many more of these particles. For the kinematical reconstruction every combination of two jets and two leptons must be evaluated and only the most accurate reconstruction of each event is considered.

If the $t\bar{t}$ system has a possible reconstruction the Higgs boson is reconstructed from the two bottom quark jets that it decays to. This adds two more jets to the event information to test in both kinematical and Higgs reconstructions. The Higgs reconstruction uses the jets that did not provide the most accurate $t\bar{t}$ system. The overall quality of the event processing depends on the quality of both reconstructions.

The ATLAS detector has an experimental resolution of $\pm 1\%$ for each measured value. This error is propagated into the $t\bar{t}$ system and Higgs reconstructions, affecting their accuracy. To improve the quality of the reconstructions several random variations are applied to the measured particles parameters, with a maximum magnitude of $|1\%|$. The quality of the reconstructions and the application execution time is directly proportional to the amount of variations performed per combination. The goal is to do as many variations as possible within a reasonable time frame.

To reconstruct the $t\bar{t}$ system and Higgs boson a data analysis application was developed, the `ttH_dilep`. The application flow is presented in figure 2. Each event data on an input file is individually loaded into a single global state,

shared between the data analysis code and the LipMiniAnalysis library[3], and it is overwritten every time a new event is loaded. The event is then submitted to a series of cuts, which filters events that are not suited for reconstruction. When an event reaches the cut 20 the $t\bar{t}$ system and Higgs boson are reconstructed in the function `ttDilepKinFit`, which is expected to be the most computing demanding. If the $t\bar{t}$ system reconstruction fails, the current combination is discarded and the next is processed. If an event has a possible reconstruction it passes the final cut and its final information is stored.

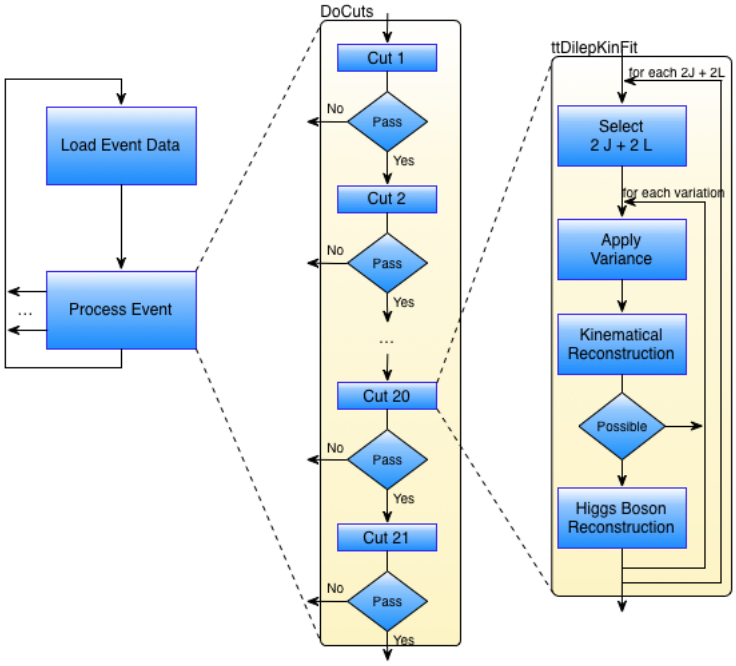

Fig. 2: Schematic representation for the `ttH_dilep` application flow.

The application also depends on the ROOT[4] framework for part of the functionalities used in the reconstructions and for result output and visualisation. The code from both ROOT and the LipMiniAnalysis library cannot be modified as many data analysis applications depend on them.

[3] The LipMiniAnalysis library provides a skeleton to several data analysis applications under study in the Portuguese LIP institution, a CERN partner in the ATLAS experiment.

[4] A C++ framework developed by CERN for helping to produce particle physics data analysis applications by implementing many specific features required by this field.

3 Coding Inefficiencies

Inefficiency removal is a two stage iterative process, where bottlenecks are identified and later removed. First, the application is profiled and analysed to identify the critical sections of the code that take longer to compute. Then, the critical section is optimized by modifying the code, algorithm, or parallelization. The identification of critical sections can be automated by using third party tools, such as gprof, Callgrind, or VTune, which produce reports listing the percentage of time spent in each of the application functions. A more detailed analysis can be obtained using tools similar to PAPI, where hardware counters are used to quantify cache miss rates, executed floating point instructions, and other low level information.

The test environment used in both this section and section 4 is a dual-socket system with two Intel Xeon E5-2670v2 with 10 cores, with hardware support for 20 simultaneous threads, at 2.5 GHz each, 256 KB L2 cache per core and 25 MB shared L3 cache, with 64 GB DDR3 RAM. The K-Best measurement heuristic was adopted to ensure that the only the best, but consistent, time measurements are considered. A 5% interval was used for a k of 4, with a minimum of 12 and maximum of 24 time measurements.

Profiling the data analysis code using Callgrind, the `ttDilepKinFit` was identified as the most time consuming function, taking 99% of the execution time for 1024 variations. `ttH_dilep` execution with this amount of variations was considered reasonable for all efficiency measurements unless stated otherwise, without compromising the application execution time.

A preliminary computational analysis concluded that the application is compute bound on the testbed system, where accesses to the system RAM memory are not a limiting factor with a ratio of 7 instructions per fetched byte for 1024 variations.

An analysis of the code showed two major inefficiencies restricting the performance. The pseudo-random number generation is consuming a large part of the `ttDilepKinFit` execution time due to coding inefficiencies. Also, when planning the parallel version it was concluded that part of the event data was stored in a LipMiniAnalysis data structure, preventing the parallel processing of events. These two inefficiencies will be addressed in more detail through the next sections.

3.1 Pseudo-Random Number Generation Inefficiencies

Pseudo-random number generators (PRNGs) are common in many Monte Carlo simulation and reconstruction applications. A good PRNG deterministically generates uniform numbers with a long period, its produced values pass a set of randomness tests and, in HPC, it must be efficient and scalable. Repeatability is ensured by providing a seed to the PRNG prior to number generation, due to their deterministic execution.

The variation for the kinematical and Higgs reconstructions is based on applying a random offset to the current particles characteristics. This offset has a

maximum magnitude of ±1% of the original value and is computed by a PRNG. An analysis of the callgraph produced for 256 variations (higher variations made the Callgrind execution time infeasible) showed that 63% of `ttH_dilep` execution time was spent on the PRNG. However, 23% of the time was spent defining a new seed for the PRNG. Figure 3 presents the callgraph for the `ttDilepKinFit` function of `ttH_dilep`.

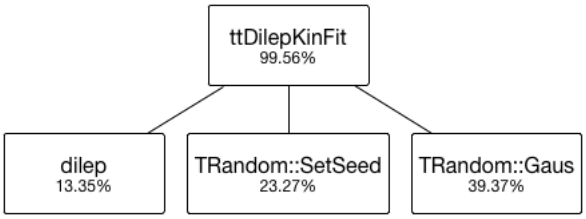

Fig. 3: Callgraph subset of the `ttDilepKinFit` most time consuming sections for 256 variations per combination.

An analysis of the code showed that the application uses a PRNG available in ROOT, which uses the Mersenne Twister algorithm, resetting the seed for every parameter variation. The Mersenne Twister period is approximately $4.3 * 10^{6001}$, while the maximum amount of pseudo random numbers generated by the application, for the input file used and 512 variations, is $1.5*10^9$, making the seed reset unnecessary. The removal of this inefficiency granted a 71% performance improvement, for 1024 variations.

3.2 Data Structure Inefficiencies

Once removed the PRNG seed reset inefficiency, the `ttDilepKinFit` still remained the critical region in the application, with no apparent code inefficiency. The most obvious solution is to process in parallel several events from the same input file. However, the function in LipMiniAnalysis that loads events from a file into memory assigns a single global space. This data structure contains information that is modified during the event reconstruction process, and it is overwritten for every event loaded. Changing the data structure to support multiple events in memory simultaneously, and loading all events in the input file at the beginning of the data analysis, would allow for a parallelization of the event processing, with a very small overhead. However, as mentioned in section 2 many data analysis applications depend on LipMiniAnalysis preventing any modifications to its structure, so alternative solutions were explored.

Next step to improve the code execution time is to parallelize `ttDilepKinFit`. Note that it is not possible to parallelize the whole event processing since only one is loaded at a time and part of its information is stored in LipMiniAnalysis

library. Besides not allowing this parallelization, reading events individually is more inefficient than reading all events at once, where in the former slower random reads are made on the hard drive and in the latter the fast sequential reads are used.

Optimization Approaches Parallelizing `ttDilepKinFit` implies modifying its flow. Currently, the data of each variation is overwritten when processing each different combination of an event, so a data structure holding all combinations of the event is necessary. Picking a lepton/jet combination depends on all previous combinations chosen, which serializes the construction of the data structure. Each parallel task (indivisible work segment) selects a combination with variations still to compute, then varies the particles parameters, performs the kinematical reconstruction, and attempts to reconstruct the Higgs boson. A parallel merge is performed after all combinations are computed to get the most accurate reconstruction for the event. Figure 4 presents the sequential and parallel workflow for `ttDilepKinFit`.

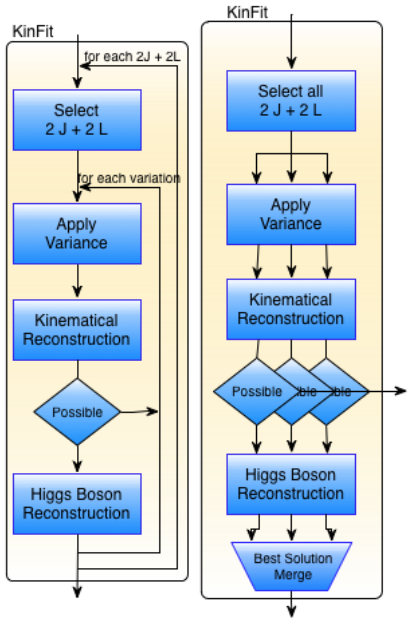

Fig. 4: Schematic representation of the `ttDilepKinFit` workflows: sequential (left) and parallel (right).

A shared memory parallelization using OpenMP was devised, as it is the bes approach for single shared memory systems. The parallel tasks are grouped into threads, which holds the best reconstruction to minimize the complexity

of the merge by reducing through all the threads instead of tasks. The amount of tasks for each thread is balanced dynamically by the OpenMP scheduler, as the workload is irregular since the Higgs boson reconstruction execution is not always computed. Each thread has a private PRNG initialized with different seeds to avoid correlation between the numbers generated.

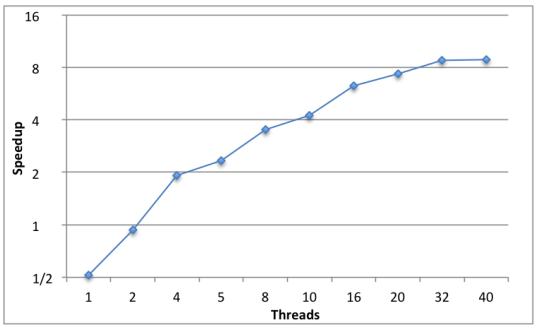

Fig. 5: Speedup for the `ttH_dilep` original parallel version of the application.

Figure 5 presents the speedups for various variations and threads. The purpose of the 1 thread test is to evaluate the parallelization overhead. The application has a constant scaling up to 64 variations but tends to stabilize for more variations. The best efficiency occurs when using 2 and 4 threads, where the application is almost using all resources from the cores used. The best overall performance occurs for 40 threads, but it only offers a speedup of 8.8, underusing the available 20 physical cores. Note that there is no significant overhead due to NUMA[5] accesses, as seen by the constant increase in performance from 10 to 16 threads.

The lack of scalability beyond a low number of parallel threads suggests that inefficiencies may still affect the application, probably caused by the parallelization overhead. Intel's VTune was used to search for hotspots (bottlenecks) on the parallel `ttH_dilep`, since this tool is best suited for profiling parallel applications while providing a user friendly graphic interface. A preliminary analysis showed that the application was spending 20% of the execution time building the combination data structure for 256 variations.

An analysis of the coded data structure showed that inefficiencies were affecting the performance in specific situations. Data that is read-only on the parallel section is being replicated in each element of the data structure. If the elements were to share a pointer to such data, the overhead of constructing the data structure would be reduced. However, this could lead to worse cache management,

[5] NUMA, Non-Unified Memory Access, since each Xeon device has its own memory controller with attached RAM: RAM access time for each core differs as the RAM is connected to the same device or the neighbour Xeon.

due to cache line invalidations proportional to the number of threads, resulting in added overhead, specifically in NUMA environments where the communication cost is increased. Nevertheless, this optimization was implemented and tested (addressed as *pointer version*), with its speedups ploted in figure 6. The reference value for the speedup computation is still the same sequential version.

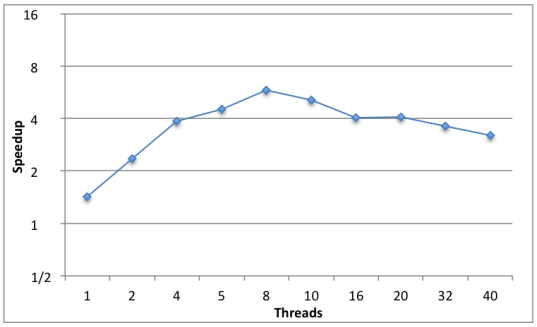

Fig. 6: Speedup for `ttH_dilep` parallel pointer version application.

As predicted, the best speedup occurs when using only one CPU device, namely for 8 threads. Even with 10 threads the performance of the application suffers due to the increase of concurrent accesses to the shared L3 cache on the device. This decrease in performance is even more noticeable when using both CPU devices, where the non-pointer version still scales while this version performance is worse than using a single CPU device. However, this implementation is more efficient than the former when using only one device. Note that the superlinear speedups is due to the reduction in the data that each thread has to process, making it suitable to be stored in the private L2 cache of each core, avoiding the slower accesses to the L3 cache.

4 Runtime Inefficiencies

When submitting a job or application for execution on a given computing system, most users trust the default configurations of the submission environment. However, if the user needs to improve the efficiency of the code execution, he/she must be aware of the environment variables that can be controlled and how those can impair the performance. Two cases will be addressed here: (i) how to spread the code parallelism, between processes and threads, and (ii) how to allocate the available cores on each device to threads and processes.

4.1 Multithreading Inefficiencies

Without the sensibility provided by the tests in section 3, a scientist would incur in the pitfall of using all available cores on the system (and even all hardware

threads, if each core supports hardware multithreading), hoping that it would provide the best performance. While it may be true for the non-pointer implementation, the system computational resources would be inefficiently used, and using the single device highly efficient pointer implementation would induce a even greater waste.

A closer look to the pointer based implementation shows in fact that it is the most efficient one. As seen in section 3.2, the scalability of the parallelization is limited by the NUMA organization on modern multiple CPU device systems. If the threads on cpu_1 do not share information with the threads on cpu_2, the NUMA bottleneck is removed by using multithreaded processes. However, parallelization at the process level, where each process performs a data analysis on a separate event, is not possible with the current implementation of LipMini-Analysis, where a single global state is allocated to store data from each event processing.

Data analysis applications are individually executed for each file (around 1GB in size) in a very large set of files, at a terabyte scale, received weekly from CERN. An alternative approach to the process parallelization over a single input data file is to balance the execution of different `ttH_dilep` processes in the system on a set of distinct input files. This reduces the complexity of the implementation, with no changes needed for `ttH_dilep`, and avoids communication between processes. A simple scheduler was devised, which takes a set of input files and spawns a given amount of `ttH_dilep` processes. The scheduler dispatches the files to the different processes in a queue-like approach, and monitor their execution as shown in figure 7. A set of 20 input files was considered for testing and evaluation purposes, with different configurations of processes and threads per process.

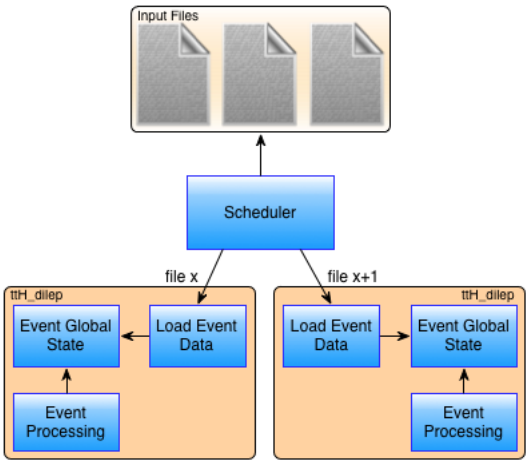

Fig. 7: Schematic representation of dispatcher workflow.

Figure 8 presents the speedups using 2, 4, 5, 8, and 10 processes for various thread configurations, with maximum number of threads limited to 40. A higher amount of processes was not tested as the efficiency decayed from 8 to 10 processes. The best speedups occur for 8 processes with 5 threads each, with a peak of 69.3, 7.8 and 11.7 times better than the best non-pointer and pointer implementations, respectively. A small number of threads such as this allows for a small overhead in the `ttH_dilep` parallelization, namely on load balancing and final best reconstruction merge for each event. For 10 processes the load on the system due to the lack of shared memory and I/O operations affects the performance, decreasing the speedups relatively to using 8 processes. A common behaviour is that when using the CPU devices hardware multithreading the speedups tend to stabilize, or even drop in the case of 2, 4, and 5 processes.

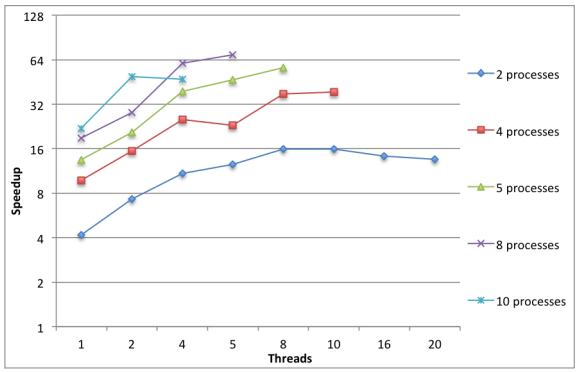

Fig. 8: Speedups for the scheduler with the pointer based implementation for 2, 4, 5, 8, and 10 processes and various threads per process.

4.2 Core Affinity Inefficiencies

The key issue in runtime efficiency is the thread affinity, namely to, control the allocation of each thread to which CPU core. By default, OpenMP lets the operating system to manage the thread affinity; as consequence, threads may migrate among cores during runtime. If a thread is running on core c_1 and moves to core c_2, all data on the private cache l_{c_1} needs to be reloaded to cache l_{c_2}, causing unnecessary overhead. This effect is amplified if the threads are moved between adjacent CPU devices. When multiple different, and (possibly) parallel processes are running on the same system, which is common in production environments, such scheduling occurrences happen more often.

Defining the thread affinity of an application may provide a more predictable, or in some cases better, performance. In theory, an optimum thread affinity scheme allocates the threads to contiguous physical cores of one CPU device, uses

the cores of the second CPU device only after the first is filled, and finally uses the multithreading after filling all physical cores. Note that using multithreading before the second CPU device is fully occupied may provide better performance in memory bound applications. This type of affinity must be defined prior to the application execution and depends on the system used. In this data analysis case the affinity was specifically tuned to this 20-core system for all threads or process/threads configurations for the scheduler.

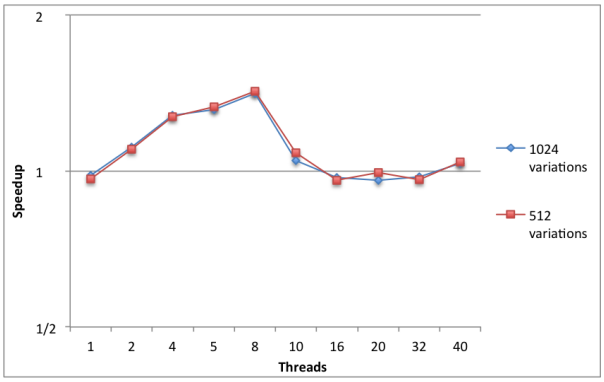

Fig. 9: Speedup of the `ttH_dilep` parallel pointer implementation with core affinity.

By analysing the speedups of the pointer implementation of `ttH_dilep` with thread affinity, in figure 9, the specification of the affinity provides speedups for the previous most efficient number of threads, i.e., up to 8 threads. For 8 threads the performance increases by 41%, relative to its no affinity counterpart. With this number of threads, and the amount of shared data, moving threads between cores at runtime causes more cache warm ups to occur, significantly affecting the performance. When using more than 10 threads the application is roughly 4% slower as the operating system uses some multithreaded cores rather than using all available physical cores and it does a better job at managing the multithreading.

The same affinity study was performed on the scheduler for 512 and 1024 variations per combination, with their speedups presented in figure 10. For 1024 variations the performance is increased for specific configurations, only diminishing the performance of 5 processes, peaking at 52%, 90%, 8%, and 25% for 2, 4, 8, and 10 processes respectively. For 512 variations the improvement is more constant but smaller, peaking at 72%, 28%, 37%, 11%, and 5% for 2, 4, 5, 8, and 10 processes respectively. There are only few cases that the performance is worse.

The performance with core affinity is less susceptible to oscillations, as with no affinity it is sometimes affected by OS thread reallocations. It is when the reallocations occur that setting the core affinity provides the best performance.

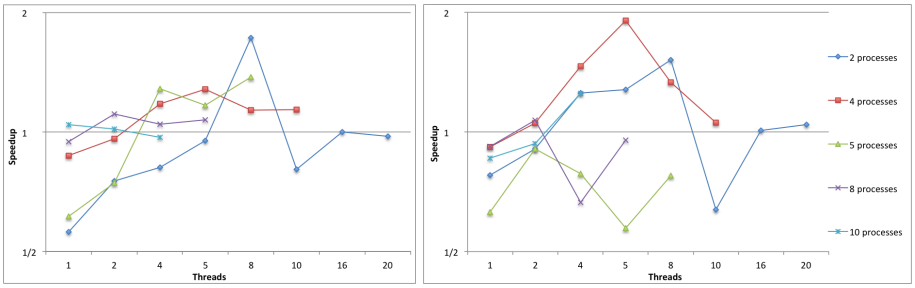

Fig. 10: Speedups of the scheduler with the pointer based implementation for 512 (left) and 1024 (right) variations and various threads per process with core affinity.

Otherwise it does not allow to use multithreading (unless it is in the scheme) for hiding the memory accesses latency, affecting the performance. For 512 variations, the applications execution time is smaller meaning that the impact from thread reallocations is higher, benefiting from setting the core affinity. It is not possible to use a theoretical affinity scheme for always improving the performance on every system, as it is highly dependent on:

- The algorithm, as memory bound algorithms suffer more from core reallocation due to the losing all data on cache, where fixing their position on a specific core avoids unnecessary accesses to the RAM;
- The application execution time, as the impact from thread reallocations is higher in faster applications;
- The operating system, as OpenMP, by default, lets it manage the thread allocation and it is susceptible to the overall system load, causing fluctuations in consecutive applications execution time.

With all optimizations considered, the best overall performance in a dual 10-core Xeon system is obtained using the scheduler combined with the pointer implementation, with 8 multithreaded processes (with 5 threads each), reaching a speedup of 113 over the original sequential application.

5 Conclusion

In this paper we presented a study of the performance inefficiencies in scientific code, using a particle reconstruction analysis application as a case study. Top quark and Higgs boson studies require reconstructing form measurements of a very large number of particle collisions, performed weekly by the `ttH_dilep` application in terabytes of data. A faster and more refined analysis of the data allows to better reconstruct more particle collisions and improve the quality of the physics research.

We identified and removed inefficiencies in different stages of the application. In the code design, tackling inefficiencies in the pseudo-random number

generation, a common component of most simulation and analysis applications, which provided a 71% increase in performance. In data structure design, by analysing the factors limiting the particle reconstruction parallelization, and presenting and testing two different solutions for shared memory environments. The former, with a constant scaling on a dual-socket NUMA system, providing a maximum speedup of 8.8, while the latter, more efficient, with a peak speedup of 5.8 but only using one CPU device, compared to the optimized application. Finally, at application runtime, where a multiprocess approach using a more efficient parallel implementation was introduced to tackle its inefficiencies on NUMA systems, providing a maximum speedup of 69.3. An efficient control on the thread affinity of the more efficient parallel implementation and multiprocess approach, provided improved performances for 512 and 1024 variations, with a peak improvements up to 37% and 90%, respectively. However, the fluctuation in performance and the dependencies on many system characteristics prevent providing a generalized heuristic to aid to control the best affinity for the application, whatever the computing system.

We hope to improve the sensibility of scientists to the efficiency pitfalls common in scientific code, to help develop more efficient and performing applications. The performance of the `ttH_dilep` application was improved by a factor of 113, for the test system used, helping physicists to execute more particle collisions with a more refined reconstruction process, which efficiently uses the available computational resources.

Even with the interesting improvements to the application efficiency, some directions of future research were identified. The scheduler could be improved to automatically predict the best process/thread configuration for each system by analysing a set of micro-benchmarks or the application itself on a small input, and ultimately identify the best core affinity scheme. Also, the application efficiency could be improved using hardware accelerators, balancing the workload among accelerators and CPU devices in heterogeneous systems. Frameworks for parallelization and workload balancing for heterogeneous systems can be analysed and tested with this case study.

Acknowledgments:

References