Optimized workflow for analyses with multiple RDataFrames

Enrico Guiraud, Stefan Wunsch

ROOT

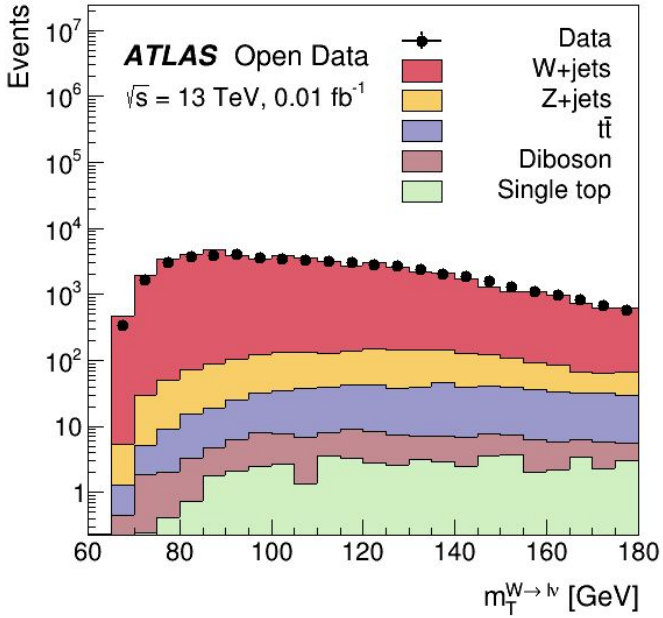
Data Analysis Framework

https://root.cern

The example analysis

- **All (realistic) analyses have to use multiple RDFs**
- At least one RDF per process
- Typically each process is described with multiple datasets (W+1jet, W+2jet, …)

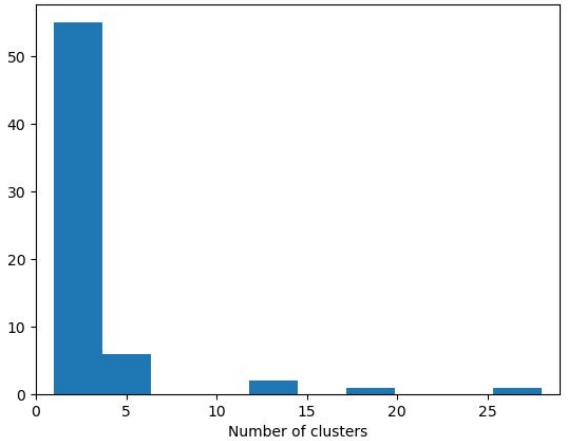
- **This example:**
- 4.8 GB in total
  - Compressed size
  - About 10% of the total dataset size
  - 2.3 GB read (measured with TTreePerfStats)
- 65 files (4 data + 61 simulation)
- 25 M events in total with 81 columns each
- Data + 5 simulated processes
- Setup requires 65 RDFs because each dataset requires different event weights



https://root.cern/doc/master/df105__WBosonAnalysis_8py.html

The core of the analysis

```
ROOT.EnableImplicitMT(12)

histos = {} # histograms
df = {...} # 65 RDFs
for s in samples:
    histos[s] = df[s].Filter(...)\
                     .Define(...)\
                     .Histo1D(...)

# Trigger the event loops
for s in samples: h[s].Draw()
```



- Run very similar computation graphs for all dataset, but no TChain is possible
  (differences between data/MC, different weights per MC sample, ...)

- Event loops are multi threaded (on very few TTree clusters), but RDFs run sequential
  → **Results in an inefficient CPU usage!**

How could we improve the workflow?

```
ROOT.EnableImplicitMT(12)

histos = {} # histograms
df = {...} # 65 RDFs
for s in samples:
    histos[s] = df[s].Filter(...)\
                     .Define(...)\
                     .Histo1D(...)

# Trigger the event loops
for s in samples: h[s].Draw()
```

- Using TChains?
  Often not possible due to small differences in the computation graphs
  →Tree or file based Define and Filter nodes?
- Setup of the computation graphs is declarative, in principle as we want the user to use RDF
- Can we parallelize the sequential execution of multiple RDFs?

Run RDFs in parallel!

```
void RLoopManager::Run()
 {
-   ThrowIfPoolSizeChanged(GetNSlots());
+   {
+      R__LOCKGUARD(gInterpreterMutex);
+      ThrowIfPoolSizeChanged(GetNSlots());

-   Jit();
+      Jit();

-   InitNodes();
+      InitNodes();
+   }

    switch (fLoopType) {
    case ELoopType::kNoFilesMT: RunEmptySourceMT(); break;
    ...
    case ELoopType::kDataSource: RunDataSource(); break;
    }

-   CleanUpNodes();
+   {
+      R__LOCKGUARD(gInterpreterMutex);
+
+      CleanUpNodes();

-   fNRuns++;
+      fNRuns++;
+   }
 }
```

Hacked in thread safety for a proof of concept to enable running multiple RDataFrames in parallel!

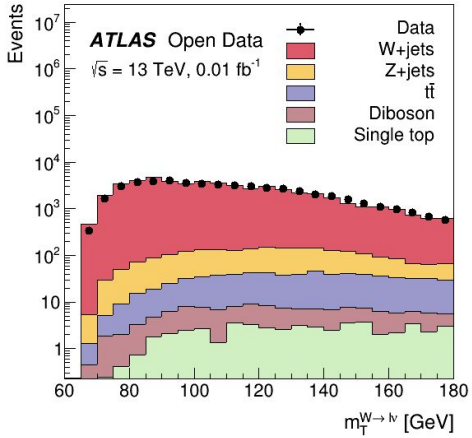
Trigger the event loops, in parallel!

```
ROOT.EnableImplicitMT(12)

histos = {} # histograms
df = {...} # 65 RDFs
for s in samples:
    histos[s] = df[s].Filter(...)\
                     .Define(...)\
                     .Histo1D(...)

# Trigger the event loops, all in parallel!
ROOT.gInterpreter.ProcessLine('std::vector<ROOT::RDF::RResultPtr<TH1D>> ptrs;')
for s in samples: ROOT.ptrs.push_back(histos[s])

ROOT.gInterpreter.ProcessLine('''
ROOT::TThreadExecutor pool;
pool.Map([](ROOT::RDF::RResultPtr<TH1D> ptr) { *ptr; }, ptrs);
''')
```

All RDFs share the same thread pool!

Is it worth it?

- Multi threading on 12 (physical?) cores (running on almost empty machine with 24/48 physical/logical cores)
- Read data from storage server with SSDs and 20 GBit connection

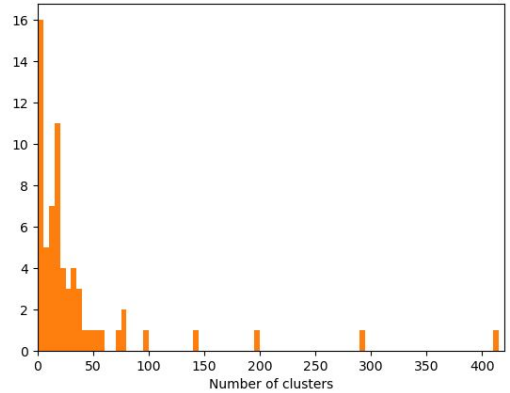


- **Sequential execution of RDFs**
  - **90s wall time**
  - 300s CPU time
  - 330% CPU usage
  - 1.3 GB max resident size

- **Parallel execution of RDFs**
  - **36s wall time** → 90/36 = **2.5x faster runtime!**
  - 310s CPU time → slightly increased, due to higher contention?
  - 870% CPU usage → 870/330 = **2.5x higher CPU usage!**
  - 2 GB max resident size → concurrent event loops?

Does it matter with big data?

- Let's try again with the full dataset of 66 GB (see plot on the bottom for # of clusters)
- In realistic settings, files with <12 clusters are still common!
- Issue amplifies when using more cores (machines with >48 cores are already typical for larger university groups)
- **Scaling up to 100 cores in a realistic analysis?**
  In practise, easily blocked by this!



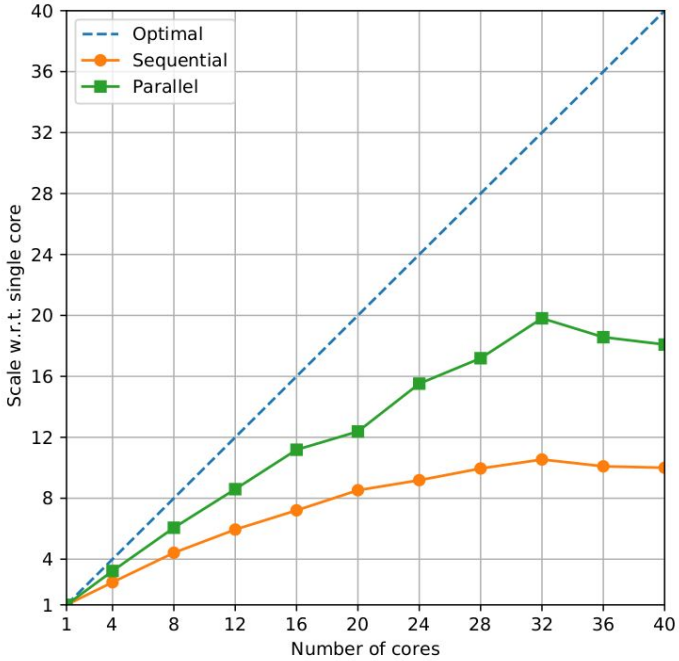
- **Sequential execution of RDFs (MT=12)**
  - **470s wall time**
  - 4500s CPU time
  - 970% CPU usage
  - 2.1 GB max resident size
- **Parallel execution of RDFs (MT=12)**
  - **380s wall time→1.2x faster!**
  - 4300s CPU time
  - 1140% CPU usage→**1.2x higher!**
  - 3.2 GB max resident size
- **Sequential execution of RDFs (MT=24)**
  - **203s wall time**
  - 3200s CPU time
  - 1610% CPU usage
  - 3.6 GB max resident size
- **Parallel execution of RDFs (MT=24)**
  - **127s wall time→1.6x faster!**
  - 2800s CPU time
  - 2210% CPU usage→**1.4x higher!**
  - 6.1 GB max resident size

Scaling is back!

- Full dataset with 66 GB
  - Compressed size
  - 23.8 GB read (measured with TTreePerfStats)

- Machine with 64/128 physical/logical cores
- Data read from storage server with SSDs and 20 GBit connection
  - iperf: 5.2 GBit/s to the storage server
  - rsync: ~**220MB/s** (copy to local SSD)

- Limited for >32 cores again by number of clusters?
- Limited by network/IO?
  - With 32 cores:
    23.8 GB / 105 s = **230 MB/s** = 1.8 Gbit/s
  - Beyond capabilities of HDDs
  - Beyond capabilities of most network connections
  - Saturates the connection to the storage server?

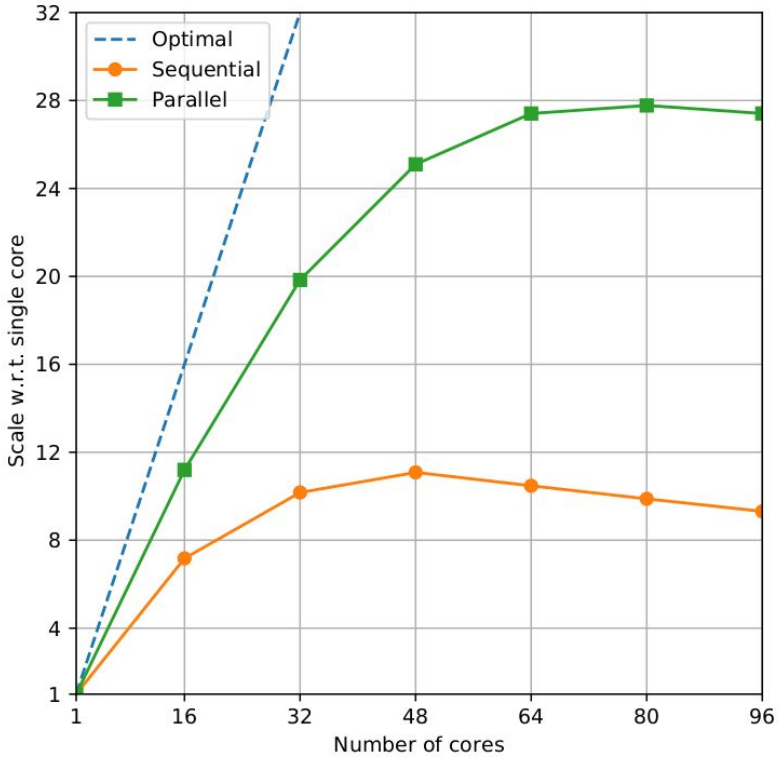


Scaling is back, reloaded!

- cgroups limited my share to 32 cores!
- Moved the dataset to a local SSD with NVMe interface

- With 64 cores:
  23.8 GB / 75 s = **325 MB/s**

- Scaling saturates at
  - 28x with parallel RDFs
  - 11x with sequential RDFs



Proposal: Reference benchmark suite

- Set up a benchmark suite which scaling we know in optimal conditions
- Most important: Simple to run and to reproduce
- Allows to “calibrate” results on new systems and can identify bottlenecks, which are not related to our software
- Make it part of rootbench?

Final thoughts

- **Multiprocessing + RDF**
  - Works!
  - But suffers heavily with imbalanced datasets because you have to wait for the slowest process (see Massimilliano's and Vincenzo's studies!)

- **Multithreading + RDF**
  - Allows to use all your resources efficiently
  - Speeds up complex analyses and restores scaling using many cores
  - Important for efficient usage of N cores batch jobs (without manual file splitting to tackle the imbalance)
  - Can we make it work with Python's `ThreadPoolExecutor`?

- **Alternative solution: Tree/file based Define and Filter transformations**
  - Solves the same issue with a different programming model
  - Less problems with implicit dependencies between RDFs
  - Goes well with the implicit parallelism model in ROOT

  - You can think of other solutions?
  - What is the strategy for PyRDF, naturally using many cores/nodes?