Parallelization and vectorization of ROOT fitting classes

X Valls Pla[1,2], **L Moneta**[1]
[1]CERN, [2]Universitat Jaume I

E-mail: root-dev@cern.ch

Abstract.
We introduce parallelization and vectorization in ROOT mathematical and statistical libraries in order to take full advantage of new computer architectures and to maximize the CPU usage with an increasing amount of data to analyze.

As part of this effort, we define in ROOT new generic classes supporting a task-based parallelization mode, which can be used for a wide range of computational tasks in the field of High Energy Physics. We also include support for different SIMD libraries, by including in ROOT a new generic library for vectorization.

These different parallelization tools are applied together when parallelizing the minimization process for solving fitting problems. We report on the improvements obtained by adding the support for SIMD vectorization and multithreaded parallelization when fitting ROOT histograms and datasets represented by ROOT trees.

1. Introduction

Many operations in High Energy Physics (HEP) data processing can be classified in generic patterns or algorithms reproduced over several analyses, adapted to the specific problems at hand.

Our goal is to identify these patterns in the HEP analysis processes, and integrate in the ROOT [1] software framework a set of tools to parallelize them across multiple levels, starting from the most common patterns, such as MapReduce, or the ones that provide more potential, such as the integration of tools for vectorization and the adoption of SIMD types.

In this paper, we describe the generic tools developed in ROOT and their contribution to the parallelization of fitting in ROOT.

2. Tools for parallelism

The strategy that ROOT adapts to speed up computations consists in exploiting both data-level and task-level parallelism. For this purpose, ROOT provides several generic classes for the expression of parallelism at different levels.

The tool used to implement task-level parallelism, ThreadExecutor [2], provides the MapReduce [3] framework to partition the evaluation into tasks and distribute the computational workload among the system threads.

In order to accomodate data-level parallelism, we rely on a dedicated library—VecCore [4]—to abstract SIMD operations and data-types from the code, making the solution portable between architectures with different SIMD instructions sets.

2.1. Task-level parallelism: TThreadExecutor

TThreadExecutor is a task-oriented, multithreaded MapReduce tool for ROOT, which provides a simple programming model for parallel MapReduce operations on loops operating on independent data. Its interface includes operations such as *Map*, *Reduce*, *Foreach* and even clustered mapping with partial reduction. TThreadExecutor is currently implemented as a façade to Intel's Threading Building Blocks (TBB) [5], specifically the function to parallelize loops (`tbb::parallel_for`), exploiting TBB's scheduling and work stealing mechanisms.

Listing 1 exposes a basic example of the usage of TThreadExecutor. First we define our map function, which implements the task to be computed in parallel. Then we propose our reduction function accumulating the intermediate result from the map and we initialize TThreadExecutor and the pool of threads. Finally, by calling `TThreadExecutor::MapReduce`, we perform the operation in parallel.

```
auto mapFunc = [](const UInt_t &i){
    return i+1;
};

auto reduceFunc = [](const std::vector<UInt_t> &mapV){
    return std::accumulate(mapV.begin(), mapV.end());
};

ROOT::TThreadExecutor pool;
pool.MapReduce( mapFunction, ROOT::TSeq<int>(100), reductionFunction);
```

Listing 1: Use example for TThreadExecutor: parallel sum of the first 100 nonnegative integers.

Currently, TThreadExecutor is used in ROOT fitting to compute the maximum likelihood and the least squares functions in parallel. In TMVA [6], it is utilized to evaluate Boosted Decision Trees and to process Deep Neural Networks. It is also applied for performing implicitly multithreaded operations in ROOT I/O (reading, parallel writing, deserialization and decompression of tree branches in parallel) and in the parallel execution of functional chains in TDataFrame [7].

2.2. Data-level parallelism: VecCore

VecCore provides efficient vectorization on a variety of platforms by offering an abstraction layer on top of the libraries Vc [8], UME::SIMD [9] and the language extension CUDA, which are supported as optional backends as well as a fallback scalar one in case SIMD operations are not available. VecCore allows developing architecture-oblivious code which will map to the appropriate backend specific types, methods and instructions.

ROOT integrates VecCore by defining new SIMD array types, for example `ROOT::Double_v` or `ROOT::Float_v`, representing double and single precision vectors respectively.

3. Fitting parallelization

The computation process used for fitting is parallelized in two independent steps. We parallelize at task-level the fitting objective function, e.g. the maximum likelihood, and at data-level the model function which is provided by the user, e.g. a Gaussian function when fitting a histogram.

Figure 1 illustrates the fitting parallelization process both at task-level and data-level. We start by dividing the data into clusters of points of equal size (*Chunking*). This step is performed by TThreadExecutor and the user can specify the number of desired chunks. TThreadExecutor receives the clustered data and distributes the tasks among threads for their evaluation in the multithreaded Map stage (*Distribution*). In this stage, the model function is evaluated on the data points and, when the user provides a vectorized function, the evaluation is done on several points simultaneously (*Evaluation*). In both cases, the map function accumulates the results of the objective function evaluation into one partial result per thread.

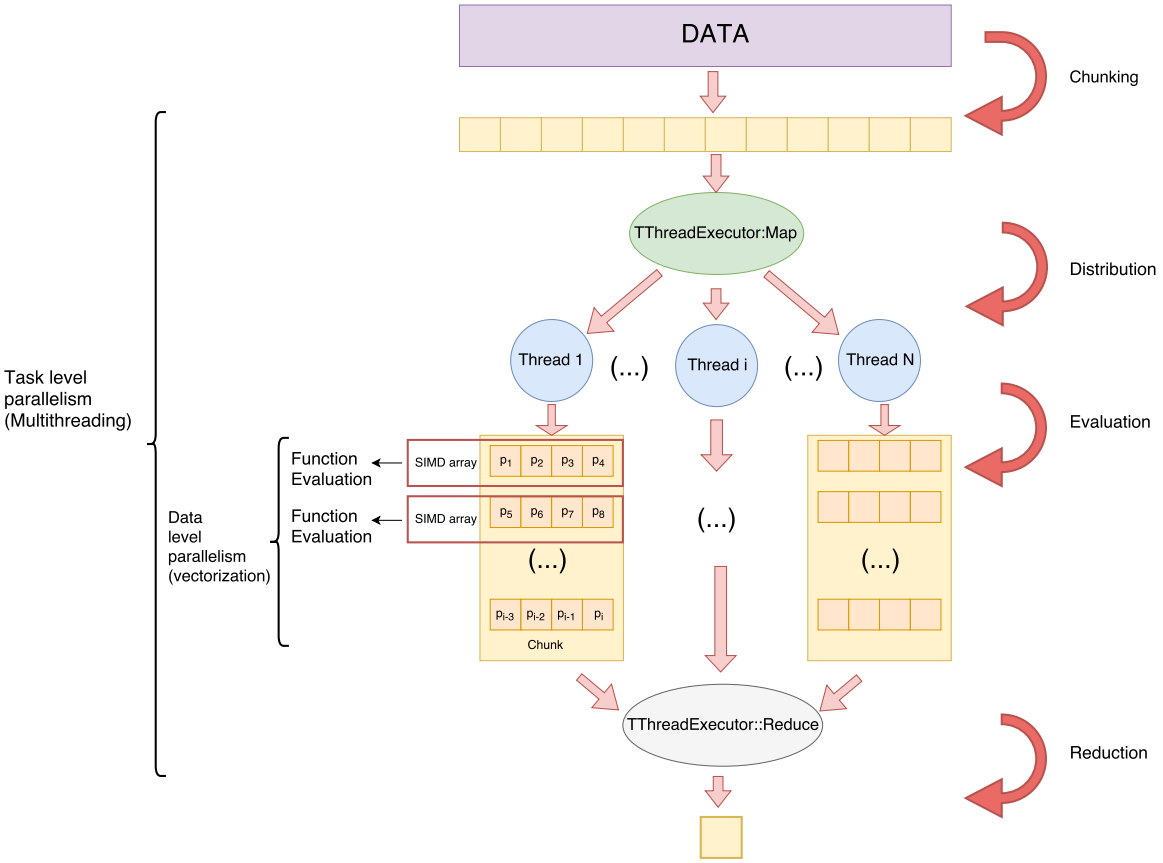


Figure 1. Depiction of the parallelization of the fitting. The parallelization at task level involves chunking, multithreaded mapping (distribution and evaluation) and reduction to a single result. At data level, we vectorize the model function evaluation.

The partial results are then returned to the main thread, which combines them into a final result representing the objective function of the fit in a reduction step applied by `TThreadExecutor::Reduce` (*Reduction*).

3.1. Case example

As a case example for benchmarking, we choose a fit of the diphoton invariant mass distribution resulting from a Higgs boson. Listings 2 and 3 offer a comparison of the scalar code for this benchmark fit against the code needed for a fully parallelized implementation of the same fit.

Note that in Listing 3 we provide parallelization and vectorization, while maintaining the previous programming model. This is an important achievement: users can benefit from multiple levels of parallelization without modifying their original code. Starting from line 9 in Listing 2 and line 13 in listing 3 the code related to the fitting operations is exactly the same. The only changes required for fitting the data in the vectorized and parallelized cases—Listing 3—are additions on top of the current workflow.

First, in order to profit from implicit task parallelization, the user only needs to call `EnableImplicitMT` (line 10 of Listing 3). This will activate all the mechanisms in ROOT that provide thread safety and parallelize all the operations that allow implicit multithreading in ROOT, such as the fitting. For vectorization, the user needs to provide a vectorized function,

```
//Example Fit: Implementation of the scalar function
double func(const double *data, const double *params)
{
   return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
          params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
          ((*data) * (0.01)) * ((*data) * (0.01))));
}

auto f = TF1("fScalar", func, 100, 200, 4);
f.SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
h1f.FillRandom("fScalar", 1000000);
h1f.Fit(&f);
```

Listing 2: Traditional scalar implementation where the model function is composed of a Gaussian signal component plus an exponential background function,

```
1  //Example Fit: Implementation of the vectorized function
2  ROOT::Double_v func(const ROOT::Double_v *data, const double *params)
3  {
4     return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
5            params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
6            ((*data) * (0.01)) * ((*data) * (0.01))));
7  }
8
9  // Enable implicit parallelization
10 ROOT::EnableImplicitMT();
11
12 //This code is totally backwards compatible
13 auto f = TF1("fvCore", func, 100, 200, 4);
14 f.SetParameters(1, 1000, 7.5, 1.5);
15 TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
16 h1f.FillRandom("fvCore", 1000000);
17 h1f.Fit(&f);
```

Listing 3: Vectorized plus parallelized implementation.

which in this case only means changing the data parameter type and the return type —to the new `ROOT::Double_v` type, representing a SIMD array of double—as shown in line 2 of the Listing 3.

3.2. Results

We perform three different types of fits, χ^2 and Poisson likelihood in the binned case and Maximum Likelihood in the unbinned case, to the invariant mass distribution studying the speedup in each one of this cases. We measured the fitting times normalized to the number of objective function calls made by the fitter, as the nature of minimization problems makes the number of calls fluctuate between examples, due to numerical precision, and influence the measured times.

Figure 2(a) reports the speed up attained by the parallel fitting on a desktop server, a 4-core Haswell machine with 8 GB of RAM and distinct sets of vectorization instructions and different evaluation functions.

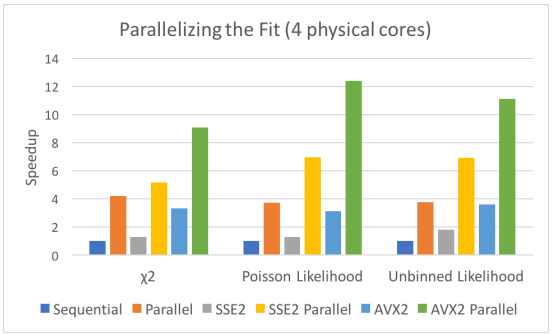
In Figure 2(b), we show the scaling of multithreaded operations on a 14-core Xeon processor with AVX2 and 64 GB of RAM. While it scales in the multithreaded scalar case, vectorization limits the total speed up.

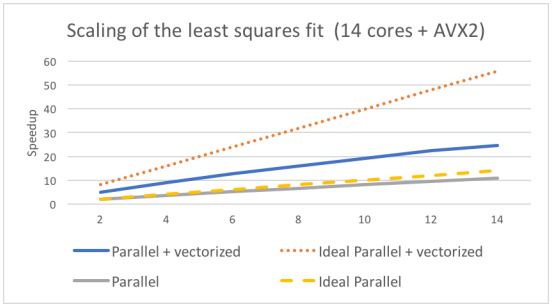
It is also interesting to evaluate how different compilers behave. The results between compilers are similar (Figures 3(a) and 3(b)), with ICC slightly outperformed by Clang and GCC.

4. Conclusions

With the appropriate tools, we can exploit parallelism across multiple levels when fitting models in ROOT while preserving the user interfaces. We have shown how the users can vectorize and parallelize their original code and profit from a significant speedup in a convenient fashion.

Results differ between different fits, but in all those three implemented we obtain good speed

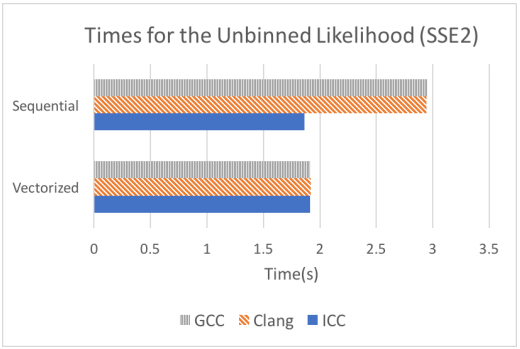




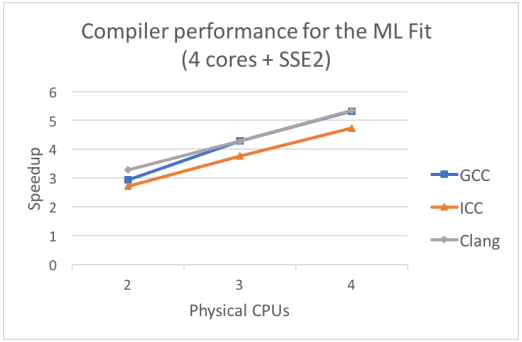
(a) Performance of the fit (4-cores Broadwell, 8GB RAM). Evaluated over 120k bins in the binned fits cores. and 120k points in the unbinned fit.

(b) Scaling of the fit with an increasing number of

Figure 2. Performance measures. Operations carried out in double precision.



(a) Autovectorization turned off.



(b) Compiler performance.

Figure 3. Compiler comparison: gcc 6.2, clang 3.8, icc17. Carried out in double precision.

ups when vectorizing and remarkable ones when parallelizing, thanks to TThreadExecutor. The combination of both task-level and data-level parallelism produces the best results, and both speed ups and execution times are consistent across compilers.

5. References

[1] Brun R and Rademakers F 1997 Root - an object oriented data analysis framework *Proc. AIHENP'96 Wksh.* vol A389 ed in Phys N I M pp 81–86

[2] Piparo D, Tejedor E, Guiraud E, Ganis G, Mato P, Moneta L, Pla X V and Canal P 2017 Expressing parallelism with root *Journal of Physics: Conference Series* vol 898 (IOP Publishing) p 072022

[3] Dean J and Ghemawat S 2008 *Communications of the ACM* **51** 107–113

[4] Amadio G, Canal P and Wenzel S Veccore [software]: release v0.4.2, available from https://github.com/root-project/veccore/ [accessed 2017-10-28]

[5] Reinders J 2007 *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism* (O'Reilly Media, Inc.)

[6] Hoecker A, Speckmayer P, Stelzer J, Therhaag J, von Toerne E, Voss H, Backes M, Carli T, Cohen O, Christov A *et al.* 2007 *arXiv preprint physics/0703039*

[7] Amadio G, Blomer J, Canal P, Ganis G, Guiraud E, Mato Vila P, Moneta L, Piparo D, Tejedor E and Valls Pla X 2018 Novel functional and distributed approaches to data analysis available in root *18th Int. Wksh. on Advanced Computing and Analysis Techniques in Physics Research* (IOP Publishing)

[8] Kretz M and Lindenstruth V 2012 *Software: Practice and Experience* **42** 1409–1430

[9] Karpinski P and McDonald J 2017 A high-performance portable abstract interface for explicit simd vectorization *PMAM@ PPoPP* pp 21–28