Enhancements to multiprocessing in ROOT

August 2015

Author:

Enrico Guiraud

Supervisor:

Gerardo Ganis

CERN openlab Summer Student Report 2015

Project Specification

ROOT is a C++ library developed by CERN for data analysis and plotting.

PROOF is a multi-process framework to run ROOT data analytics in parallel on distributed resources. It implements a 3-tier client-master-workers architecture, with the master in charge of dynamic work distribution and collection of the results.

PROOF-Lite is a version of PROOF optimized for multi-core machines. It has found much success in the HEP community because of its direct and simple way of exploit multi-cores. However, it suffers from some difficulties coming from the fact that it inherits the setup technology from PROOF. In particular, setting up the environment of the workers requires the manual replication of the client configuration, a fragile process open to inconsistencies and failures.

In this project the student will prototype the possibility to create the workers from the client using process forking just before the execution of the query, so eliminating the need of any additional configuration of the workers and taking advantage of copy-on-write system feature.

Abstract

In this report we outline the results and the products of our **investigation of ROOT multi-processing capabilities**: not only we showed that **it is possible to fork a ROOT session** running in Linux environment, but we also built a **framework** that allows to easily exploit this capability to build **parallel applications** based on a client-worker architecture.

Moreover, using this framework as a foundation, we built a **new ROOT feature, the `Map` function**, inspired by python's `pool.map`: this function allows to execute the same task many times in parallel on different arguments, giving users an easy and lightweight access to multi-processing.

Section 1 is dedicated to a brief explanation of the current approach to multi-processing in ROOT, namely the PROOF and PROOF-Lite facilities.
Section 2 describes our new approach, its advantages and issues and some implementation details.
Section 3 describes the Map application, a few usage examples, some implementation details and the analysis of its performance.
Section 4 outlines some of the possible future developments of the work done so far.

Kudos

I would like to thank my supervisor **Gerardo Ganis** for the constant support he provided during all phases of my work: it's been great and refreshing to receive constant feedback and constructive criticism, and I feel like I have learnt a preposterous amount of things.

Danilo Piparo and **Pere Mato**'s ideas and suggestions were also invaluable for the development of the project; I am thankful he took the time to help me among his other numerous commitments.

More broadly, I feel grateful towards the whole **PH-SFT group** for the friendly and productive environment it provides: this has been the single most peaceful work experience I had, and I appreciate it immensely. Many thanks to the CERN openlab summer students program to make this possible.

Last but not least, I truly owe a debt to the **Linux community** for the tools and the support it provides me and every other Linux developer, regardless of differences in programming languages or operating system flavour. I am constantly amazed by the greatness that came from putting together many little snippets of code.

Table of Contents

Project Specification 2
Abstract 3
Table of Contents 4
1 Parallel tasks in ROOT: PROOF and PROOF-Lite 4
1.1 PROOF 4
1.2 PROOF-Lite 4
1.3 PROOF-Lite's problematics 5
2 A new framework for multiprocessing in ROOT 5
2.1 The 'multiproc' module 5
2.2 Pros and cons of the new approach 7
3 Building on top of the framework: parallelization utilities for the end-user 9
3.1 Tpool::Map 9
3.2 Tpool::MapReduce 10
3.3 Benchmarking 11
4 Possible further developments 12

1 Parallel tasks in ROOT: PROOF and PROOF-Lite

1.1 PROOF

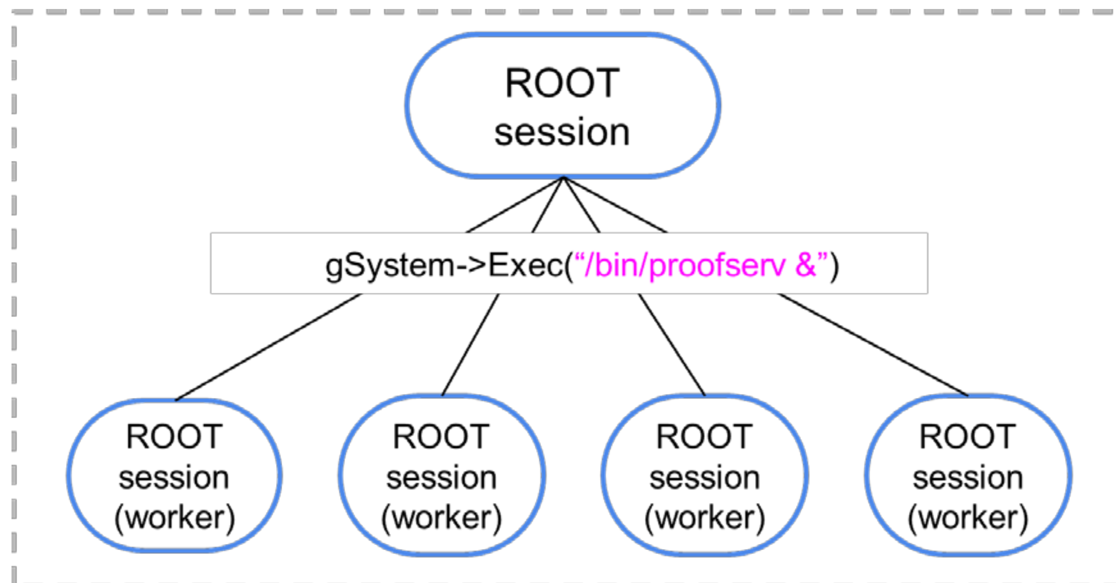
When it comes to executing ROOT tasks in parallel, be it data-mining or toy Monte Carlo simulations, there is currently basically one possible method: **PROOF**, the Parallel ROOT Facility. The original version of the framework was oriented to **multi-node parallelism** (i.e. exploitation of multiple machines, as opposed to multiple cores on one machine), and introduced a client-master-worker architecture in which a certain machine (the master) would distribute tasks to several others (the workers), collect results from them and possibly do some finalizing before quitting.

Obviously a workflow like this requires some **setup** of the nodes involved, in particular it requires that the worker nodes run a **specific service** that waits for connections and that can understand and execute orders, and obviously there is a requirement that the client be aware of this worker nodes and can connect to them.

1.2 PROOF-Lite

PROOF-Lite is a re-adaptation of PROOF that works on a **single machine**, using different processes as workers instead of different nodes. The operating system automatically assigns the worker processes to **different cores** to optimize resource consumption, allowing PROOF-Lite to use the same structure (and most of the times,

code) as PROOF: the main difference is that **workers are processes** running on several cores instead of several computers. Here is a representation of this framework:



1.3 PROOF-Lite's problematics

PROOF-Lite **suffers from being heavily based on PROOF**. In particular, the way the worker processes are spawned is not optimal: a `system` call is made that starts an ad-hoc binary: this binary basically runs a special ROOT session configured to wait for tasks, execute them and return the results. This means that for PROOF-Lite to work we need a different binary than the usual `root.exe`, and moreover and more importantly, since the workers are spawned as totally new processes, we need to set their environment "by hand", sending them information on which scripts and libraries were loaded in the client session and so on.

In short, we cannot take advantage of the fact that we are doing everything on the same machine: all the information about the client session environment is available in memory, but we cannot access it.

Finally, both PROOF and PROOF-Lite are designed with a specific application in mind: data-analysis by means of the `TSelector` ROOT class. This means that executing an arbitrary task in parallel is not a trivial task for the user.

2 A new framework for multiprocessing in ROOT

2.1 The 'multiproc' module

The first part of my project consisted in investigating **new possibilities for the creation of the workers** in the client/worker architecture of PROOF-Lite. We wanted to exploit the fact that both client and workers would be spawned on the same machine and make it possible for the workers to **access the client's memory**.

Threads come to mind: workers could be spawned as different threads of the client process (the main ROOT session) and they could execute tasks reading from a shared memory. The problem with a multi-threaded ROOT application is that the huge complexity of the ROOT framework and the heavy usage of global variables and singletons make shared memory a perilous path, prone to race conditions and crashes.

There's another option though: we might **fork the original process** multiple times to obtain several worker subprocesses whose memory is a copy of the original process', but is not the *same* memory. We wouldn't have to worry about workers writing to the same variable at the same time or other race conditions, because each forked process would have its own copry of the memory. The Linux **system call `fork`** can be used for this purpose. The Linux kernel optimizes the operation using **copy-on-write** techniques for the subprocesses' memory, which means that the memory of the original process is not duplicated upon forking but remains shared until a write is made on it – only then the relevant part of the memory is copied. In short, we obtain workers as seperate processes, each with its own memory, but this memory is a copy of the client's and it is not concretely duplicated in RAM until it's really needed thanks to copy-on-write.

The first week of my project was therefore spent on prototyping and investigating the forking of a ROOT (interactive) session. I soon discovered it was indeed possible to duplicate a session and connect it to its clone using ROOT's TSockets; for the subprocess to function as a worker, though, a few tweaks are needed:

- the forked session inherits the same file descriptors as the original, which on one hand means we can print errors happening in the workers on the clients console (handy!) on the other hand we have to explicitly close the workers' standard input otherwise what is typed in the keyboard will be randomly sent to the client or one of the workers, creating weird behaviour

- workers must not respond to graphical events and should not make use of graphical output. For this reason we make the forked process similar to a ROOT batch session

- interrupt signals like the one sent by pressing ctrl+C (SIGINT) propagate from the client to the worker processes, but the reaction to this kind of signal must be different. For the subprocesses, we set a custom signal handler that shuts down the worker in case a SIGINT is received. The behaviour can easily be extended to other types of signals

- workers' standard output and error are hidden by default, and only shown when needed. Many routine messages that the user gets while using a ROOT interactive session (e.g. the prompt, "root[0]") must not appear on console if sent by the workers

Finally, much time was spent on devising an **optimal way to handle the initial connection** between parent/client process and subprocesses/workers: the non-trivial part is that we would like the workers to be able to connect to the parent as soon as they are spawned, but the parent takes some time forking and subsequently accepting connections one by one. One of the last iterations of this process has **the workers create a unix socket and listen on it**, whereas the client process can "take its time" forking and connecting to each of the workers one by one. We might change this

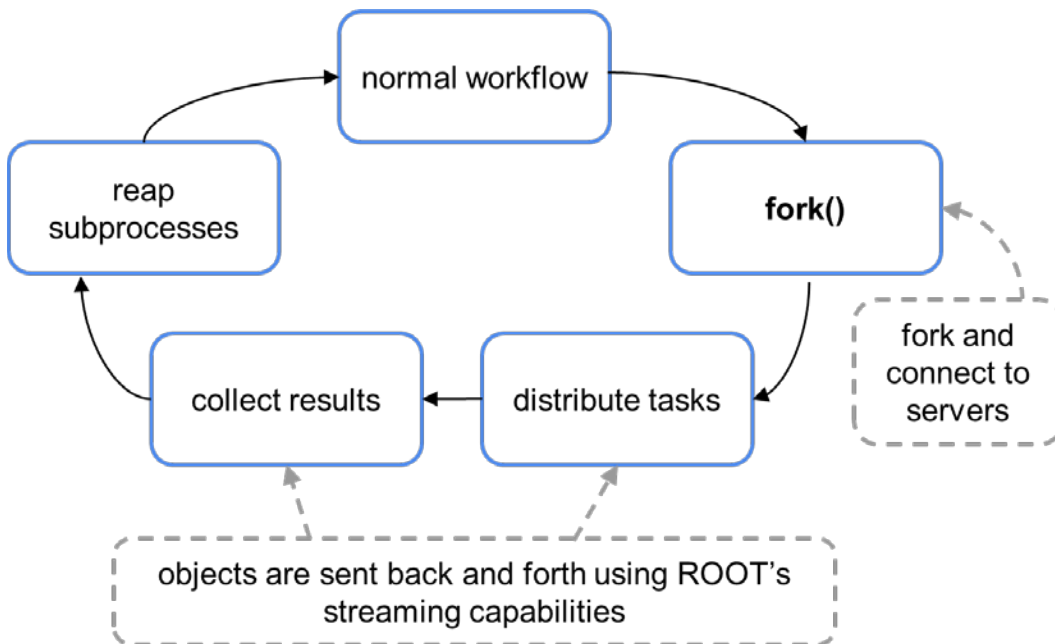
behaviour in the near future if a more robust (and possibly faster) one becomes available.

2.2 Pros and cons of the new approach

Let's recap what are the advantages and the issues arising from this new way of spawing workers using fork:

- workers can now access the client's memory
- memory footprint is small: workers' memory is copy-on-write
- the first two points mean that creating workers is now cheaper and faster
- workers can write directly to the client's console (to be used with moderation)

With all this, a new type of workflow becomes possible, in which we switch between single-process and multi-process operations with ease. We don't necessarily need PROOF-Lite anymore, but instead we can build a more general framework for multi-process applications. The user can use an interactive ROOT session or write his/her code, and when parallel computation must be performed we can simply spawn a few workers, distribute tasks to them, collect the results and finally go back to the original flow. This type of workflow is depicted here:

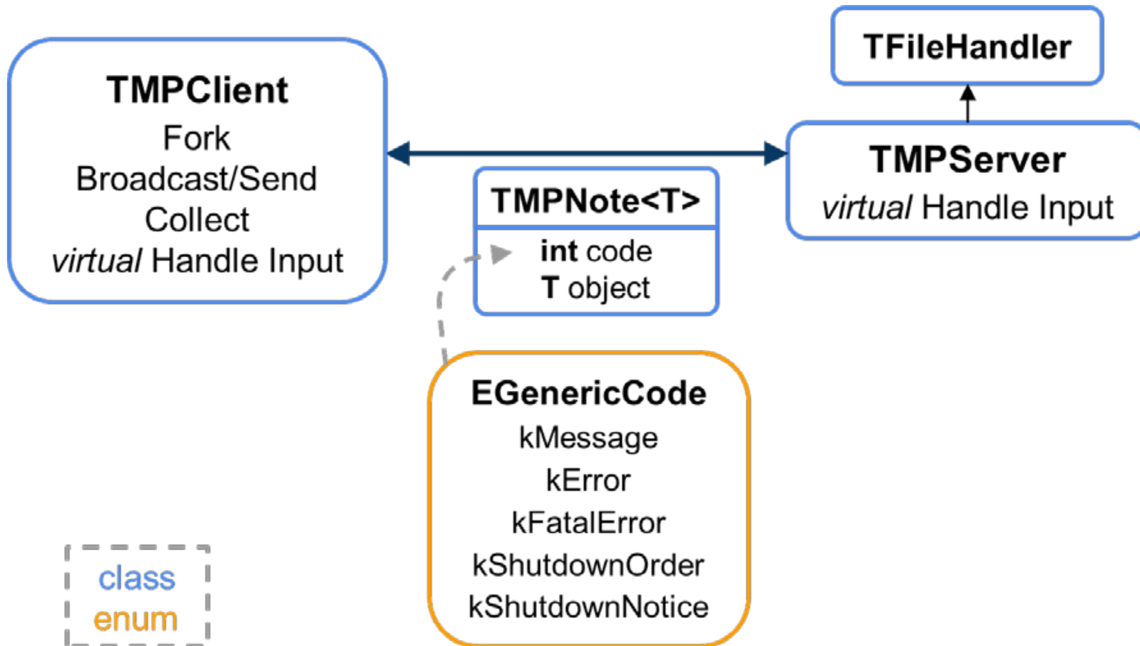


The main **disadvantage** of this approach is **portability**: `fork`, which is at the hear of the framework, is a Linux system call and as such is **inherently unportable**. A porting to windows might be possible by switching the call with a function call that reproduces a similar behaviour. Porting to OSX has turned out to be very hard due to ROOT relying heavily on the CoreFoundation library on this platform: CoreFoundation does not allow for non-thread-safe operations to be performed and automatically aborts execution when this happens. Needless to say, there can be non-thread-safe operations performed in

the parent process before forking and in the workers after forking, making it impossible to operate without CoreFoundation halting the program. More information on this behaviour can be found here.

2.3 Implementation details

The base framework for multi-process applications in ROOT looks like this:



TMPClient offers a **simple interface** for developers to access functionalities as **forking** of a ROOT session and **exchanging messages** with the workers. The `HandleInput` method of TMPClient and TMPServer can be overridden by inheriting classes to specify what actions are to be performed upon reception of a certain message. TMPClient and TMPServer will always handle messages with codes contained in the EgenericCode enumeration, which form the base communication protocol between client and workers. The object that is actually sent back and forth between processes is a TMPNote, a templated structure containing a code (which should tell what kind of message we are sending) and possibly an object (the actual content of the message). Due to limitations in ROOT's streaming capabilities, the object can only have a type for which we built a TMPNote dictionary entry (for more information about ROOT dictionaries, see here). We are currently working on relaxing this limitation and allow all types of objects to be sent back and forth between client and workers.

Multi-process applications that make use of this framework only need to define a client-like class that inherits (privately, to hide TMPClient's interface from the user) from TMPClient and defines its own HandleInput method, a server-like class that inherits from TMPServer and defines its own HandleInput method, and an enumeration similar to EGenericCode that establishes which kind of messages can be exchanged. Codes in EGenericCode have numeric values starting at 1000, which means that multi-process applications can specify their own codes using numbers lower than 1000 without incurring in clashes. The framework is intelligent enough to check the code of the

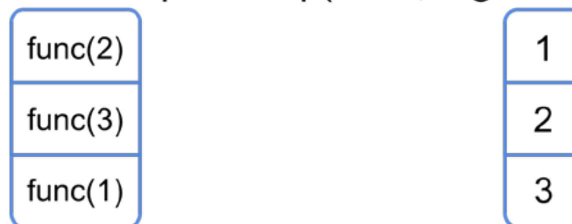
message received and then call the appropriate HandleInput method: TMPClient's for codes above 1000, inheriting class otherwise.

3 Building on top of the framework: parallelization utilities for the end-user

We wanted to test the base framework on a real-life application. This would provide ROOT users with a new facility for parallel execution of tasks and at the same time would test our software's possibilities and limitations. So we took inspiration from python's `Map` function and tried to implement a similar thing for ROOT users. A short investigation showed that a feature like this is not present in the standard library nor in other major or minor support libraries, such as boost or github projects. We were doing something useful and new.

3.1 TPool::Map

So what does Map do? Imagine you want to execute the same function on many different arguments, or you need to run the same analysis on many different datasets; more in general, let's say you want to execute the same task many times, maybe with different parameters. Normally you would use a for loop or something similar, and execute tasks sequentially. Map takes care of executing tasks in parallel and return the end result. Here is a sample call to Map:



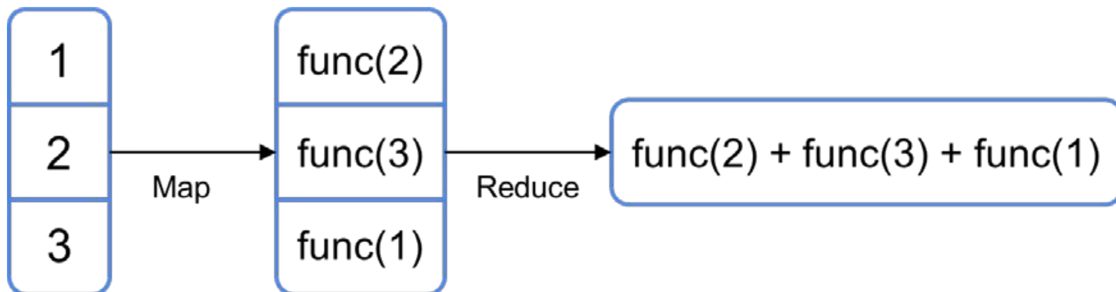
And here is a more convoluted one, in which we imagine to be running a ROOT interactive session in which we load and compile a macro. The macro takes three parameters, and we use a lambda function to fix the first two and only change the filename we pass to the macro. Map executes the lambda with the three different filenames in parallel, and the lambda in turns calls the macro. Both this example and the former are valid syntax in TPool::Map.

```
TPool  pool(8)

.L  myMacro.cxx+

auto res = pool.Map(
        [ ] (string f) { return myMacro("opt", 12, f); },
        { "file1", "file2", "file3" }
        )
```

The application is heavily templated to allow the usage of many different types when calling Map: everything that is a C++ *callable* object can be passed as a function. STL collections and ROOT collections are allowed as arguments, as well as initializer lists (as shown in the second usage example above). The returned type is an STL vector if the argument type is an STL collection or an initializer list, a TObjArray (ROOT's native collection type) otherwise. This way users can stick to using ROOT's collection types if they want, and at the same time users that rely on standard library can keep using it.

3.2 TPool::MapReduce

Another method that we made available is `MapReduce`: it allows the user to specify a function through which the set of results returned by `Map` can be squashed into a single object (e.g. sum of integers, merge of histograms…).



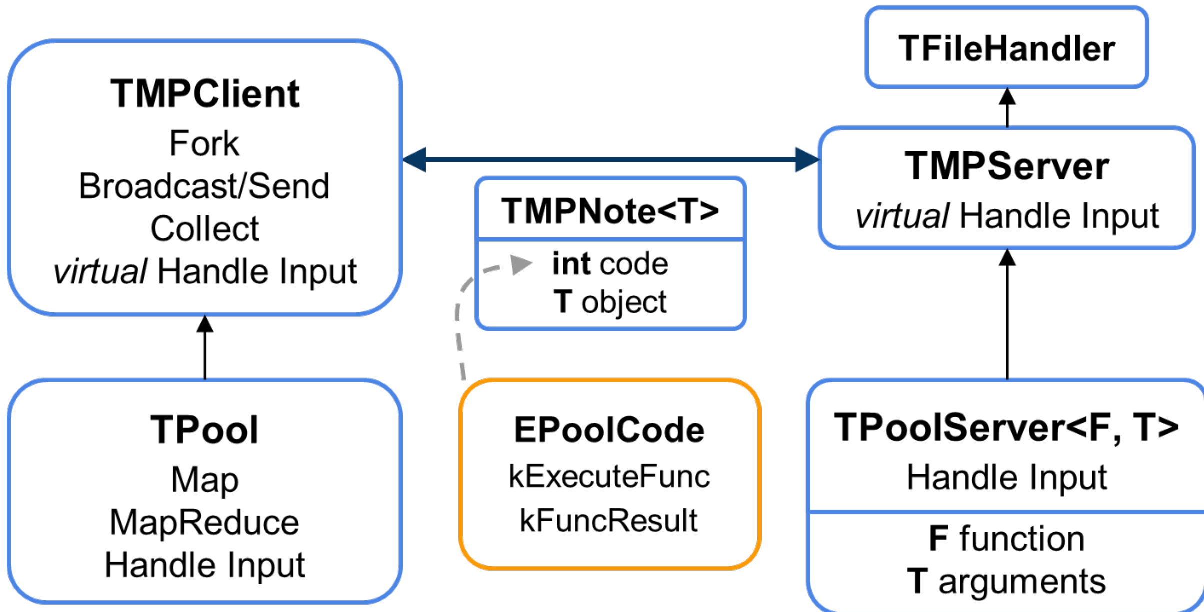
A call to the `MapReduce` method looks like this:

```
pool.MapReduce( func, args, reduce_func )
```

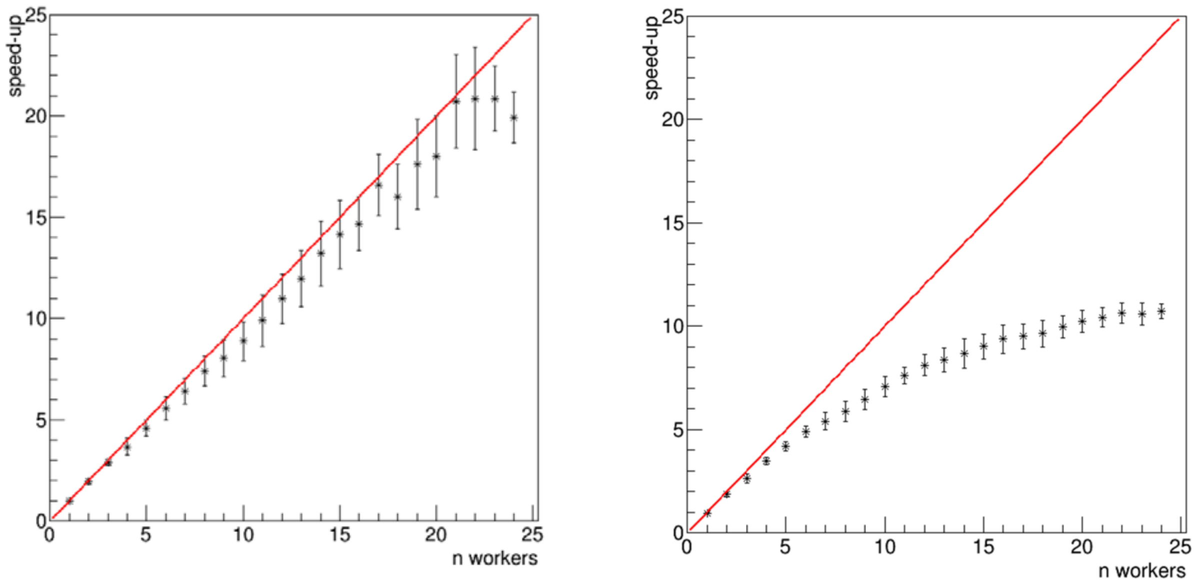
The reduce operation is currently not parallelized, but in the future we would like to implement a "greedy worker" paradigm, in which workers do not return the result of their calculations to the server as long as there are more arguments to be processed in the queue. Workers reduce the results of their own calculations and only send the result back to the client once there are no more arguments to be processed. The client then performs one last reduce operation with the results of each worker.

3.3 Implementation details

TPool takes advantage of the base framework, hiding the interface of TMPClient through private inheritance and only exposing the Map and MapReduce methods to the users. A new set of message codes is put in place and new HandleInput methods are defined that override the ones of the base classes.



3.4 Benchmarking



The graph on the left is the speed-up results of executing a trivial task (filling an histogram with 1e9 random points) with an increasing number of workers (i.e. cores) on a 24 core Linux machine mounting Ubuntu. The number of points has been equally divided between workers, and no merging of histograms is performed. The ideal speed-

up is outlined in red: that would be the speed increase we would have if we didn't execute the task through TPool and we just divided the time taken by one core by the number of cores used. We can see TPool has a good speed-up up until 23 workers, and a sharp decrease in performance at 24: this is because if *24* workers are spawned, that makes it *25* processes running on the machine (the client counts too!).

The graph on the right is what happens when the histograms produced by each worker are merged. The time needed for this extra operation increases linearly with the number of workers since the reduce operation is not parallelized (yet!), which makes for worse performance at an higher number of workers.

4 Possible further developments

As for the future, several improvements to both the base framework and the Map/MapReduce application can be made, which I didn't have time to add until now. I will just give a quick overview of the main ones.

4.1 Improvements to the base framework

- integration with PROOF-Lite: as per the original goal of my project, we can and should make use of the base framework and its new way of spawning workers by forking the client process in PROOF-Lite. This would certainly result in an improvement in performance, cleaner implementation and higher flexibility
- integration with TTree::Draw: this method is used to read several variables present in a tree structure (a TTree ROOT class) and draw them on a histogram.

4.2 Improvements to TPool

- explicit support for TSelector: TPool::Map might implement a signature that takes a TSelector as an argument and performs its task as it would be done via PROOF-Lite, allowing for a much simpler syntax and much friendlier usage
- support for TTree: many operations on TTree's can be parallelized, and TPool::Map offers a simple interface to do so
- parallel reduce: as explained in the section about `MapReduce`, we have a simple way to distribute reduce operations between servers that is yet to be implemented

5 Conclusions

A new way of spawning workers has been developed for the client/workers architecture of ROOT multi-process facilities. It makes use of forking to create workers quickly and (thanks to copy-on-write optimizations performed by Linux kernel) it has a small memory footprint. The base framework grants developers

access to this functionality to define their own multi-process applications and their own protocol of communication between client and workers.
A real-life application of this framework has been developed: the TPool class implements a Map method, similar to python's `pool.map`, that grants high speed-up on parallel execution of tasks on multicore machines.
Improvements can be made on both parts of my project, and I do hope to have a part in that too.