EPJ Web of Conferences **214**, 05033 (2019) https://doi.org/10.1051/epjconf/201921405033
CHEP 2018

A Parallelised ROOT for Future HEP Data Processing

Danilo Piparo[1,*], *Philippe* Canal[2], *Guilherme* Amadio[1], *Enrico* Guiraud[1,3], *Axel* Naumann[1], *Xavier* Valls[1,4], and *Enric* Tejedor[1]

[1]CERN, Switzerland
[2]FNAL, US
[3]Oldenburg University, Germany
[4]Jaume I University, Spain

Abstract. In the coming years, HEP data processing will need to exploit parallelism on present and future hardware resources to sustain the bandwidth requirements. As one of the cornerstones of the HEP software ecosystem, ROOT embraced an ambitious parallelisation plan which delivered compelling results. In this contribution the strategy is characterised as well as its evolution in the medium term. The units of the ROOT framework are discussed where task and data parallelism have been introduced, with runtime and scaling measurements. We will give an overview of concurrent operations in ROOT, for instance in the areas of I/O (reading and writing of data), fitting / minimization, and data analysis. This paper introduces the programming model and use cases for explicit and implicit parallelism, where the former is explicit in user code and the latter is implicitly managed by ROOT internally.

1 The Opportunities Offered by the High Luminosity LHC Challenges

The processing of the data in the High Luminosity LHC [1] era poses unprecedented challenges to computation in High Energy Physics (HEP). The data produced by LHC [2] experiments will increase by one order of magnitude. The dataset will also be significantly more complex than the one of today due to the dramatic rise of number of simultaneous collisions per bunch crossing. Several strategies to fit the expected budget envelope while honouring the foreseen ambitious Physics programme are being discussed. Given present and realistically foreseeable computer hardware, what is certain is that the exploitation of parallelism in HEP code, from simulation to end user analysis is a prerequisite for our success.

2 ROOT's Approach to Parallelism

In the aforementioned scenario, as a foundation library present in virtually every HEP experiment software stack, ROOT [3] is already evolving in order to become parallel and support expression of parallelism. Such an endeavour presents at least two main hurdles. Components that cannot be evolved to support a parallelised environment or that cannot be parallelised in a performant way need to be replaced by new implementations. In addition, a programming

*e-mail: danilo.piparo@cern.ch

model which makes scientists productive quickly is necessary. Ergonomic interfaces and ease of use are of paramount importance given that it's impossible to require a too broad palette of technical skills to physicists, especially the ones in the very early stages of their career (see example in Figure 1).

```
TFile f(filename);
TTreeReader tr(treename, &f);
TTreeReaderArray<double> px(tr, "px");
TTreeReaderArray<double> py(tr, "py");
TTreeReaderArray<double> E(tr, "E");

TH1F h("pt", "pt", 16, 0, 4);

while (tr.Next()) {
   for (auto i=0U;i < px.GetSize(); ++i) {
      if (E[i] > 100) h.Fill(sqrt(px[i]*px[i] + py[i]*py[i]));
   }
}
h.Draw();
```

(a) Traditional TTreeReader based approach.

```
ROOT::EnableImplicitMT();
RDataFrame f(treename, filename);
f.Define("good_pt", "sqrt(px*px + py*py)[E>100]")
 .Histo1D({"pt", "pt", 16, -.5, 3.5}, "good_pt")->Draw();
```

(b) New RDataFrame based approach.

Figure 1: An example of ergonomy of the new ROOT interfaces. The traditional imperative (a) and declarative (b) approaches compared. The result produced by the two snippets is identical. The conciseness and clarity of (b) can be appreciated. In addition, the code in (b) runs in parallel on all available cores.

For the aforementioned reasons, the concept of implicit parallelism has been introduced in ROOT, i.e. the ability of the library to execute in a parallelised fashion some expensive operations, requiring no intervention from users. The implicit parallelism can be activated with a call to the static routine `ROOT::EnableImplicitMT(nWorkers)`. To implement implicit parallelism, ROOT adopts a task based paradigm, supported by the TBB library [4]. That invocation schedules the creation of a pool of worker threads which is initialised upon the submission of the first task. Being the TBB thread pool global to the process, the usage of ROOT can be combinded with the one of other libraries exploiting TBB for the expression of in-process task based parallelism without incurring in the risk of overcommitting the worker node. Successful examples of this interoperability are the CMSSW [5] and Gaudi [6] frameworks, both relying on the TBB library.

To satisfy the need of expert users to express parallelism explicitly, ROOT provides a set of low level utilities. For example the `TThreadExecutor` and `TProcessExecutor` implement the Map and Map-Reduce [7] patterns respectively exploiting thread and process based parallelism. Protection of resources can be guaranteed by highly efficient mutexes, such as the `TRWLock`, and asynchronicity exploiting the TBB pool can be expressed with `TAsync`.

2.1 Components Currently Parallelised and example Benefits

Several operations carried out with ROOT in analysis and central data procssing are already parallelised. Table 1 summarises such operations and the benefit for users.

Table 1: Operations parallelised in ROOT as of release 6.14.

Operation	Short Description
`RDataFrame` event loop	Processes multiple entries of the input columnar dataset in parallel.
`TTree::GetEntry`	Reads, decompresses and deserialises multiple columns of a ROOT's TTree data structure.
`TTree::FlushBaskets`	Writes multiple portions of a dataset's column on disk.
`TTreeCacheUnzip`	Decompresses in parallel big chunks of columnar datasets.
`TH{1,2,3}::Fit`	Performs in parallel the evaluation of the objective function over the data during a fitting procedure.
`TMVA::DNN`	Trains Deep Neural Networks in parallel.

In addition, a class which allows the parallel writing of the same `TTree` has been introduced, `TBufferMerger`. Its usefulness has been demonstrated by the CMS experiment, for which some central production workflows, could be significantly optimised, for example bringing the rate of processed events per second from 2.94 to 4.59, where the non realistic asymptotic maximum rate achievable excluding writing from the workflow was 5.41 [8].

The newly introduced parallelism brings up tangible benefits for analysis. Even on extreme architectures such as Intel KNL, good runtime and satisfactory scaling is achieved (see Figure 2).

3 Ongoing R&D Work

Section 2.1 describes the components which are parallelised in the released ROOT version 6.14 [9]. A substantial R&D effort is presently taking place in order to take advantage of distributed systems and heterogeneous platforms for ROOT based HEP data analysis and processing, which is described in the following sections:

3.1 Distributed Declarative Analysis

The computing power provided by an individual server, even when exploiting all of its CPU cores, might not be enough to carry out intensive computations involving the access of big datasets. A solution involving the usage of a distributed system is therefore necessary.

This is the reason why the `PyRDF` R&D has been started [10] [11]. The objective of the effort is to expose to the user a declarative interface as similar as possible to the one of `RDataFrame`: no code changes are required to the scientists for moving an analysis running on local resources to a distributed system (see Figure 3). Once the workflow is generated, the tool allows to either run it locally on one or multiple CPU cores or on a distributed backend. At the time of writing, only a Apache Spark [12] based backend has been completed and successfully tested. Other backends are foreseen, for example relying on Dask [13].

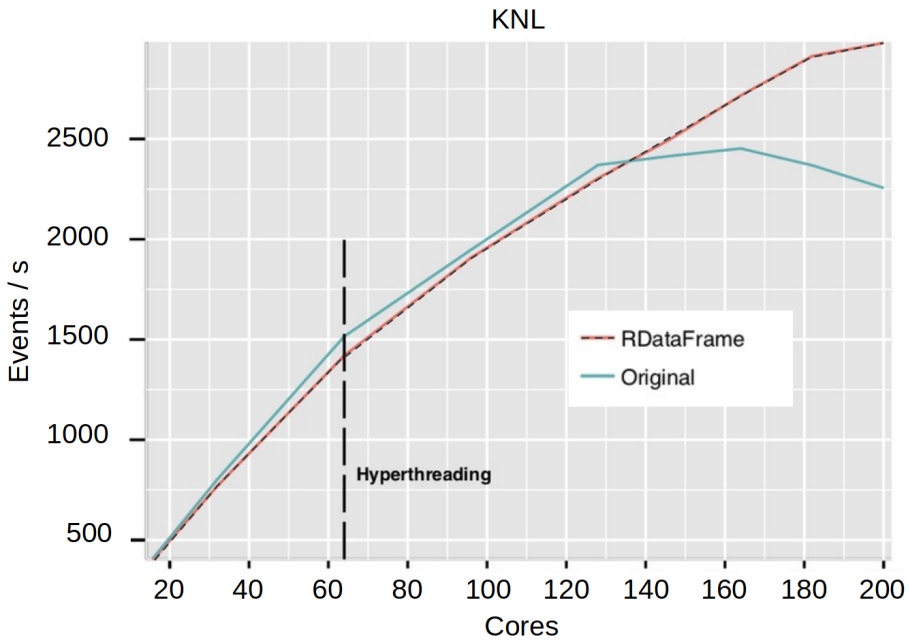

Figure 2: A Monte Carlo generation of QCD events featuring low transverse momentum is considered. No I/O from disk is performed but a large set of histograms filled. The plot shows the event rate scaling with respect to the number of cores used on a KNL device. The blue line shows the scaling of the original version of the application, based on a customised version of ROOT 5.34. The red line is relative to a porting of the application to ROOT 6.14. The scaling continues up to 200 cores, taking advantage of the accelerator while the application based on the old ROOT version features a degradation of the performance.

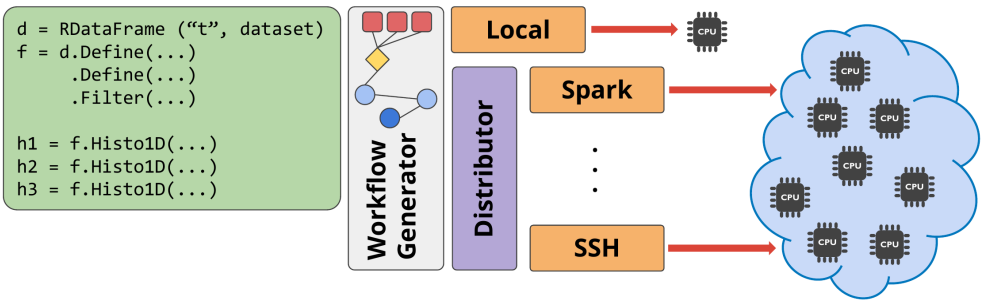

Figure 3: The PyRDF architecture.

3.2 Support of Heterogeneous Platforms

Heterogeneous platforms are very likely to be part of the future HEP computing infrastructure, even more than today. Therefore, it is necessary for ROOT to be able to interoperate with widely adopted runtimes and platforms enabling heterogeneous computing. This lead us to investigate the integration of NVidia CUDA [14] into Cling, ROOT's LLVM based code interpreter. At the time of writing, it is possible to interpret CUDA code with Cling, therewith

being able to access NVidia accelerators on the system directly from it. The implementation generates at runtime device-code through a second compiler pipeline. This approach relies on the Clang Cuda Toolchain up to Parallel Thread Execution pseudo-assembly language. After that, SASS code on the NVidia driver side is generated (see Figure 4).

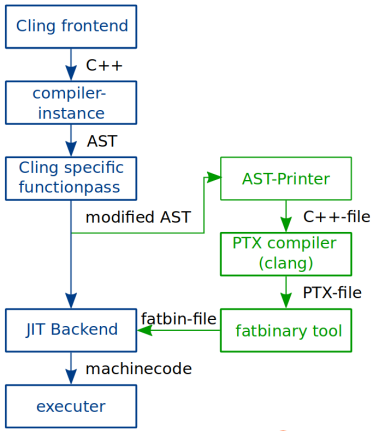

Figure 4. Diagram showing the Cling-CUDA compiler pipeline (courtesy of S. Ehring) [15]

4 Conclusions and Future Work

ROOT is being modernised in order to be ready for HEP data processing and analysis in the HL-LHC era, also through its parallelisation. For what concerns parallelisation, the emphasis has been put on the achievement of an ergonomic programming model, a runtime reduction as well as appropriate scaling on multicore platforms.

The present ROOT production release, ROOT 6.14, provides a substantial amount of parallelism. The scaling of Monte Carlo generator level studies not involving IO is at the level of ad-hoc solutions written by scientific software experts. The event processing rate of CMS was significantly boosted. It was demonstrated that factors in runtime could be saved for certain Machine Learning and minimisation use cases.

Significant ROOT related R&D activities are ongoing. A prototype of a system providing distributed declarative analysis based on multiple computing back-ends, among which Apache Spark, is being developed. Experimental support of heterogeneous architectures is being delivered, both in the case of ready-to-use perceptrons and interpretation of CUDA code.

References

[1] G. Apollinari, I. Béjar Alonso, O. Brüning, M. Lamont, L. Rossi, *High-Luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*, CERN Yellow Reports: Monographs (CERN, Geneva, 2015)

[2] O.S. Brüning, P. Collier, P. Lebrun, S. Myers, R. Ostojic, J. Poole, P. Proudlock, *LHC Design Report*, CERN Yellow Reports: Monographs (CERN, Geneva, 2004)

[3] R. Brun, F. Rademakers, *ROOT Object Oriented Data Analysis Framework*, in *New computing techniques in physics research V. Proceedings, 5th International Workshop, AIHENP '96, Lausanne, Switzerland* (1996)

[4] C. Pheatt, J. Comput. Sci. Coll. **23**, 298 (2008)

[5] *CMSSW Framework*, https://github.com/cms-sw/cmssw

[6] P. Mato, Tech. Rep. LHCb-98-064, CERN, Geneva (1998)

[7] J. Dean, S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (USENIX Association, Berkeley, CA, USA, 2004), OSDI'04, pp. 10–10

[8] D. Riley, *CMS and ROOT I/O*, in *ROOT I/O Workshop* (2018), https://indico.cern.ch/event/715802/contributions/2942558/

[9] F. Rademakers, P. Canal, A. Naumann, O. Couet, L. Moneta, V. Vassilev, D. Piparo, G. GANIS, B. Bellenot, wverkerke et al., *root-project/root: v6.14/02* (2018), https://doi.org/10.5281/zenodo.1292566

[10] J. Cervantes Villanueva, J.T. Palma Méndez, E. Tejedor Saavedra (2018), presented 13 Sep 2018

[11] S. Murali, E.T. Saavedra, E. Guiraud, D. Castro, J.C. Villanueva, D. Piparo, *Pyrdf* (2018), https://doi.org/10.5281/zenodo.1464080

[12] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin et al., Commun. ACM **59**, 56 (2016)

[13] M. Rocklin, *Dask: Parallel Computation with Blocked algorithms and Task Scheduling*, in *Proceedings of the 14th Python in Science Conference*, edited by K. Huff, J. Bergstra (2015), pp. 130 – 136

[14] D. Kirk, *NVIDIA Cuda Software and Gpu Parallel Computing Architecture*, in *Proceedings of the 6th International Symposium on Memory Management* (ACM, New York, NY, USA, 2007), ISMM '07, pp. 103–104, ISBN 978-1-59593-893-0, http://doi.acm.org/10.1145/1296907.1296909

[15] S. Ehring, *Adding CUDA Support to Cling: JIT Compile to GPUs*, in *ROOT Users Workshop* (2018), https://indico.cern.ch/event/697389/contributions/3085538