Removing Inefficiencies From Scientific Code: A Case Of The ATLAS Experiment

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 for removing inefficiencies in scientific code. A scientific application, of the ATLAS experiment, is used as a case study. Here, we implement and test a set of optimizations to deal with performance inefficiencies in a organized approach: identification, removal at application development and at application runtime. These optimizations consider code, algorithm and data structure design, parallelization in shared memory systems, and runtime configurations for NUMA environments. These optimizations target different multithreading and multiprocess combinations, and CPU affinities. We expect that this communication will aid scientists to become more aware of common efficiency pitfalls in scientific code.

Keywords: High Performance Computing, Efficiency Optimization

1 Introduction

The European Organization for Nuclear Research (CERN) is a consortium of 20 European countries, founded in 1954, with the purpose of operating the largest High Energy Physics (HEP) experiments in the world. The instrumentation used in nuclear and particle physics research is essentially divided into 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, resulting in a controlled collision inside the detectors (each collision is considered an event). The detectors record various characteristics of the resultant particles, 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, a key project at CERN, is conducting most of the crucial experiments on both the top quark and Higgs boson measurements. Approximately 600 millions of collisions occur every second at the LHC. Particles produced in head-on proton collisions interact with the detectors of the ATLAS experiment, generating massive amounts of raw data as electric signals. It is estimated that all the detectors combined produce 25 petabytes of data per year,

and it is expected to grow after the LHC upgrade, which purpose is to increase the amount of energy of the accelerated particle beams. This data passes a set of processing and refinement until it is ready to be used to reconstruct the events by specific applications. The application varies according to the HEP theory being analysed. At this stage, different research groups in the same experiment enforce positive competition to produce quality results in a fast and consistent way.

These factors enforce the need to process more data, more accurately, and in less time. Research groups often opt to increase computational resources in a brute force approach to improve their research quality. However, applications use inefficiently the available computational resources, so, if properly optimized, the computational throughput could be highly boosted. A properly optimized application can provide an equal or greater performance increase at a much lower cost. This paper aims to provide a set of techniques to identify and remove inefficiencies in scientific applications, using a top quark and Higgs boson analysis application as a case study, to help scientist develop and optimize applications that efficiently use the available resources.

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 `ttH_dilep` application used as case study; section 3 presents and characterises the inefficiencies identified in the application; section 4 shows the process of removing the inefficiencies and optimizing the application at development and runtime stages; finally, section 5 concludes the paper and mentions future work.

2 Top Quark and Higgs 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. From this head-on collision results a chain reaction of decaying particles, and most of the final particles react with the detector allowing their characteristics to be recorded. One of the experiments being conducted at the ATLAS detector is related to the studies of 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). However, neutrinos do not interact with the detector and, therefore, 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, in a process known as 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.

The amount of Bottom quark jets and leptons detected may vary between events, due to other reactions occurring alongside the top quark decay. As represented in figure 1, 2 jets and 2 leptons are needed to reconstruct the $t\bar{t}$ system, but the input data for an event may have many more of these particles associ-

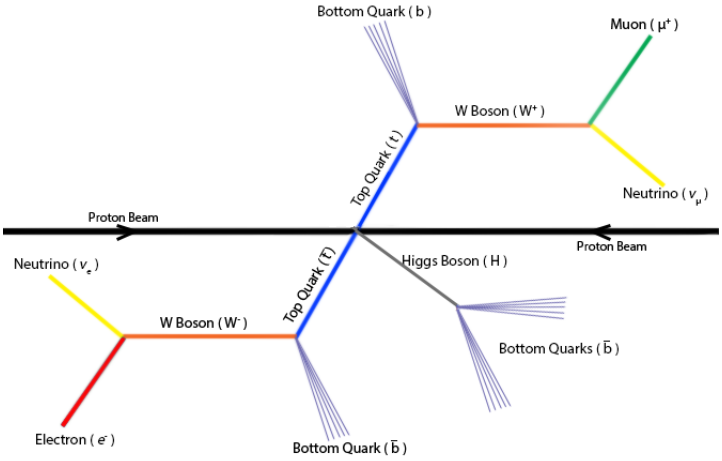

Fig. 1: Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

ated. It is necessary to reconstruct the system for every combination of 2 jets and 2 leptons in the input data (referred only as combination). Then, only the most accurate reconstruction of each event is considered.

The Higgs boson is reconstructed from the two jets that it decays to, but only if the $t\bar{t}$ system as a possible reconstruction. This adds at least two more jets to the event information, increasing the number of possible combinations, as they are the same as the $t\bar{t}$ system jets. The Higgs boson reconstruction uses the jets that were not needed in the $t\bar{t}$ system. The overall quality of the event reconstruction depends on the quality of both $t\bar{t}$ system and Higgs boson reconstructions.

The ATLAS detector has an experimental resolution that induces an error of 2% in each measure of the particle characteristics. This error is propagated into the $t\bar{t}$ system and Higgs reconstructions, affecting their accuracy. To improve the quality of the reconstructions a random variation to the particles parameters is applied, with a maximum magnitude of 2%, a given amount of times and chose the event reconstruction with the highest accuracy. The quality of the reconstructions and application execution time are directly proportional to the amount of variations performed per combination. The goal is to do as many variations as possibly within a reasonable time frame.

The `ttH_dilep` application was designed to reconstruct of the $t\bar{t}$ system and Higgs boson, as explained above. The application flow is presented in figure 2. Each event on the input file is individually loaded to a single global state shared among the application and the LipMiniAnalysis library, 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 a function named

`ttDilepKinFit`. 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.

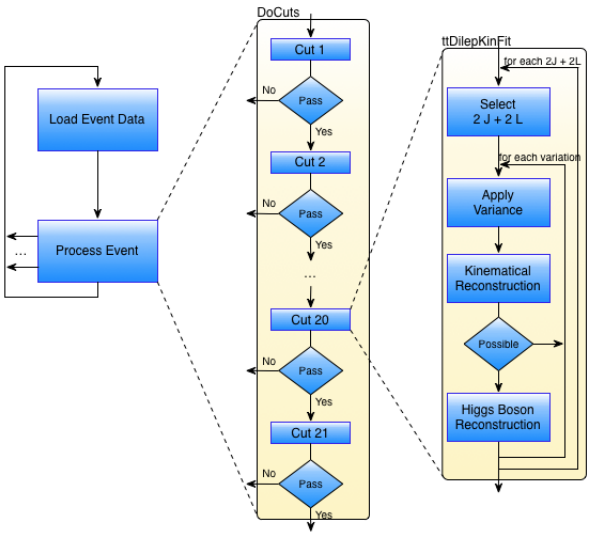

Fig. 2: Schematic representation for the `ttH_dilep` application flow.

3 Code an Algorithm Inefficiencies

Inefficiency removal is a two stage iterative process. First, the application is analysed to identify the critical sections of the code that take the most time to compute. This can process can be automatised 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 measured to obtain cache miss rates, float 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 E5-2670v2 with 10 cores (20 hardware threads) at 2.5 GHz each, 256 KB L2 cache per core and 25 MB shared L3 cache, coupled with 64 GB DDR3 RAM. The KBest policy 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.

In a preliminary analysis using Callgrind, the `ttDilepKinFit` was identified as the most time consuming function, specially when considering a significant number of variations. Figure 3 shows the relative execution time for the

`ttDilepKinFit`, I/O of event loading, and the rest of the computations (the remaining cuts). `ttH_dilep` execution with 1024 variations was considered for all efficiency measurements, as it is a reasonable amount without an exaggerated compromise of the application execution time. Note that the application is dependant on the ROOT framework and LipMiniAnalysis library, which code cannot be modified.

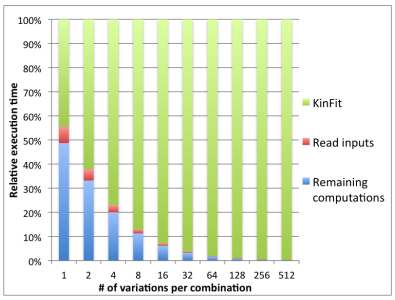

Fig. 3: Relative execution time of the critical sections of the `ttH_dilep` application.

A preliminary computational analysis determined that the application is compute bound on the test system, where accesses to the system RAM memory are not a limiting factor with a ratio of 8 instructions per byte fetched for 512 variations.

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 values produced 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 2% of the original value and is determined by a PRNG. An analysis of the callgraph produced for 256 variations (higher variations made the Callgrind execution time infeasible) revealed that 63% of `ttH_dilep` execution time was spent on the PRNG. However, 23% of the time was spent on setting the seed for the PRNG. Figure 4 presents the callgraph for the `ttDilepKinFit` function of `ttH_dilep`.

An analysis of the code revealed that the application uses a PRNG available in ROOT, which uses the Mersenne Twister algorithm, resetting the seed in every variation. The Mersenne Twister period is approximately $4.3 * 10^{6001}$,

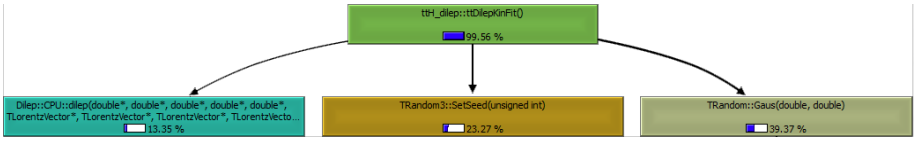

Fig. 4: Callgraph subset of the `ttDilepKinFit` most time consuming sections for 256 variations per combination.

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 an increase of 71% in performance for 1024 variations.

3.2 Data Structure Inefficiencies

Even with the PRNG optimization the `ttDilepKinFit` remains the critical region in the application. An analysis of the functions code revealed that there are no relevant code inefficiencies left, so the next step 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, which we did not have permission to refactor. 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 5 presents the sequential and parallel workflow for `ttDilepKinFit`.

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.

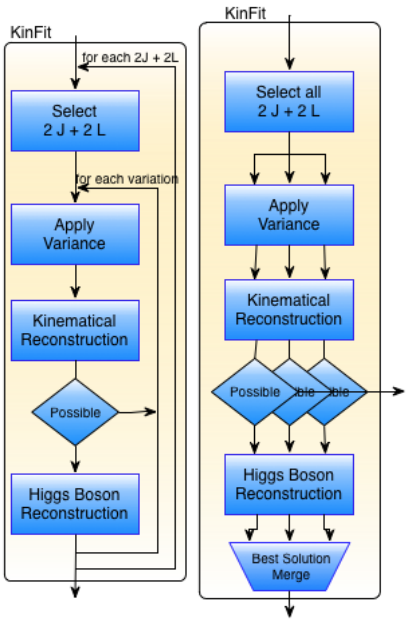

Fig. 5: Schematic representation of the `ttDilepKinFit` sequential (left) and parallel (right) workflows.

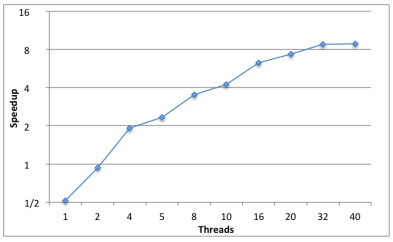

Fig. 6: Speedup for `ttH_dilep` parallel non-pointer version application.

Figure 6 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 accesses, as seen by the constant increase in performance from 10 to 16 threads.

The lack of performance means that there are still inefficiencies on 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 an easy to use GUI. A preliminary analysis revealed that the application was spending around 20% of the time building the combination data structure for 256 variations.

An analysis of the data structure code revealed that there were inefficiencies that 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, 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 presented in figure 7.

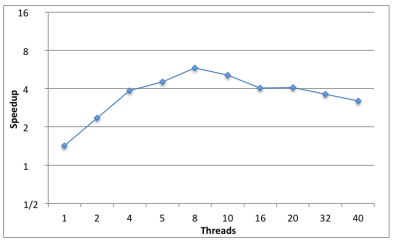

Fig. 7: Speedup for `ttH_dilep` parallel pointer version application.

As predicted, the best speedup occurs when using only one CPU device, specifically for 8 threads. Even with 10 threads the performance of the application suffers due to the increase of concurrent accesses to the L3 cache on the system due to the shared data. 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 with one 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 L3 cache.

4 Runtime Inefficiencies

The identification and removal of inefficiencies follows the same iterative approach presented in section 3.

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

A closer look to the pointer implementation is needed since it is the most efficient. As seen in section 3.2, the scalability of the parallelization is restricted by the NUMA memory accesses required by the shared memory. If the threads on cpu_1 do not share information with the threads on cpu_2 the bottleneck is removed, which can be achieved by using multithreaded processes. An MPI paralllelization at the event level is impossible by the reasons referred in section 3.2.

Analysis applications are executed individually for each file (around 1GB in size) in a huge set of files, usually reaching the terabyte scale, received from CERN each week. An alternative approach to the MPI parallelization is to balance the execution of different `ttH_dilep` processes in the system on a set of 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 creates a given amount of `ttH_dilep` processes. It is then responsible for dispatching the files to the different processes in a queue-like approach, and monitor their execution, as shown in figure 8. A set of 20 input files was considered for testing purposes, with different configurations of processes and threads per process.

Figure 9 presents the speedups using 2, 4, 5, 8, and 10 processes for various thread configurations, with maximum number of threads was limited to 40. The best speedups occur for 8 processes with 5 threads each, with a peak of 55.6, 6 and 10 times better than the 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. 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.

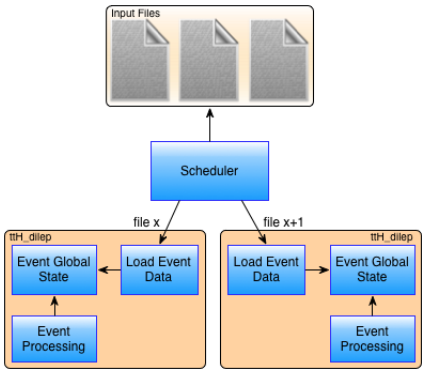

Fig. 8: Schematic representation of dispatcher workflow.

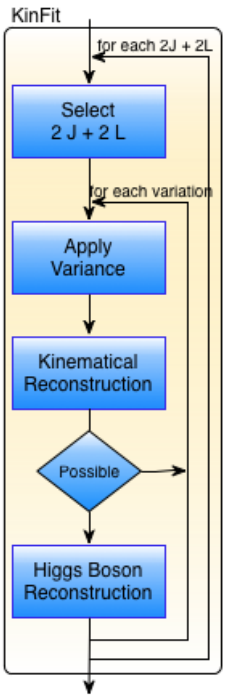

Fig. 9: Speedups for the scheduler with 2, 4, 5, 8, and 10 processes with various thread configurations.

4.2 Core Affinity Inefficiencies

The final possible optimization at runtime regards the thread affinity, i.e., control to which CPU cores a given thread is allocated. By default OpenMP allows for the Operating System to manage thread affinity, allowing for threads to be changed between cores during runtime. If a thread is running on core c_1 and it is moved to core c_2, all the 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 CPU devices. When multiple different, and possibly parallelized, processes are running on the same system, which is common in production environments, occurrences of bad scheduling occur more often.

Defining the thread affinity of an application may provide a more constant, 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 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 case the affinity was specifically tuned to this system for all threads, or process/threads configuration for the scheduler.

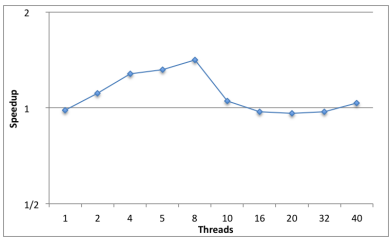

Fig. 10: Speedup of the `ttH_dilep` parallel pointer implementation with affinity *vs* no affinity.

By analysing the speedups of the pointer implementation of `ttH_dilep` with thread affinity, in figure 10, it is concluded that specifying the affinity is 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.

5 Conclusion

Acknowledgments:

References