Benchmarking and optimising large scale parallel workflows

August 2019

AUTHOR(S):
Jesús Perales Hernández

SFT Group

SUPERVISOR(S):
Guilherme Amadio
Danilo Piparo

CERN openlab

PROJECT SPECIFICATION

The main idea of this project is to carry out performance analysis on the RDataFrame class within the ROOT operational framework. For this purpose, scalability analysis are performed on the execution of the Dimuon and Higgs tutorials evaluating concretely how the source code of these tutorials scales with different number of execution threads.

Among others, it analyzes the comparison between compiled code and JIT (Just-In-Time), different types of storage (NVMe, SSD, EOS), memory usage and even measures the impact that NUMA effects have on performance.

TABLE OF CONTENTS

1. INTRODUCTION

In order to carry out the purposes established in the project specification, we make use of a series of configurations, tools, etc. which are detailed below:

The server where the different analyses are executed is an Intel Xeon E5-2698 dual socket with 16 cores / socket, 64 GB of RAM, 2.3GHz base and 3.6GHz boost.

VTune 2019 and Perf have been used as profiling tools. In addition, a tool from Brendan Gregg (a Senior Performance Architect) has been used to generate flamegraphs in order to visualize the execution stack of the tutorials and be able to identify which functions are executed and the time they consume.

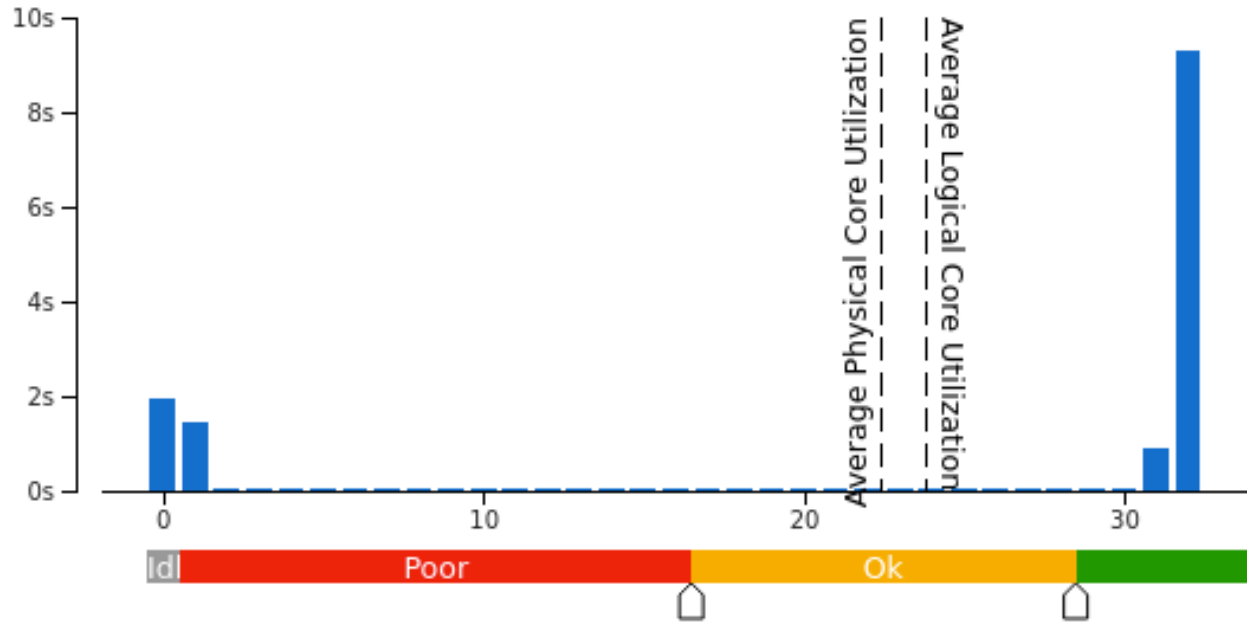
In addition, in order to carry out performance and scalability analysis and to get meaningful information to draw conclusions, we had to translate the source code from JIT code to compiled code in some cases.

2. DIMUON ANALYSIS EXECUTION

This tutorials consists in matching muon pairs and producing an histogram of the dimuon mass spectrum showing resonances up to the Z mass. This tutorial is the first one we want to execute and benchmark.

a. CORE USAGE

Beginning with performance analysis and the use of profiling tools such as VTune, one of the main metrics that we are interested in knowing regarding the execution of tutorials is the one that indicates the workload to which the server's computational resources are subjected.

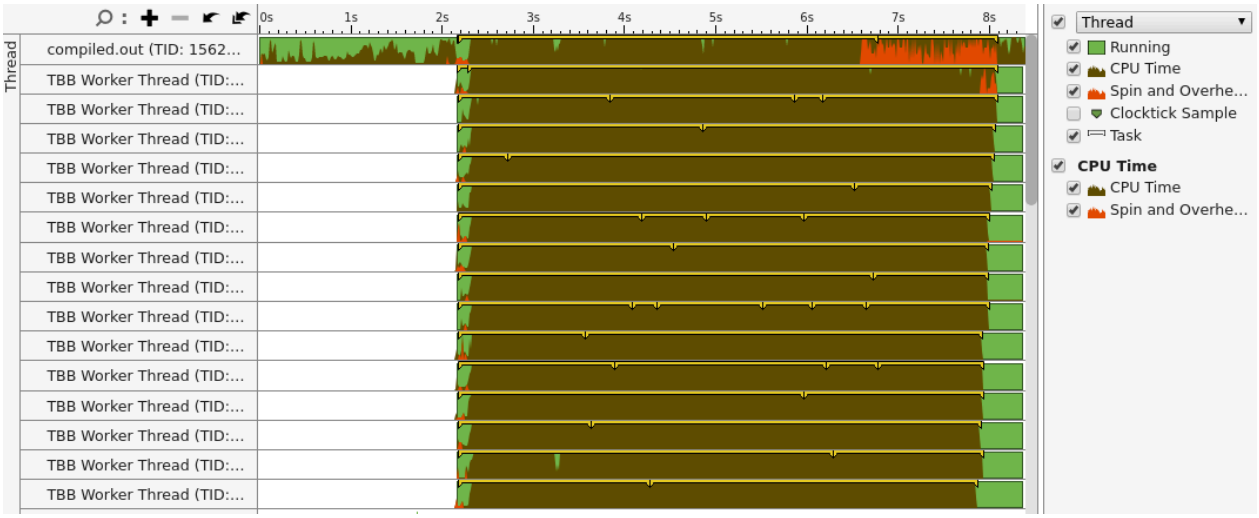


In this case we can observe that, of the 32 physical cores that the machine has, most of the time they are being used and carrying out some task. However, there is a percentage of the time that only 1 of the cores is being used. Apparently, this is due to the tasks of initialization, reading/writing, etc.

b. THREAD ACTIVITY

Knowing the data provided by the graph of the previous section, we decided to check which is the activity of the threads assigned to the execution of the tutorial. In this case, 16 threads are used which, ideally, should run in 16 independent cores.

In the figure, the brown colour inside an execution thread indicates that this thread is doing some computational task. That is, it is not on hold for any input/output or blocked process.



As we can see at first glance, all threads are performing computational tasks and are not kept blocked or waiting for input/output operations.

We can also see how one of the threads assumes the largest workload at the beginning of the execution to perform initialization tasks. Similarly, at the end, this same thread is in charge of performing some other tasks.

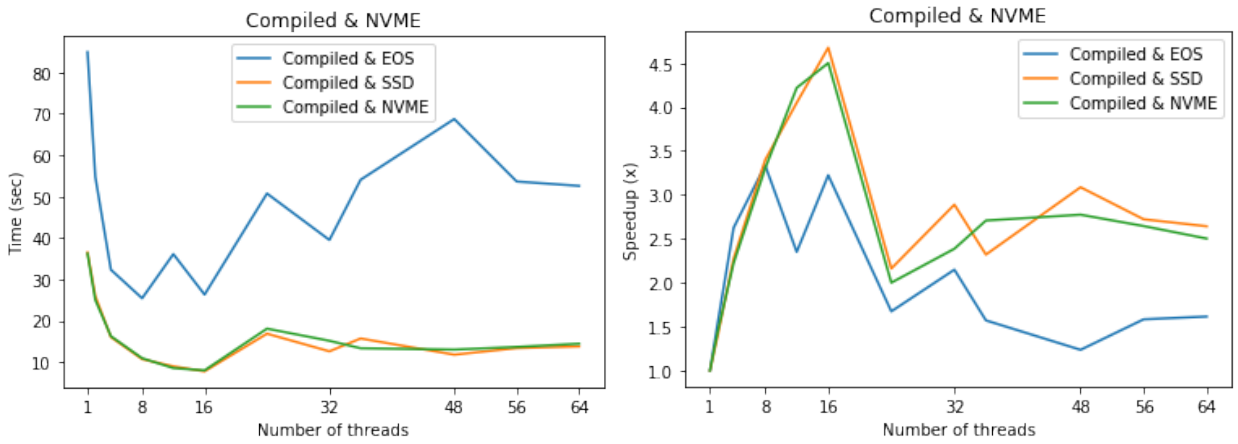
c. SCALABILITY (TIME & SPEED-UP)

In addition to all this, we wanted to run some scalability analysis to see how the code present in this tutorial scales as we increase the number of threads that we want to use to run it. To minimize the influence of external factors, we have run these scalability analyses taking the average of three runs for each of the experiments.

In this particular case we are interested in knowing how the type of storage in which the files used by the tutorial (EOS, SSD, NVME) are located affects the execution time. For this experiment we have made use of the tutorial in compiled code instead of JIT.

The speed-up is the number of times one execution runs faster or slower than another. Let's suppose that we have two executions of the same program: the first of them, making use of only 1 thread and a second that makes use of 8 threads. Ideally, this second should run 8 times faster than the first, although we will see later that this does not always happen.

In the abscissa axes (X axis) we can find the number of execution threads used for each execution, which can be: 1, 2, 4, 8, 16, 24, 32, 36, 48, 56, 64 while in the ordinate axes (Y axis) we can find the execution time (left graph) and the speed-up (right graph).

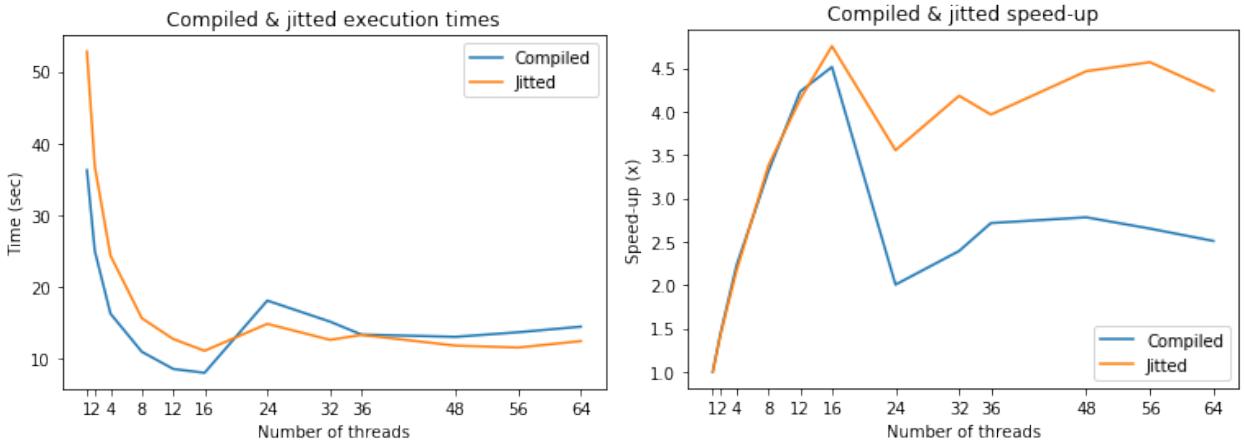


As we can see, the worst execution time corresponds to the execution with EOS (blue line). It is also especially noticeable that it is the storage that has more variability. This is normal since it is a network storage and has a lot of influence that is beyond our control.

On the other hand, the two fastest executions correspond to the SSD and NVME solid state disks. Accordingly, the speed-up is the highest also in these two, with no significant difference between them.

d. SCALABILITY (JITTED & COMPILED)

Continuing with the section on scalability analysis, we also wanted to check the differences that exist between the two types of source code we have: compiled code and JIT code.



As the graph on the left shows us, there is no significant difference in elapsed time between a compiled code execution and a JIT code execution. However, the execution time in JIT code is higher than the compiled code up to 16 threads while from that point on, this time is lower.

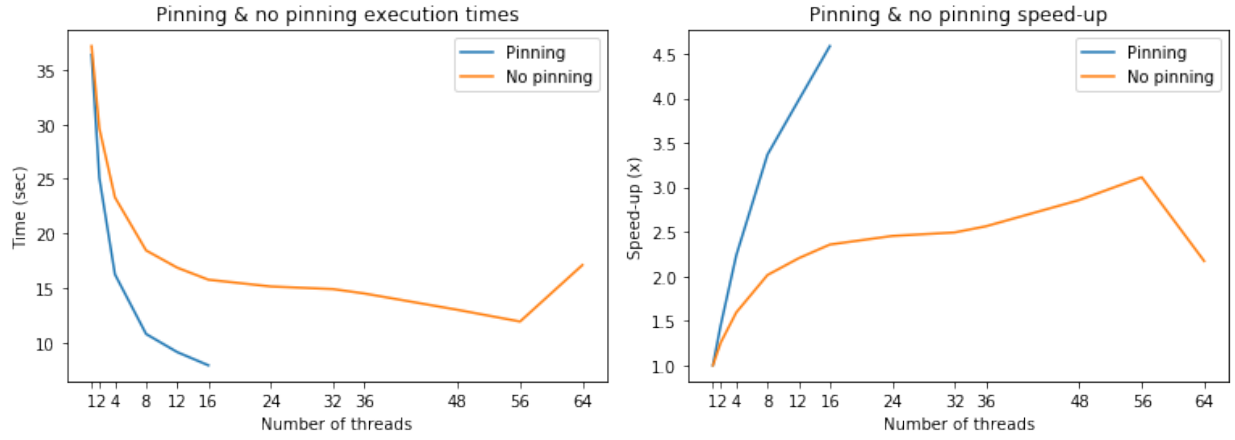
On the other hand, it is especially remarkable to observe that the JIT code scales much better than the compiled code, getting approximately twice the speed-up.

e. SCALABILITY (PINNING & NO PINNING)

To finish the scalability analysis, we also decided to test what is the difference between running the tutorial with pinning against no pinning.

Before advancing to the graphs to see the results, we must know what it is to pin the execution of a program. When we do pinning, basically what we are saying to our program is: You are using X execution threads and I want those execution threads to remain in certain physical cores of the processor, without being able to move freely through the CPU.

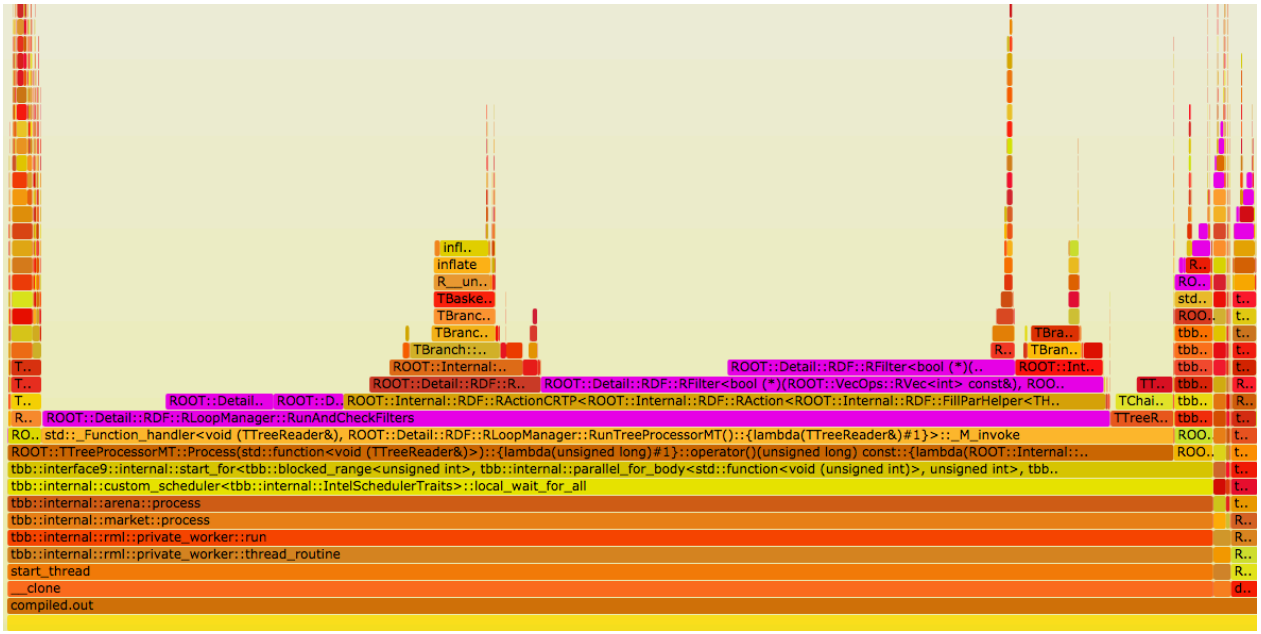
When pinning is not activated, the execution threads move freely through the CPU according to a scheduler. However, pinning can only be activated when we use 16 threads or less since, for our server, it is the maximum number of cores that 1 socket of the processor has.



According to the graphs, there is a very significant difference between activating the pinning or not activating it. When we execute with pinning, the execution time is much shorter. In the same way, the code tends to scale much better, getting almost double the speed-up in the execution with pinning.

f. FLAMEGRAPH

Once the different analyses have been executed, we are interested in knowing which code is causing the tutorial's scalability to be sub-optimal.



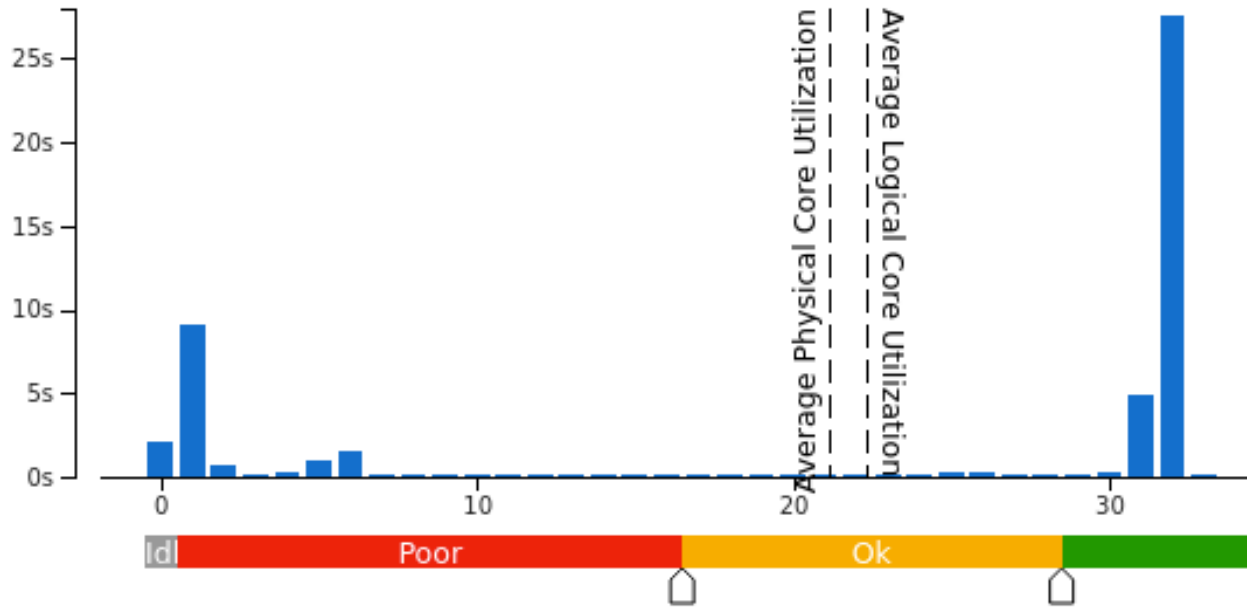
Marked in purple, we can see that the function that consumes the most time and that represents a bottleneck in execution and scalability is: **CheckFilters.**

3. HIGGS ANALYSIS EXECUTION

This tutorial consists in reconstructing the Higgs boson decaying to two Z bosons from events with four leptons. This is the second and last tutorial that we want to benchmark and analyse.

a. CORE USAGE

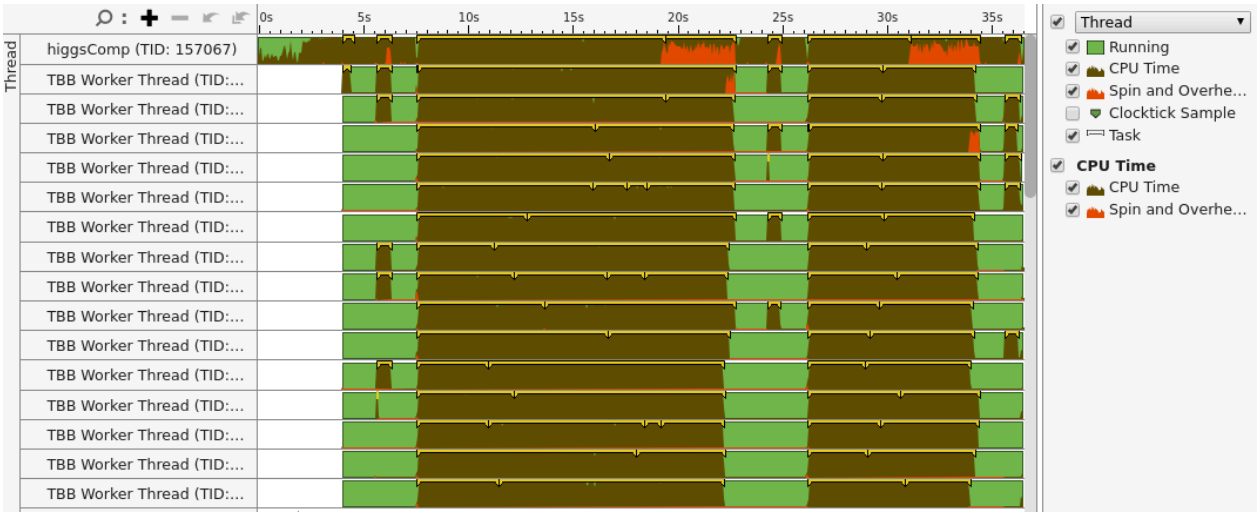
In the same way as in the previous tutorial, we are interested in knowing which is the activity that the computational resources of the server have. In this case, 32 physical cores.



According to the data shown by the graph, most of the time (approximately 25 seconds) is made use of all the physical cores of the server. However, unlike the previous tutorial, the time in which a single core assumes all the workload is much longer. This core works in a unitary way for approximately 10 seconds. Possibly performing initialization tasks, input/output, etc.

b. THREAD ACTIVITY

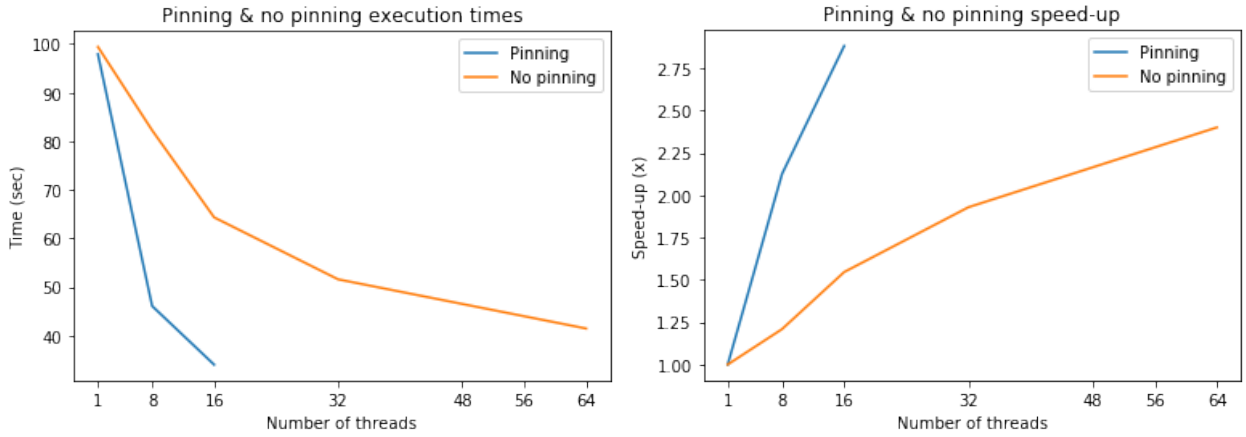
For the analysis of the activity of the threads in this execution of the Higgs tutorial 16 threads of execution have also been used.



However, unlike the previous tutorial, here we can see that there is much more time in which the threads are not carrying out computational tasks (green color) but that they are idle longer.

c. SCALABILITY (PINNING & NO PINNING)

As for the results of the scalability analysis for this tutorial to determine if the pinning (forcing the execution threads to remain in certain physical cores), these are the generated graphics:

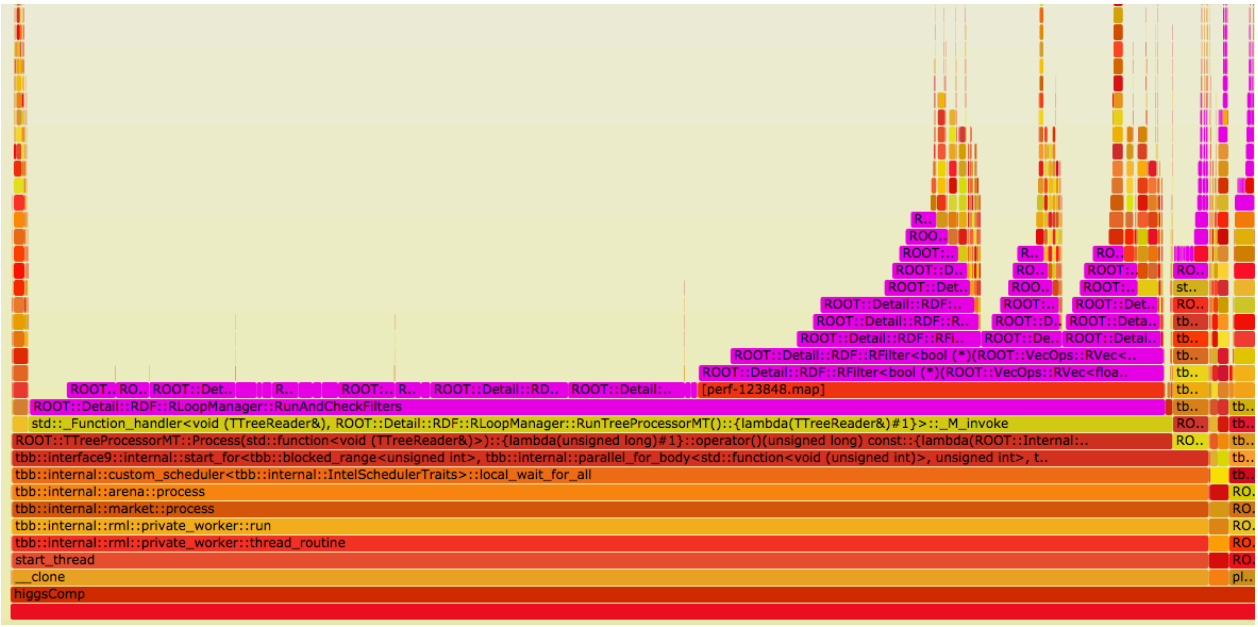


As we can see, the results are very similar both in runtime and speed-up to those of the previous tutorial.

The execution time for pinning is much shorter than that in which pinning is not used. The same happens with the speed-up, which is much higher for those execution with pinning.

d. FLAMEGRAPH

By using Brendan Gregg's tool to check the execution stack, you get the following data:



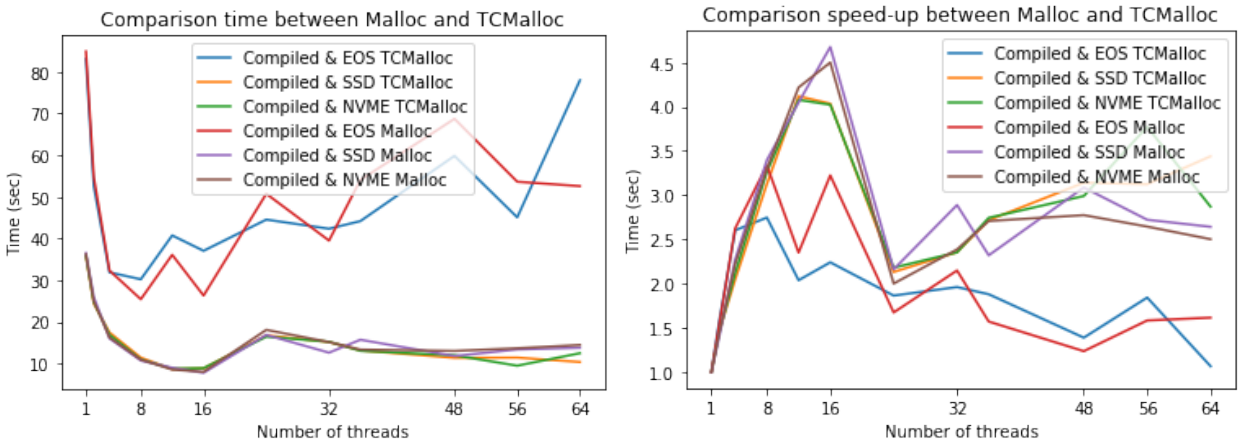
The result indicated in purple indicates that the checkFilters function of the RDataFrame class is the one which is carrying a large part of the execution load.

4. MALLOC VS TCMALLOC

Beyond the analyses of the two tutorials (Dimuon and Higgs), we were interested in knowing if the library in charge of memory allocation could be having a negative impact on the execution time of the tutorials as it is a multithreaded and multi-core execution.

Therefore, we wanted to compare the standard C++ allocation library (Malloc) with a multithreaded/multicore execution library (TCMalloc). To do this, we preload the TCMalloc library with the LD_PRELOAD command.

These two graphs show a direct comparison of time and speed-up between all executions with compiled code using both the Malloc library and the TCMalloc library.



In terms of execution time, we can see that the most efficient executions correspond to those that use SSD and NVME with both TCMalloc and Malloc without significant differences between them.

In the speed-up something similar happens, those executions that have a higher speed-up are those corresponding to SSD and NVME. And although there is a speed-up difference between both libraries, this difference is also practically irrelevant to the case at hand.

5. FUTURE WORK

The next steps to be considered on the basis of the analyses carried out are as follows:

In the first place it would be desirable to perform scalability and performance analysis not only with tutorials but with larger data analysis code.

In addition, it is necessary to assess and consider what could be an adequate refactoring of the RDataFrame class to allow good scalability.

6. CONCLUSION

We can draw several common conclusions from the executions of the Dimuon and Higgs tutorials.

First, the scalability of the code is far from adequate. In the best case, the speed-up achieved is 4.5x with 16 threads compared to execution with 1 single thread. And, from that point on, decreasing.

Using pinning has a positive impact on the reduction of the execution time and also on the speed-up by forcing the execution threads to remain restricted in certain physical cores of the processor.

The most efficient and least variable storage media are SSD and NVME.

Compiled code and JIT code have very similar execution times. However, the latter scales better, with a speed-up difference in some cases of almost 2x.

In principle, using one library or another of memory allocation (Malloc vs TCMalloc) does not provide any significant advantage in terms of efficiency and/or execution time.

After the analyses and the conclusions drawn from them, it is necessary to evaluate and consider what could be an adequate refactoring of the RDataFrame class to allow good scalability.