Journal of Physics: Conference Series

Expressing Parallelism with ROOT

CHEP IOP Publishing
IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 072022 doi:10.1088/1742-6596/898/7/072022

Expressing Parallelism with ROOT

D Piparo[1], E Tejedor[1], E Guiraud[1], G Ganis[1], P Mato[1], L Moneta[1], X Valls Pla[1] and P Canal[2]

[1]CERN CH-1211, Switzerland
[2]Fermilab, Batavia, IL., USA

E-mail: danilo.piparo@cern.ch

Abstract. The need for processing the ever-increasing amount of data generated by the LHC experiments in a more efficient way has motivated ROOT to further develop its support for parallelism. Such support is being tackled both for shared-memory and distributed-memory environments.

The incarnations of the aforementioned parallelism are multi-threading, multi-processing and cluster-wide executions. In the area of multi-threading, we discuss the new implicit parallelism and related interfaces, as well as the new building blocks to safely operate with ROOT objects in a multi-threaded environment. Regarding multi-processing, we review the new MultiProc framework, comparing it with similar tools (e.g. multiprocessing module in Python). Finally, as an alternative to PROOF for cluster-wide executions, we introduce the efforts on integrating ROOT with state-of-the-art distributed data processing technologies like Spark, both in terms of programming model and runtime design (with EOS as one of the main components).

For all the levels of parallelism, we discuss, based on real-life examples and measurements, how our proposals can increase the productivity of scientists.

1. Types of parallelism and support offered by ROOT

In this paper two main different approaches to parallelism can be identified: *multi-processing* and *multi-threading*. The former relies on the execution and management of sub-processes which act as workers. Communication among workers takes place through pipes and serialized messages. On the other hand, multi-threading parallelism foresees as workers threads sharing the same address space.

The two approaches are complementary. Multi-threading features less overhead in the creation of workers and the possibility to communicate through shared memory, it presents difficulties in presence of thread-unsafe code: resources must be protected not to incur in data corruption or instabilities.

From the programming model point of view, two useful categories to describe parallelism can be identified *explicit parallelism* and *implicit parallelism*. In presence of *explicit* parallelism when the user is given full control on the expression of parallelism, for example managing the protection of the resources or steering the lifetime of threads. If a library offers high-level interfaces and all details about parallelism are addressed internally in a transparent way for the user, one can talk about *implicit* parallelism.

ROOT [1] provides building blocks for the explicit and implicit expression of parallelism, both following the multi-threading and multi-processing approach.

This paper focusses on some of the features in the area of parallelism which are offered by ROOT starting from its 6.08 release [2].

1.1. The runtime for multi-processing

ROOT features an internal multi-processing engine. Its uniqueness is the tight coupling with the serialization capabilities of the framework, unique in the landscape of C++ libraries up to our knowledge. Arbitrarily complex C++ objects can be streamed between processes exploiting the same mechanism which allows to persistify data in ROOT format.

1.2. The runtime for multi-threading

ROOT adopted an external library to provide a threading runtime, TBB [3]. The functioning of ROOT in sequential mode does not depend on the presence of TBB: the build system allows to install it as an optional component. ROOT does not offer any reference to TBB components in any of its interfaces leaving the room to complement TBB with equally or more adequate and performant runtimes.

1.3. Cluster-wide executions

ROOT traditionally provides its own cluster management tool, the Parallel ROOT Facility, PROOF [4]. The discussion of PROOF goes beyond the scope of this document. Section 5.2 focuses on a complementary approach, i.e. the integration of ROOT with Apache Spark [5].

2. Parallel executors

A powerful but yet simple way to organize calculations in a parallel fashion consists in relying on widely-adopted patterns such as Map, Reduce and MapReduce. In order to provide access to such patterns, ROOT adopted an approach inspired by the one incarnated in the `concurrent.futures.Executor` of Python.

A minimal interface, `ROOT::TExecutor`, is adopted both by `ROOT::TProcessExecutor` and `ROOT::TThreadExecutor` and can be accessed as shown in code box 1 and 2 respectively.

Listing 1. TThreadExecutor usage.

```
ROOT::TThreadExecutor myExecutor(NWorkers);
auto myNewCollecction = myExecutor.Map(myLambda, myCollection);
```

Listing 2. TThreadExecutor usage.

```
ROOT::TProcessExecutor myExecutor(NWorkers);
auto myNewCollecction = myExecutor.Map(myLambda, myCollection);
```

3. Implicit parallelism

As described in section 1, implicit parallelism only requires the user to adopt certain high-level interfaces and the data treatment to achieve parallel execution is handled by the library internally. Details such as protection of resources, chunking of work and submission of tasks to the runtime are not exposed to the user.

The first area of ROOT where implicit parallelism was adopted was the processing of columnar data, the so called ROOT *trees*. This area has been chosen because of the immediate consequences parallelization would have had for data analysis and central data processing. The ability to read, decompress and deserialize several columns, also called *branches*, of the dataset simultaneously and to process several chunks of entries of the same dataset in parallel was implemented. The chunking respects the structure of the ROOT file on disk: a cluster of entries

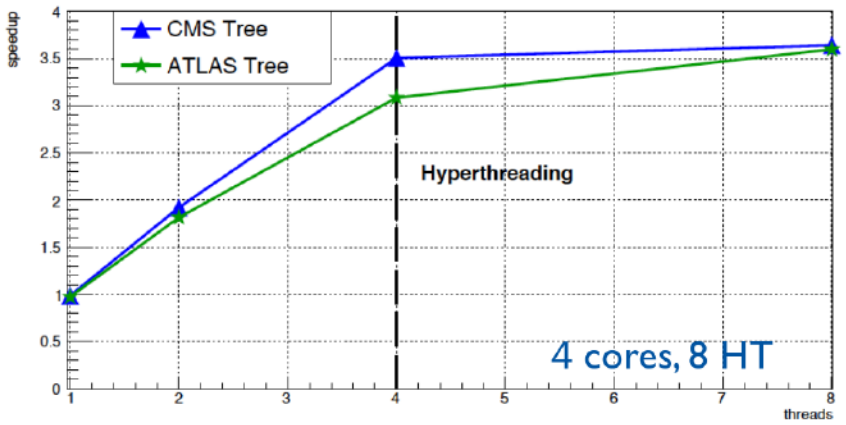

Figure 1. Reading, decompressing and deserializing a dataset. The CMS dataset features about 70 columns holding information about event simulation performed with Geant4 (*GenSim* data tier). The Atlas input is a dataset meant for final analysis (*xAOD* format): it features about 200 columns. The runtime reduction is between 3 and 3.5 on a machine offering four physical cores. No change in the user code was necessary, just a call to the global function `ROOT::EnableImplicitMT()`.

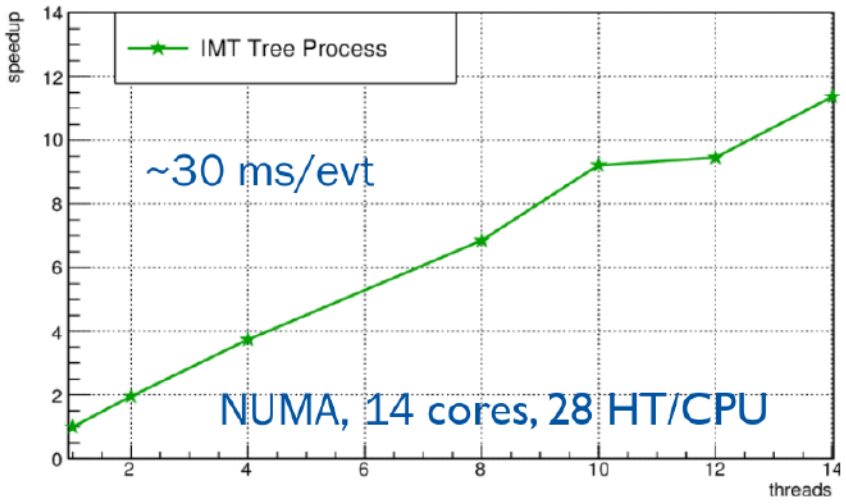

Figure 2. Basic analysis of generated particles tracks without the simulation of any detector. The entries are treated in parallel on machine featuring a processor with 14 physical cores. An unbalanced execution leads to the hindering of the scaling that stops at around 12: this is due to some tasks being shorter than the others. The class offered to steer parallel execution in this case is `TTreeProcessor`.

is treated per task therewith avoiding duplication of work for decompression and deserialization that occurs when creating items of work based solely on entry indexes.

Figure 1 and figure 2 show the scaling offered by ROOT when treating columns and entries in parallel.

It is important to observe that only a single, global switch is necessary to activate implicit parallelism within a program (see listing 3). No further change in user code is required.

Listing 3. Activation of implicit multi threading in ROOT.

```
ROOT::EnableImplicitMT();
```

A class ROOT offers in this context to analyse data sets stored in columnar format in ROOT files is the `ROOT::TTreeProcessor` [1] class - see listing 5. This utility allows the user to express the operation to be performed on a "view" of the dataset offered by `TTreeReader`. This approach is analogous to the *mapPartitions* functionality of Apache Spark.

4. Explicit parallelism and protection of resources

The first condition to be met in order to make the explicit management of parallelism accessible to users is to guarantee that the most commonly accessed code paths in ROOT are thread-safe. In a way which is analogous to the one available to enable implicit multi-threading, ROOT offers a centralized manner to activate thread safety of some of its code paths - see listing 4.

Listing 4. Activate thread safety of most common code paths in ROOT.

```
ROOT::EnableThreadSafety();
```

Upon invocation of that function

- interaction with the central type system from different threads is thread-safe
- interaction with the interpreter from different threads is thread-safe
- access to files and contained objects is thread-safe (one file per thread)

As expected from a foundation library, in addition to guaranteeing its own thread safety, ROOT offers also handy building blocks not available in the C++ Standard Library to protect user's resources. The three most notable examples are: `ROOT::TSpinMutex`, `ROOT::TThreadedObject<T>` and `ROOT::TRWSpinLock`. The first two classes are highly optimized implementations of the widely known concepts of spin mutex and read-write lock. It must be noted that, where needed, the components have been designed to be usable in combination with the STL. For example the `ROOT::TSpinMutex` can be used seamlessly together with the `std::condition_variable`.

The `ROOT::TThreadedObject<T>` is a smart pointer (see listing 5), offering a handy way to manage objects replicated for each thread offering an easy access to their final merging. The objects are made thread private lazily, i.e. upon invocation any of its methods via the overloaded arrow operator.

All the components described above are not dependent on the ROOT threading model but, on the contrary, are very versatile and threading model independent. Entities like TBB, OpenMP[6], pthreads or STL threads can be adopted but the user application.

Listing 5. Usage of ROOT::TThreadedObject to manage transparently instances of objects in a multi-threaded context. Parallelism is achieved via the TTreeProcessor class.

```
ROOT::TThreadedObject<TH1F> ptHist("pt_dist","pt_{dist}", 128, 0, 64);
ROOT::TTreeProcessor tp("myDataSet.root", "collisionEvents");
auto myAnalysis = [&](TTreeReader& reader) { // This is the work item
   TTreeReaderArray<Track> trks(reader, "tracks");
   while(reader.Next()) {
      for (auto& trk : trks) ptHist->Fill(trk.Pt());
   }
};
tp.Process(myFunction); // Here ROOT manages the parallelization
auto ptHistMerged = ptHist.Merge();
```

[1] After ROOT 6.08 also available as TTreeProcessorMT

5. Research and development lines

This section is dedicated to the description of two lines of research which are active at the time of writing: the creation of a ROOT subsystem allowing to express data analyses adopting a functional approach (or *functional chains*) and the integration of ROOT with Apache Spark.

5.1. Functional chains

The expression of data analyses as a pipeline of operations on the data without resorting to a loop over the entries is a paradigm being adopted by several data science tools such as R or Apache Spark. Other libraries such as ReactiveX [7] propose, in a slightly different context, similar solutions. The two basic ideas behind the ROOT approach to functional chains are:

- hide the event loop from the user, forcing the expression of data transformations and operations to a functional form
- offer a minimal interface to the user to allow maximal optimization of operations in the library in order to take advantage of the resources offered by the hardware

Presently a prototype in Python 6 is available and the development of a C++ counterpart is under rapid development.

Listing 6. Functional chain prototype in Python. A C++ version is under rapid development at the time of writing.

```
import ROOT
f = ROOT.TFile("aliDataset.root")
aliTree = f.Events
dataFrame = TDataFrame(aliTree)
# Prepare now an histogram after some filtering and draw it
dataframe.filter(sel1).map(func2).cache().filter(sel3).histo('var1:var2').Draw()
```

5.2. SparkROOT

HEP data offers tremendous opportunities for the expression of parallelism. Experiments deal with streams of statistically independent collision events, which can be processed independently. This feature has been exploited with great success by the LHC Computing Grid and other tools such as PROOF. The SparkROOT project addresses the evolution of the ideas behind the PROOF solution adapting them to modern technologies, in particular Apache Spark.

The Spark framework offers several desirable properties, among which the following ones

- it is a battle tested, fault-tolerant distributed computation framework
- it is widely adopted in the data science community and also industry
- it is written in Scala but bindings for Python are provided

The Python bindings are indeed the element this project line exploits to connect ROOT to Spark. This connection takes place thanks to PyROOT, the ROOT Python interface. This allows to take advantage of ROOT just-in-time C++ compilation capabilities and direct usage of all of the existing C++ libraries developed by analyzers and experiments. The software in the Spark driver and Spark workers is distributed with CVMFS [8] in order to achieve an identical environment. The user and experiment data is located on the EOS cloud storage. The ROOT files are read natively from EOS with the ROOT I/O libraries orchestrated via PyROOT.

These ideas have been tested demonstrating that it is possible to process a real-life dataset of an experiment (a CMS open dataset [9]) in ROOT format exploiting a general purpose Spark cluster.

In this case kinematic properties of simulated particle jets have been studies and the final result was a set of histograms. The cluster used was the CERN-managed *Hadalytic*. Thanks

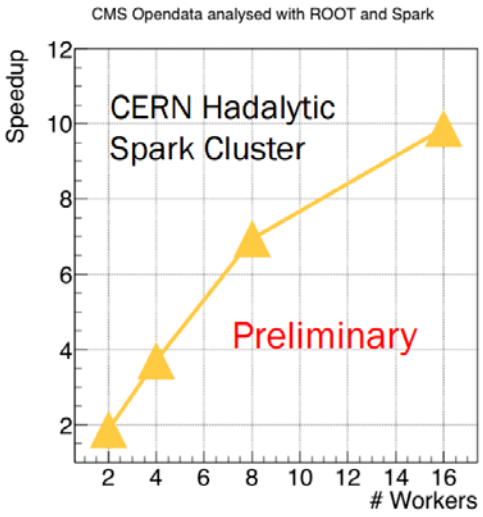

Figure 3. An analysis of simulated jets kinematic properties running on CERN's general purpose *Hadalytic* Spark cluster. The necessary software is served via CVMFS. The input data, a CMS open dataset, is located on EOS. The PySpark bindings in combination with PyROOT are exploited. The data is read natively via the ROOT I/O subsystem.

to the fact that all the necessary software is distributed with CVMFS, the client node a vanilla SLC6 node on which the correct environment was sourced. The preliminary results of the scaling are illustrated in figure 3.

6. Conclusions and future work

ROOT is an evolving framework and a variety of utilities have been added to release 6.08 in order to ease the expression of parallelism by its users' community. Examples are executors for running code on multicore architectures offering the Map, MapReduce and Reduce patterns, utilities to facilitate the expression of parallelism and protecting resources which complement the the existing tools provided by the Standard C++ Library. In addition ways to access parallelism with almost no effort on the user's side are now available. The new components are programmed according the most recent C++ standards and foresee, where it makes sense, full interoperability with elements in the Standard Library.

Particular attention was payed to the programming model exposed to the user but also to the runtime performance of use cases which are common in HEP. It was demonstrated that multithreaded analysis of ATLAS and CMS datasets exhibited an excellent scaling behavior on multicore architectures.

In the future ROOT releases, an incarnation of the Python based functional chain prototype will be provided as a complement to existing and widely adopted tools such as `TTree::Draw`. The Python layer necessary for the integration of ROOT with Spark via the respective Python interfaces will be added to the regular ROOT releases.

This work featured contributions from Fermilab operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy. The authors would like to thank the colleagues of the CERN IT-ST and IT-DB groups for the concrete help and fruitful discussions about the ROOT-Spark project.

References

[1] Brun R et al. 1997 *Nucl. Inst. Meth. In Phys.* vol 389

[2] `https://root.cern.ch/content/release-60800`
[3] Intel Threading Building Blocks `https://www.threadingbuildingblocks.org`
[4] Ganis G et al. 2009 *Proceedings of Science PoS(ACAT08)*
[5] Zaharia R et al. 2016 *Commun. ACM* vol 59
[6] Gaum L et al. 1998 *Computational Science & Engineering, IEEE* vol 5
[7] Meijer E 2010 *ACM SIGPLAN Commercial Users of Functional Programming*
[8] J Blomer et al. 2010 *Computer Communications and Networks (ICCCN)* doi: 10.1109/ICCCN.2010.5560054
[9] CMS collaboration 2016 *CERN Open Data Portal* doi: 10.7483/OPENDATA.CMS.JCDC.9CUH