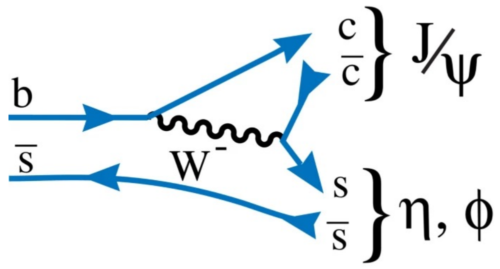
Performance study of HEP software tools

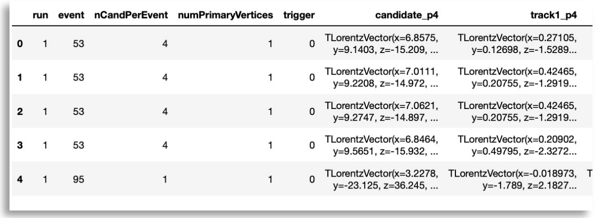
Adriano Di Florio (INFN & Politecnico Bari)
Alexis Pompili (INFN & Università Bari)
Dung Hoang (Rhodes College)

142nd PPP meeting - Performance study of HEP software tools
29th September 2022

The Goal

- **8 weeks** project with a student from INFN-DOE collaboration (Chris).
- The goal of the project was to write up a «framework» which could allow us to deploy the full chain from data handling, selection, skimming and up to fitting in a single **Jupyter notebook**.
- The simple benchmark we used was the reconstruction of **B^0_s meson into J/Psi and Phi mesons** with the CMS experiment.
- We have chosen to run it locally rather than relying on services such as SWAN:
    - opportunity reasons (Chris couldn't get access to it immediately given is not affiliated to CERN)
    - we wanted to benchmark a «realistic» setup in which the final user could run on its institution cluster (where available);
    - **to test the performances of Bari T2 new GPU-cluster**.
- The first block of the chain is accessing, filtering and skimming the ROOT files coming from CMSSW analyzers. This is the piece I will be discussing here.
- In the past we had been using **uproot** that, before the introduction of RDFs, allowed quick way of interfacing ROOT with the python-based data-analysis tools (pandas, matplotlib). Thus we have tried to compare the two.
- Everything is based on CERN Run2 **OpenData** and the ROOT files in output are "flat" tuples composed of either TLorentzVectors or scalars.





	run	event	nCandPerEvent	numPrimaryVertices	trigger	candidate_p4	track1_p4
0	1	53	4	1	0	TLorentzVector(x=6.8575, y=9.1403, z=-15.209, ...	TLorentzVector(x=0.27105, y=0.12698, z=-1.5289...
1	1	53	4	1	0	TLorentzVector(x=7.0111, y=9.2208, z=-14.972, ...	TLorentzVector(x=0.42465, y=0.20755, z=-1.2919...
2	1	53	4	1	0	TLorentzVector(x=7.0621, y=9.2747, z=-14.897, ...	TLorentzVector(x=0.42465, y=0.20755, z=-1.2919...
3	1	53	4	1	0	TLorentzVector(x=6.8464, y=9.5651, z=-15.932, ...	TLorentzVector(x=0.20902, y=0.49795, z=-2.3272...
4	1	95	1	1	0	TLorentzVector(x=3.2278, y=-23.125, z=36.245, ...	TLorentzVector(x=-0.018973, y=-1.789, z=2.1827...

Why here?

We had found some strange behaviour (RDF much slower than uproot) and so we asked a slot here to give our feedback and get yours to understand if this was expected.

In the meanwhile we understood what was happening but this was still a good occasion to exchange feedbacks and present here as “user experience”.

The Setup

- The measurements were done on the following machines at the ReCaS-Bari T2.
  - **tesla04**: 32 CPUs, 251GB RAM, Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
  - **wn-gpu-8-3-22**: 256 CPUs, 2003GB RAM, AMD EPYC 7742 64-Core Processor
  - **wn-1-8-9**: 64 CPUs, 251GB RAM, AMD EPYC 7281 16-Core Processor
- For **tesla04,** data could be accessed either locally (via optical disks, no SSD) or remotely (via InfiniBand). For other machines, data could only be accessed remotely.
- There were 126.78GB of data in total (45M events).
- The original dataset was split into 128 files, each of which contains the same number of events.
- The following operations are included in the runtime measurements:
  - opening the TTrees inside root files;
  - applying specific filters on the TTree to expose the invariant mass distribution of the B^0_s **meson**;
  - converting the mass column to a **numpy** array (to be sent to the next step).
- At the time of the measurements, there were no other tasks on the machine that may interfere with the above operations.
- Software setup (here it comes the unintentional bias):
  - ROOT 6.26 for tesla04
  - ROOT 6.24 for the other two nodes (we were having some issues with glibc at the beginning)

Parallel processing with Uproot

- **Uproot** doesn’t have a built-in option for implicit parallel processing.
- **But** it allows user to **specify the number of events to be processed** in each TTree.
- To enable parallelism, we set up a routine through **multiprocessing** module. Each subprocess will handle a slice of data.
- So it’s always possible to **split the work equally** between the subprocesses.
- In other words, each subprocess will handle approximately the same number of events. This means all the subprocesses will take similar amounts of time to finish.

Parallel processing with RDataFrame

- RDF has a built-in option for parallel processing: **EnableImplicitMT()**.

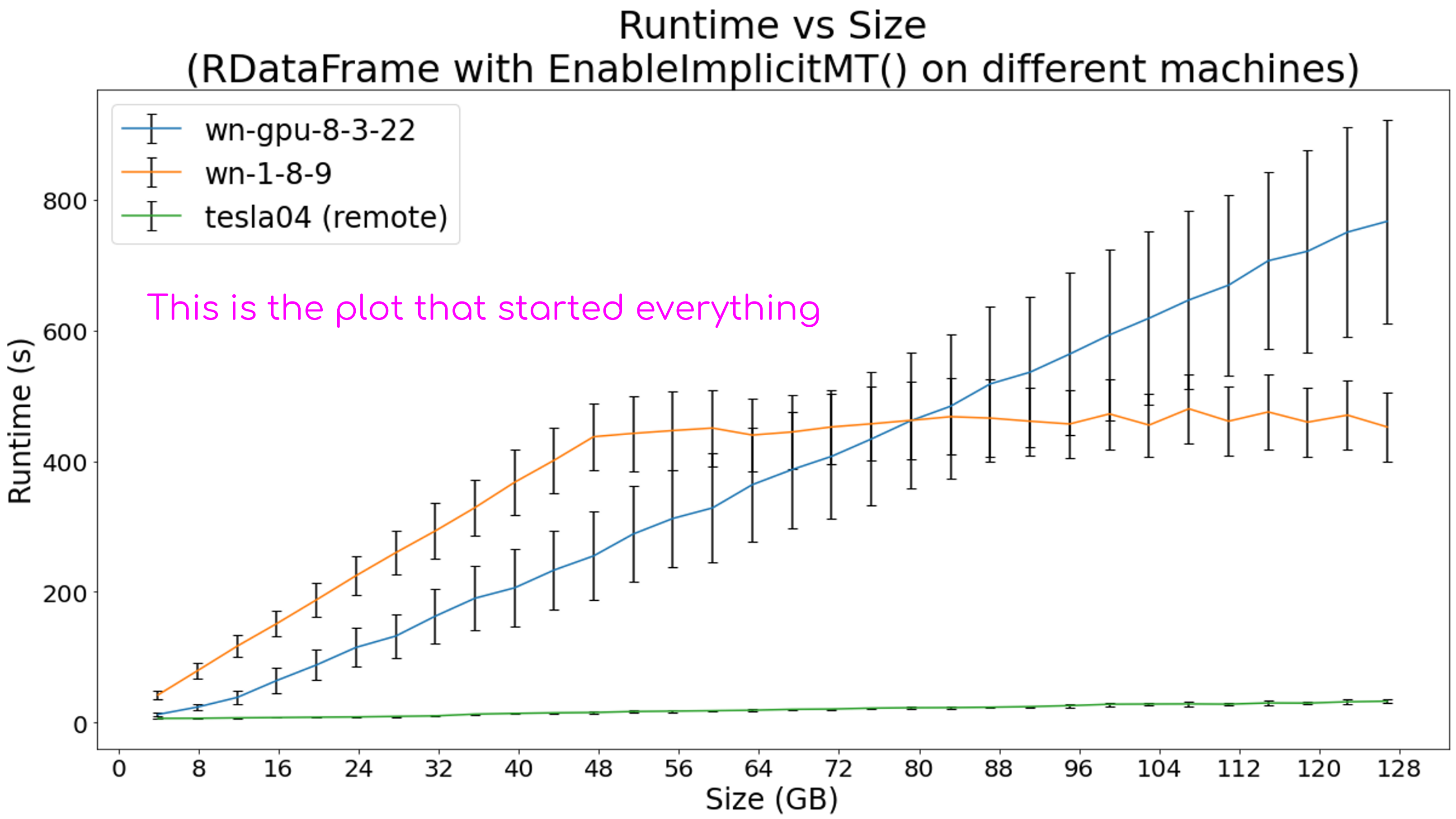
- RDF doesn’t provide (does it?) the option to specify the number of events to be processed in each file. Thus, with this approach, RDataFrame could only parallelize over ROOT files. In other words, each subprocess will work on approximately the same number of files rather than the same number of events.

- *As a result, the workload is not always divided equally.* For example, if we have 5 files but only 4 subprocesses, one subprocess will have to deal with 2 files. Even when every subprocess handle exactly the same number of files, we may still face uneven workload distribution if the **files have different sizes**. Since we have to wait for the slowest subprocess to finish, this may create a bottleneck.

- To mitigate this issues, we split the original dataset into 128 files (to match the #of CPUs we have), each containing the same number of events.

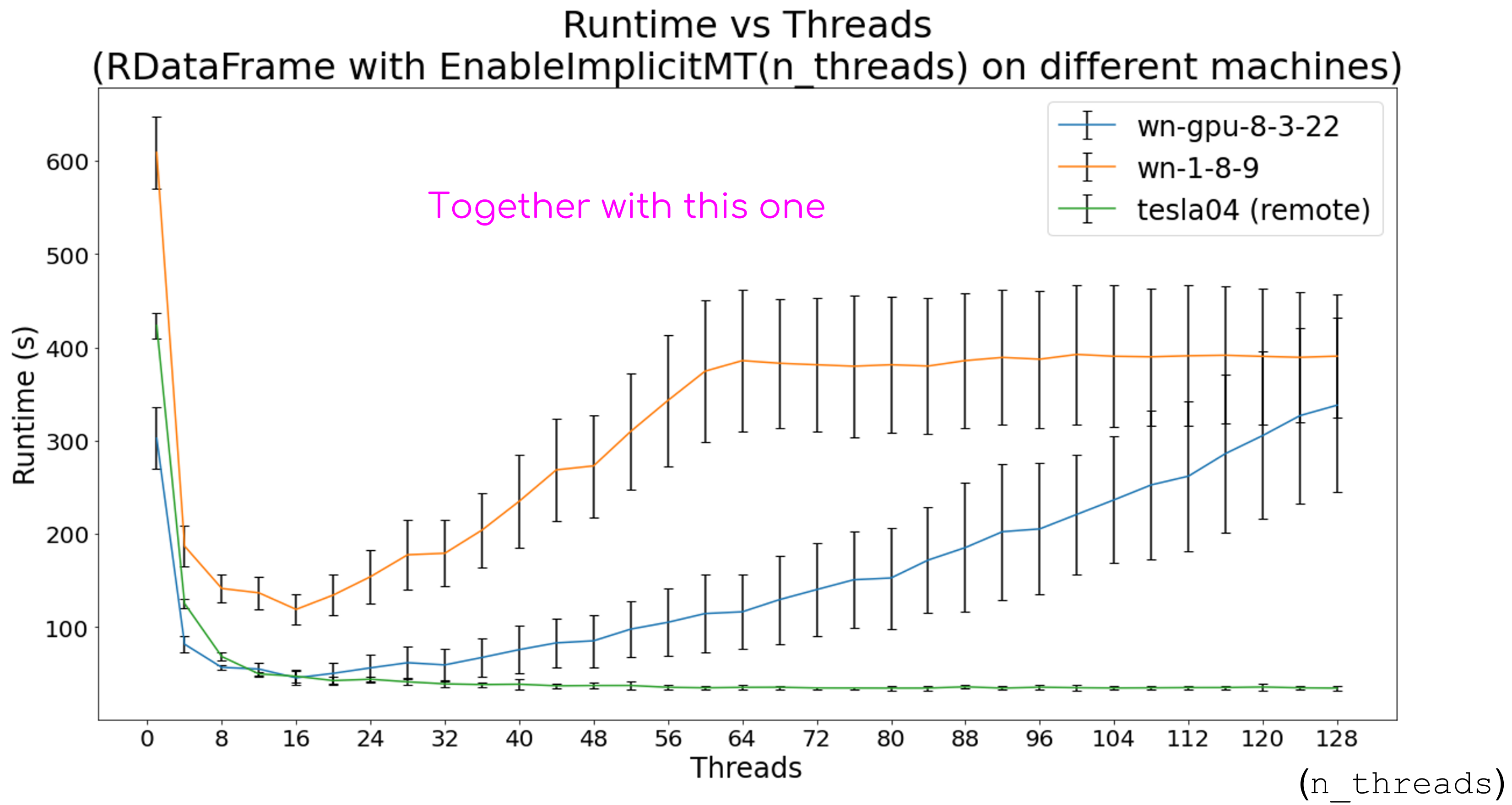
- For a fair comparison, we used these files as input for both Uproot and RDataFrame performance studies.

RDF With EnableImplicitMT()



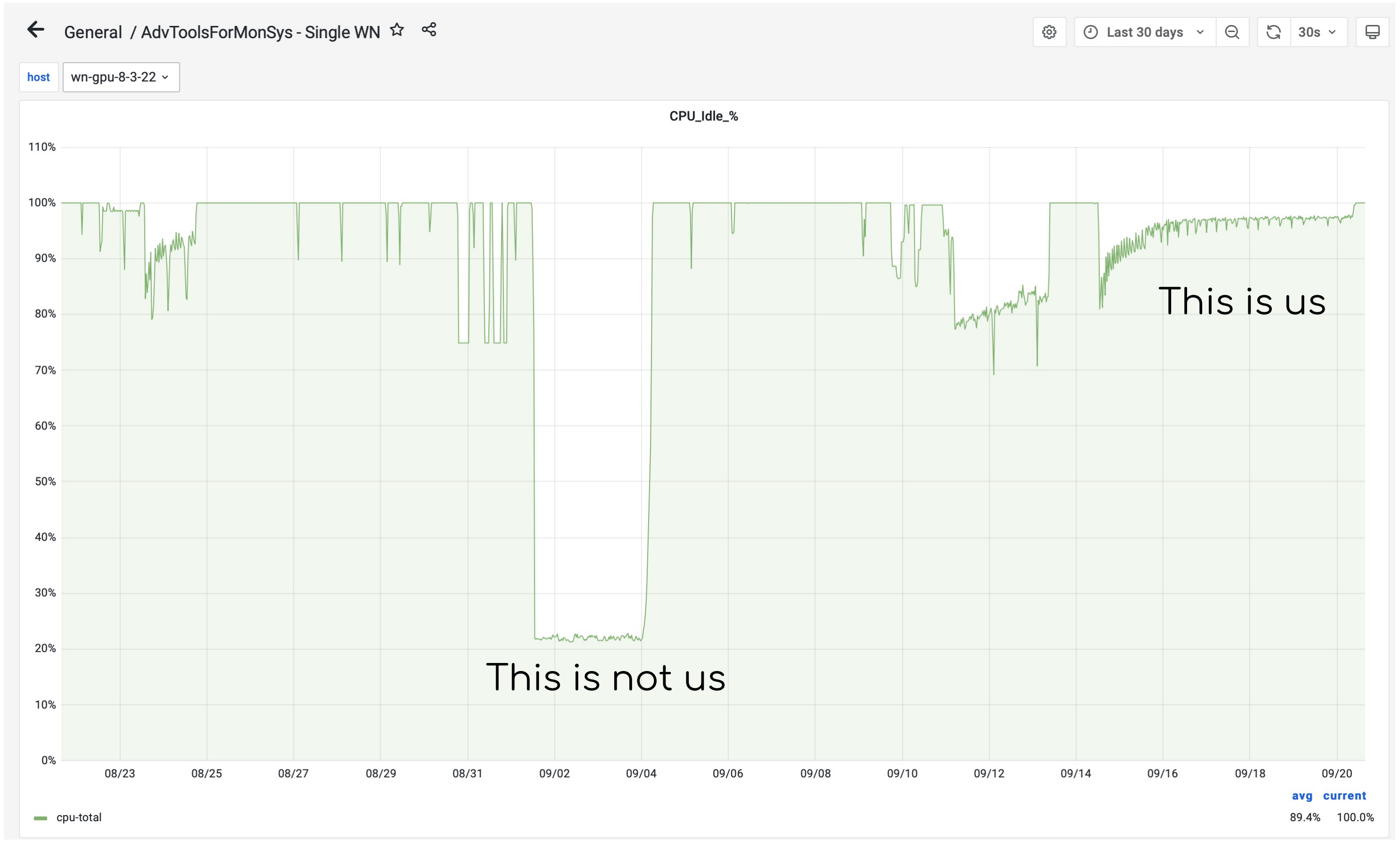
This is the plot that started everything

Runtime vs Threads



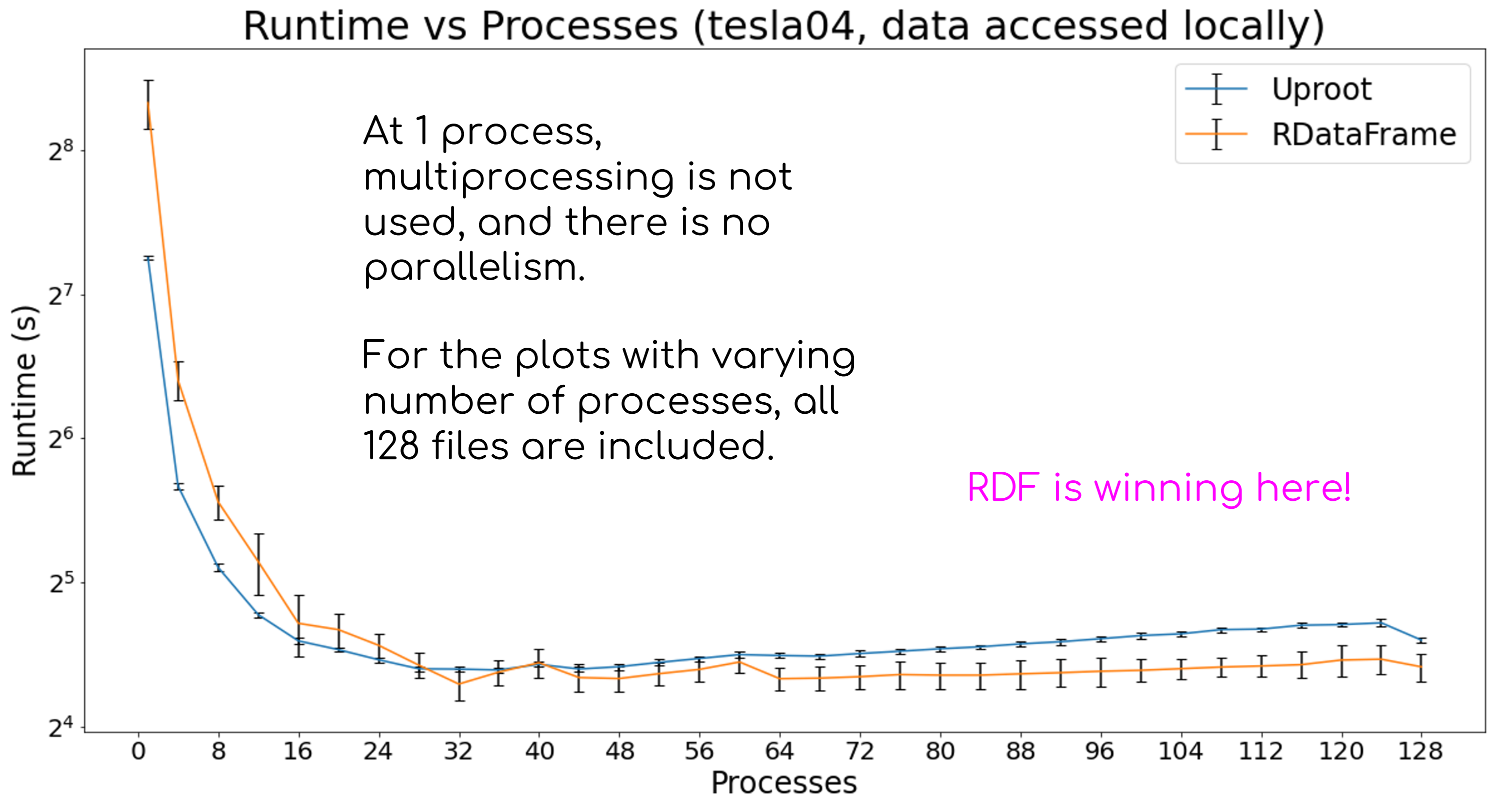
Specifying the number of threads didn't help with the inconsistency. In every test the full data sample is processed.

Server monitoring



	avg	current
cpu-total	89.4%	100.0%

Runtime vs Processes

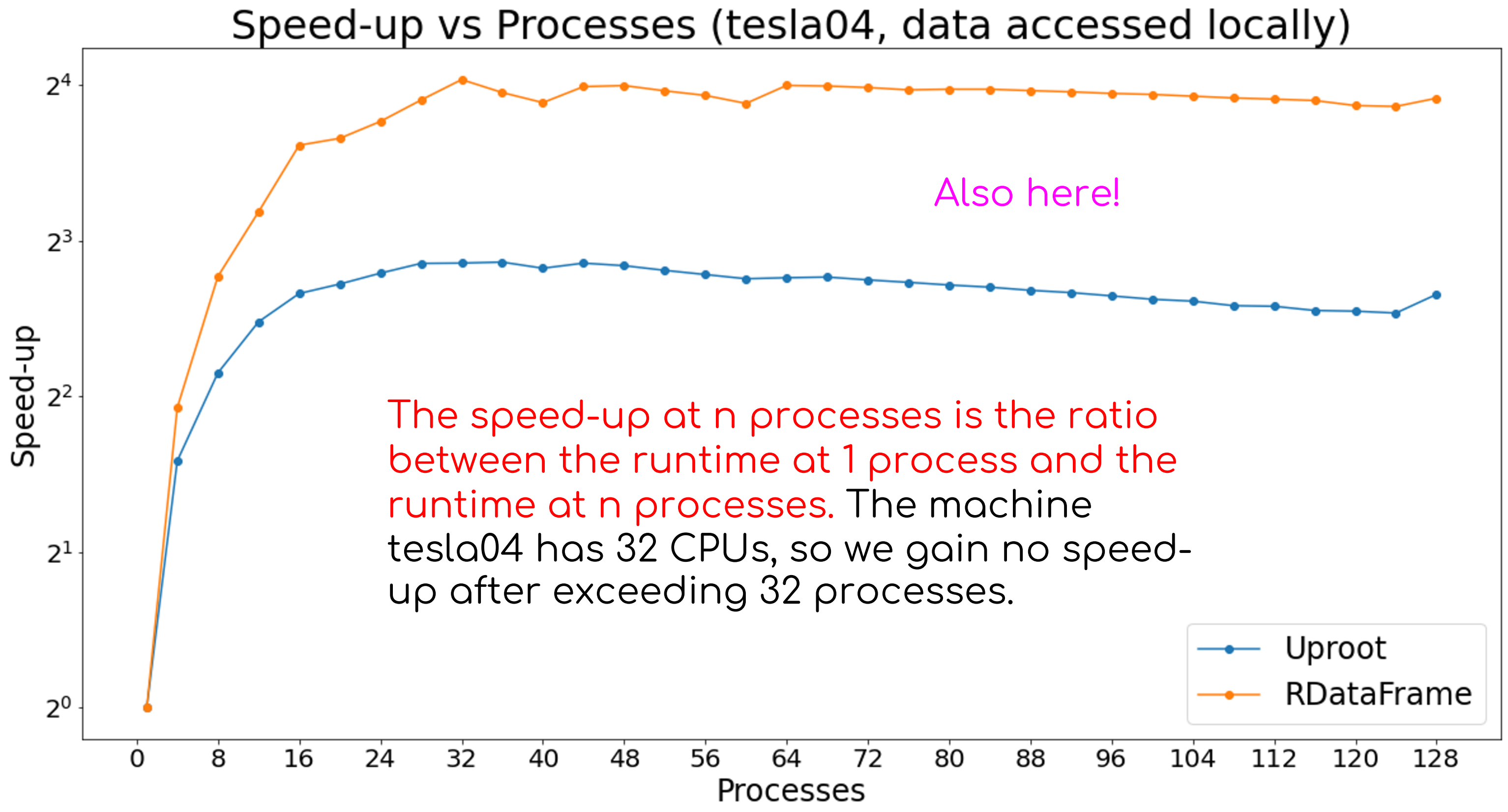


At 1 process, multiprocessing is not used, and there is no parallelism.

For the plots with varying number of processes, all 128 files are included.

RDF is winning here!

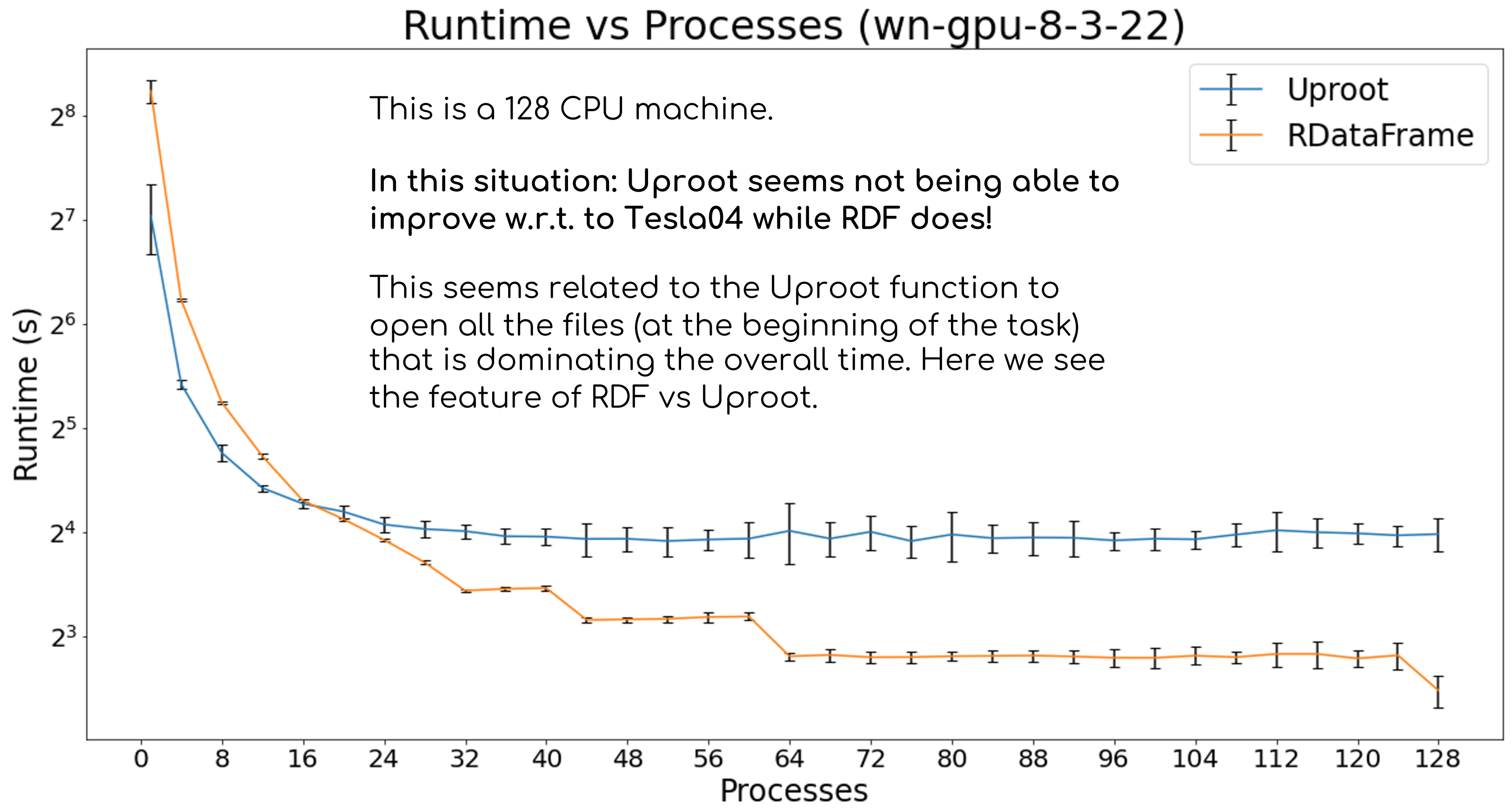
Speedup vs Processes



Also here!

The speed-up at n processes is the ratio between the runtime at 1 process and the runtime at n processes. The machine tesla04 has 32 CPUs, so we gain no speed-up after exceeding 32 processes.

Runtime vs Processes (ROOT 6.24)

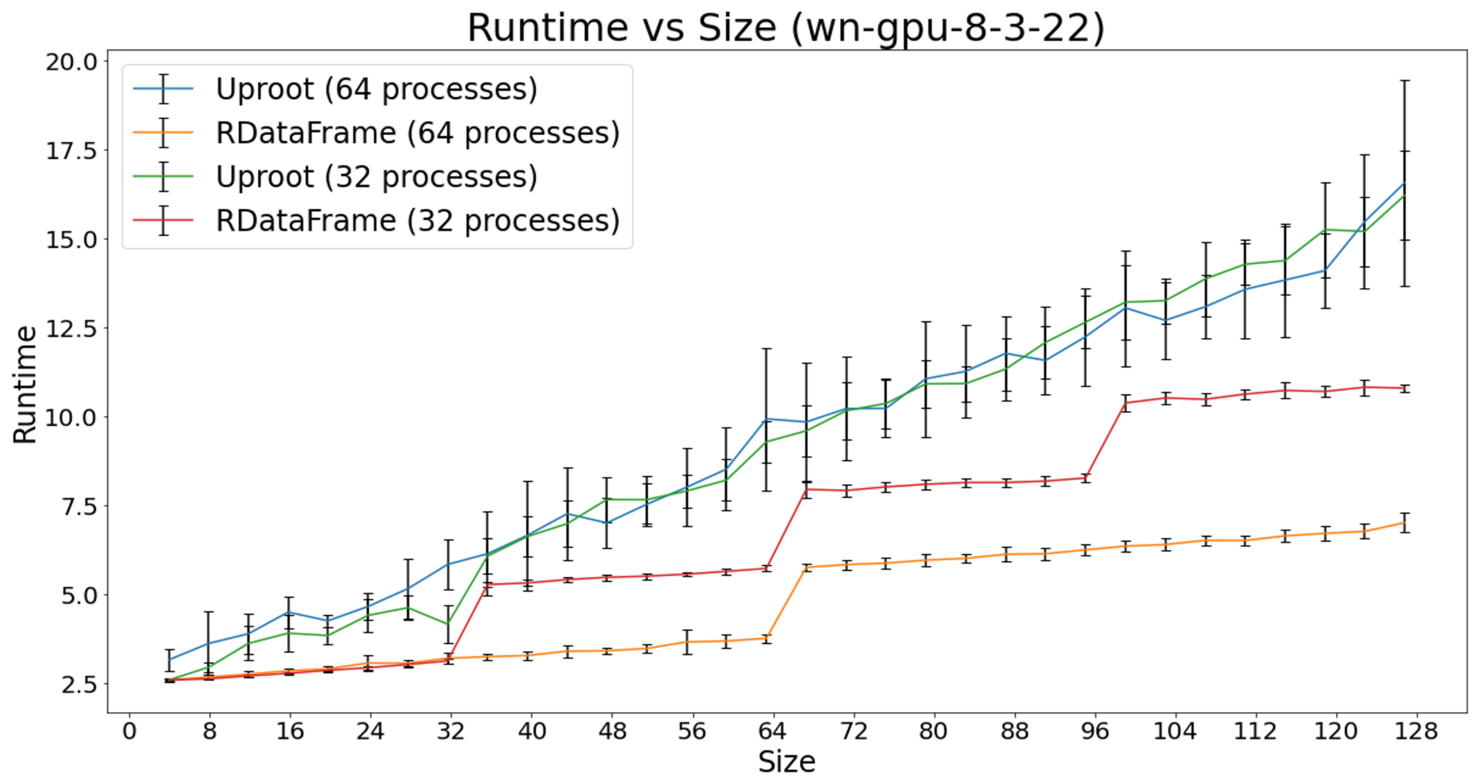


This is a 128 CPU machine.

In this situation: Uproot seems not being able to improve w.r.t. to Tesla04 while RDF does!

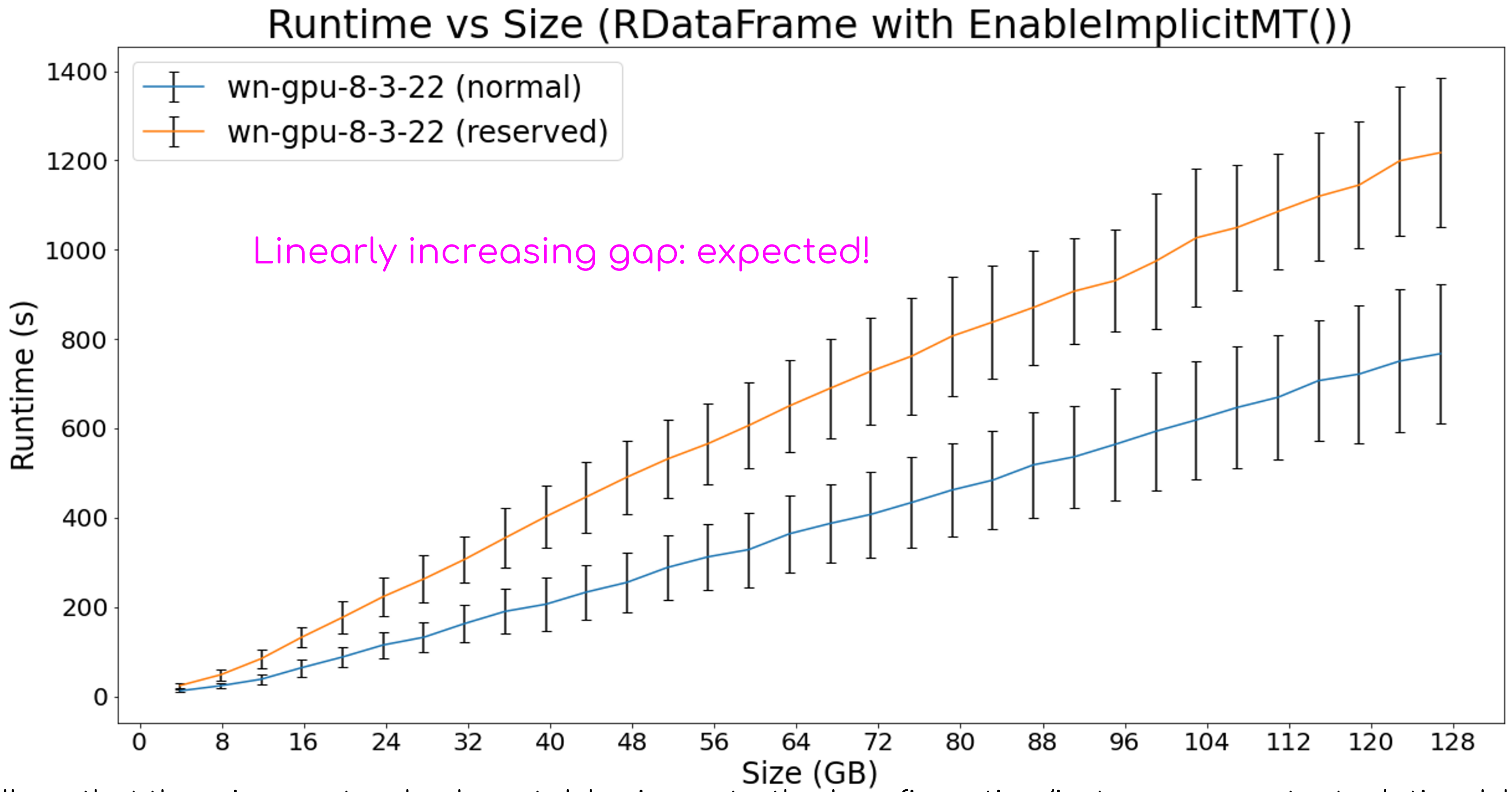
This seems related to the Uproot function to open all the files (at the beginning of the task) that is dominating the overall time. Here we see the feature of RDF vs Uproot.

Runtime vs SIzes (multiple machines)



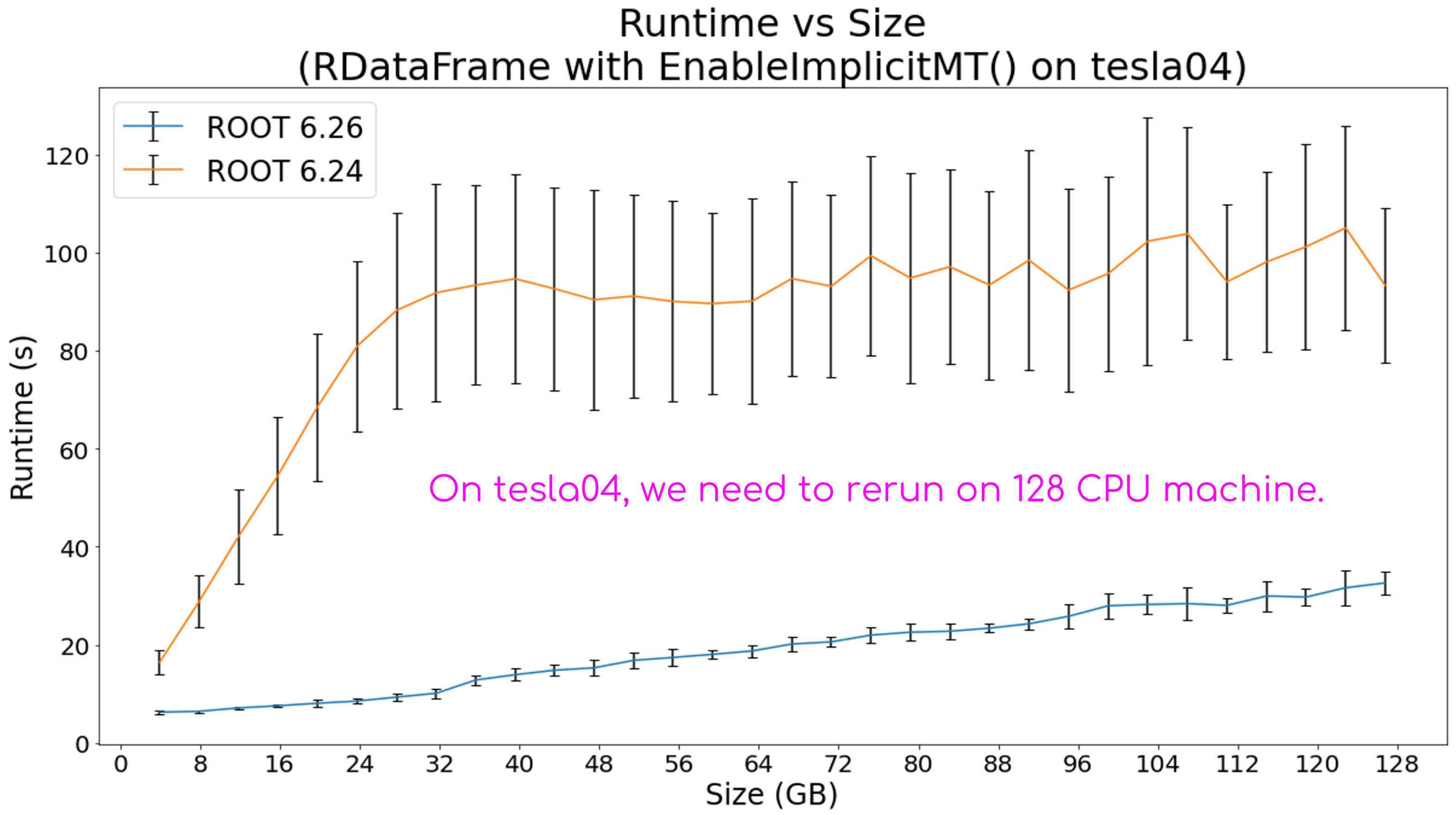
The jumps are expected and reflect the fact that RDataFrame can only parallelize over files? With 32 subprocesses, for example, it took similar amounts of time to process 36 and 64 files. This is because in both cases, the subprocess with the heaviest workload has to handle 2 files (⌈36 / 32⌉ = ⌈64 / 32⌉ = 2).

Comparing locally with remotely



This tells us that there is no network relevant delay in our testbed configuration (just some ~ constant relative delay of ~20% amount)

The smoking gun: 6.24vs6.26



On tesla04, we need to rerun on 128 CPU machine.

Bottom lines

- The bottom line here is that going from 6.24 to 6.26 things improve dramatically.
- At the end of the day RDF performs much better than uproot!
- We have chosen it to go in our workflow.

Thanks!