Utilities for parallelism at task-level and data-level in ROOT

Xavier Valls

ROOT
Data Analysis Framework
https://root.cern

Motivation

Reduce the time physicists spend processing and analyzing data

Motivation: How

- Improving the execution time of the analysis
  - processing an increased amount of data per time unit

- Improving the programming model of the analysis
  - reducing the time physicists spend dealing with the complexity of the tools
  - putting the spotlight on the analysis instead of on its implementation

Goals

- Build a set of tools in ROOT to provide parallelization at task-level that can be applied recurrently throughout ROOT's codebase.

- Introduce data-level parallelism in ROOT mathematical libraries

- Parallelize the fit minimization process at task-level and data-level.

- Deploy and leverage these tools in ROOT critical performance areas and analyze their impact on performance.

Goals (2)

Improve current code used for parallelization in:

- User friendliness
- Performance
- Reliability
- Reusability

Data-level parallelism

Data-level parallelism

- Achieved in ROOT by exploiting SIMD operations on arrays of data (vectorization)
- Integration of VecCore in ROOT's mathematical libraries
- Introduction of two new SIMD types in ROOT: ROOT::Double_v and ROOT::Float_v

VecCore

- Provides efficient vectorization and portability by offering a layer of abstraction on top of each of its exchangeable backends: Vc, UME::SIMD, CUDA

- Allows the development of architecture-oblivious code that maps to the appropriate backend specific types, methods and instructions

- -Dveccore=ON -Dvc=ON

[See Guilherme's talk: Support for SIMD Vectorization in ROOT]

Deployment

Mostly in the mathematical libraries:

- Fitting
- TFx
- TFormula
- TMath (work in progress)

Performance

Case 1

$$f(x,\theta) = \theta_0 e^{-\frac{(x-130)^2}{2}} + \theta_1 e^{-\left(\theta_2 \frac{x}{100} - \theta_3 \left(\frac{x}{100}\right)^2\right)};$$

Case 2

$$f(x,\theta) = -\theta_0 \frac{(x-130)^2}{2} + -\theta_1 \left(\theta_2 \frac{x}{100} - \theta_3 \left(\frac{x}{100}\right)^2\right);$$

Case 3

$$f(x,\theta) = \theta_0 x + \theta_1 x^2 + \theta_3 x^3.$$

Case	Implementation	Scalar time (ns)	Vectorized time (ns)	Speed up
Case 1	Free function	11.6	12.1	0.96
	TF1 (free function)	28.5	13.2	2.15
	Formula	22.9	12.8	1.79
	TF1 (formula)	26.2	13.1	2.00
Case 2	Free function	1.89	0.541	3.50
	TF1 (free function)	8.48	2.39	3.55
	Formula	22.8	13.0	1.76
	TF1 (formula)	27.0	12.9	2.10
Case 3	Free function	1.50	0.467	3.22
	TF1 (free function)	7.91	1.93	4.09
	Formula	22.1	8.36	2.64
	TF1 (formula)	24.6	8.50	2.90

Vectorized speed up using VecCore with respect to the scalar execution

Task-level parallelism

Task-level parallelism

- Introduced the Executors, implementations of the MapReduce pattern:
  - TExecutor (common interfaces: Map, Reduce, MapReduce)
  - TSequentialExecutor (sequential implementation)
  - TProcessExecutor (multiprocess implementation, old TPool class)
  - TThreadExecutor (multithread implementation)

- TPoolManager, a centralized manager for the TBB task scheduler.

The Executors

- Convenient and flexible programming model (TProcessExecutor):

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

Interface: https://root.cern/doc/master/classROOT_1_1TExecutor.html

The Executors

- Convenient and flexible programming model (TThreadExecutor):

```
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);
```

Interface: https://root.cern/doc/master/classROOT_1_1TExecutor.html

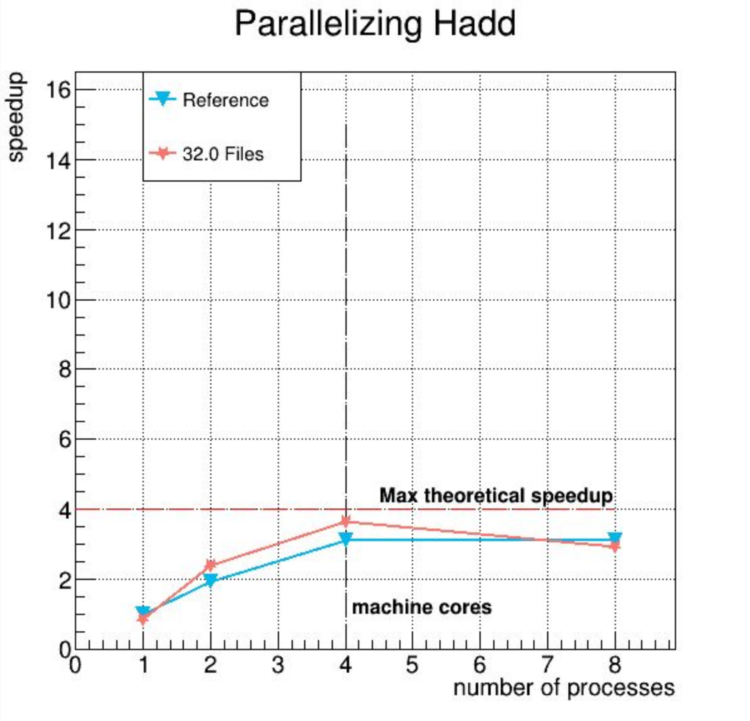
Deployment

The executors have been deployed throughout ROOT’s codebase, supporting implicit parallelism

- TProcessExecutor:
  - hadd -j
- TThreadExecutor:
  - TMVA: BDT, DNN,...
  - Fitting
  - I/O: TTreeGetEntry, TTreeProcessorMT, TTreeCacheUnzip,...
  - RDataFrame

Performance: hadd

125 MB file duplicated N times, containing hundreds of histograms in a really complex structure of directories (simplified from CMS quality monitoring)



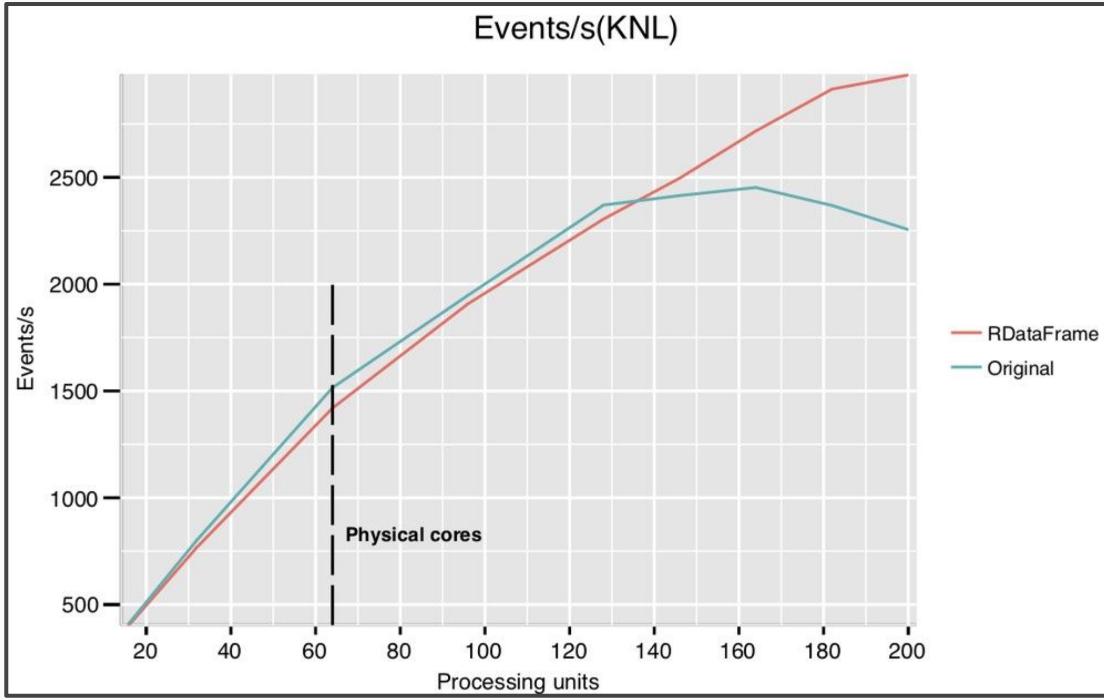
Performance: RDataFrame

Monte Carlo QCD low-pt events generation+analysis on the fly

Ad-hoc implementation (patched ROOT5 & POSIX threads) Vs RDataFrame

[See Enrico's talk:RDataFrame: ROOT's Declarative Approach for Manipulation and Analysis of Datasets]

KNL 64 physical cores, 256 threads



Performance: RDataFrame

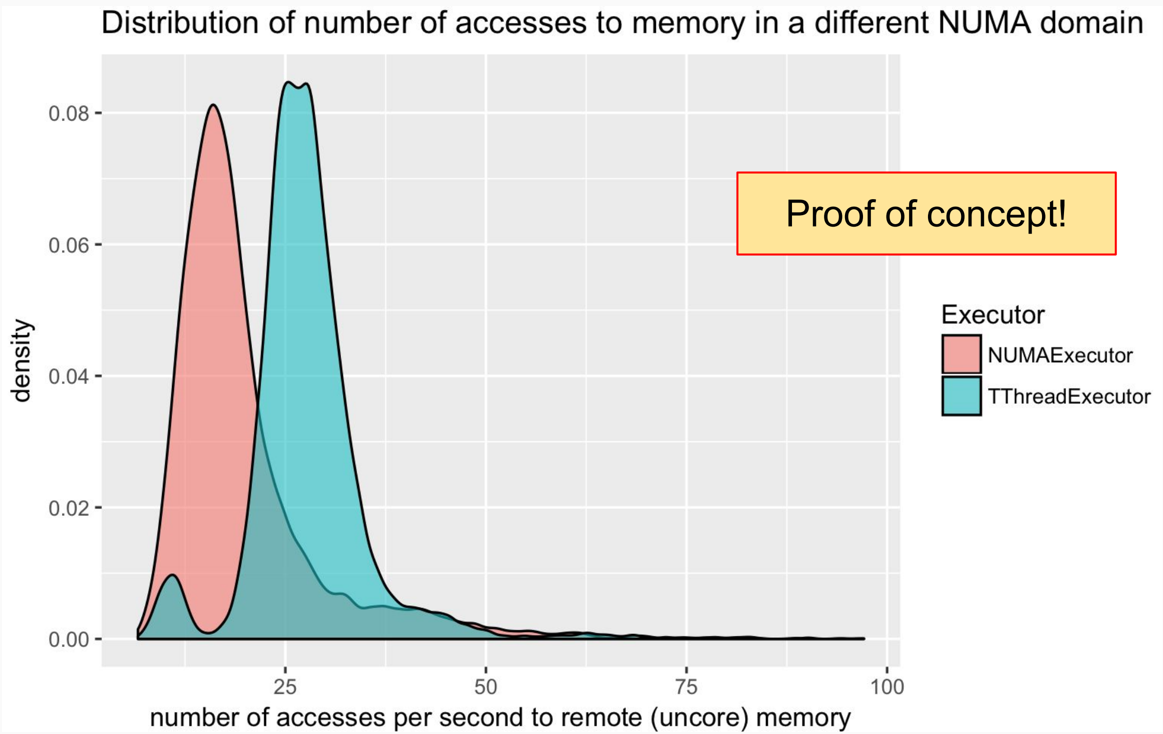
[See Enrico’s talk:RDataFrame: ROOT’s Declarative Approach for Manipulation and Analysis of Datasets]

Performance: TNUMAExecutor

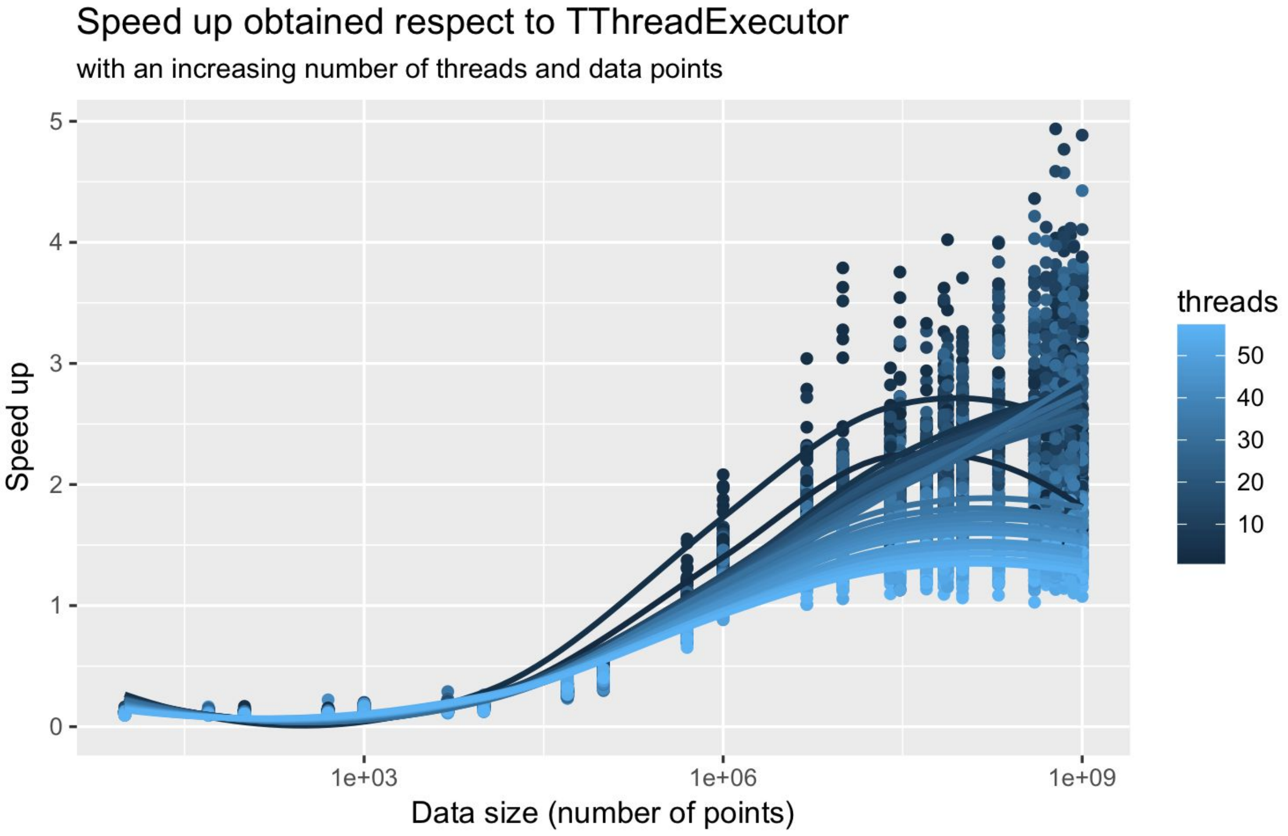
Intel Xeon, 2 NUMA domains of 14 threads each (28 hyperthreading)

We can reduce uncore events in NUMA architectures by dividing the workload into as many processes as NUMA domains and conceal the multithreaded execution of each process in a NUMA node.

30% reduction in accesses to remote memory when performing, over 9.5 million events, an unbinned fit of the diphoton invariant mass distribution resulting from a Higgs boson, with an analytically computed integral. Resulting in **1.8x speed up.**

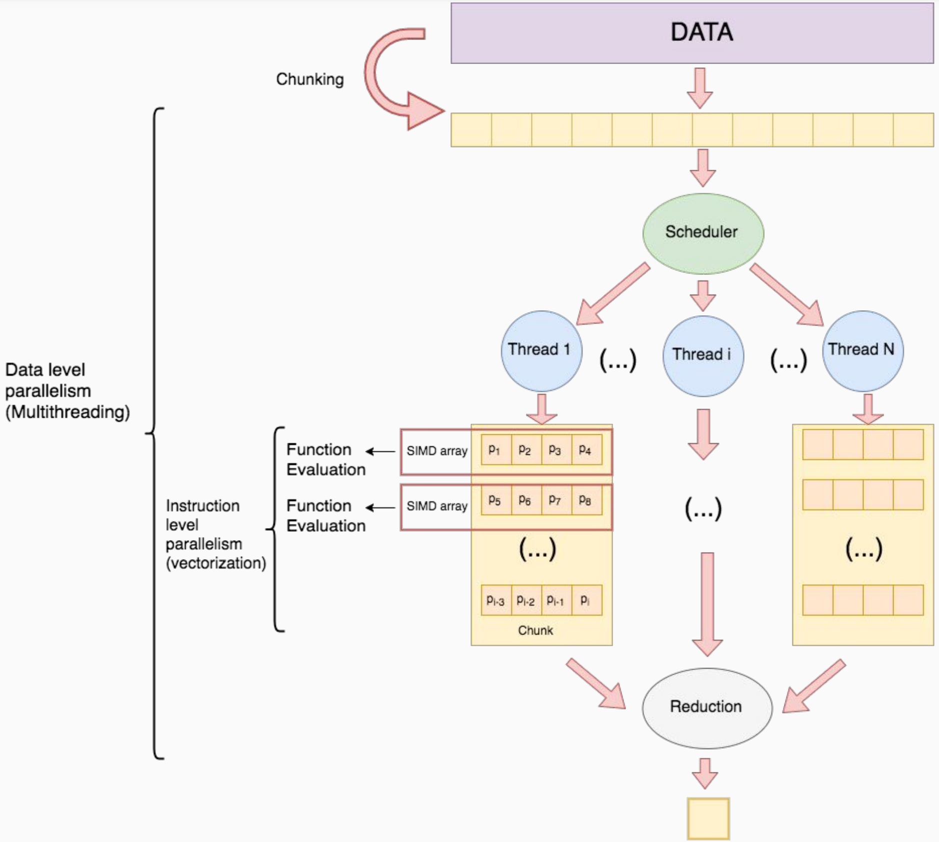


Performance: TNUMAExecutor



Parallelization of the fitting at task-level and data-level

Fitting


DATA
Chunking
Scheduler
Thread 1
(...)
Thread i
(...)
Thread N
Data level
parallelism
(Multithreading)
Function
Evaluation
SIMD array
p1 p2 p3 p4
Function
Evaluation
SIMD array
p5 p6 p7 p8
Instruction
level
parallelism
(vectorization)
(...)
(...)
(...)
pi-3 pi-2 pi-1 pi
Chunk
Reduction

Adapting the fitting classes

- Templating ROOT fitting interfaces to accept the new ROOT SIMD types

- Adapting the data classes for safe vectorization:
  - From AoS to SoA
  - Padding mechanisms

- Guaranteeing thread-safety

- Refactoring of the event loop into event evaluation function

Small changes needed

Scalar

```
//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);
```

Small changes needed

Vectorized
+
Parallelized

```
//Example Fit: Implementation of the vectorized function
ROOT::Double_v func(const ROOT::Double_v *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))));
}

// Enable implicit parallelization
ROOT::EnableImplicitMT();

//This code is totally backwards compatible
auto f = TF1("fvCore", func, 100, 200, 4);
f.SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
h1f.FillRandom("fvCore", 1000000);
h1f.Fit(&f);
```

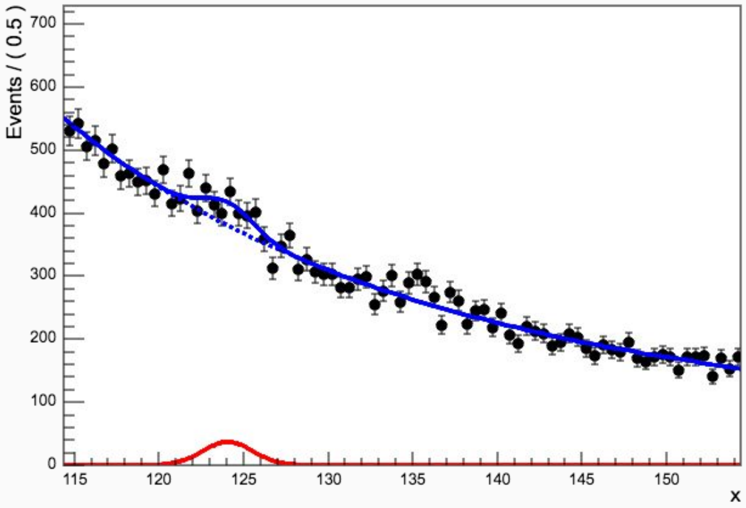
Minimal changes in progamming model

Vectorized
+
Parallelized

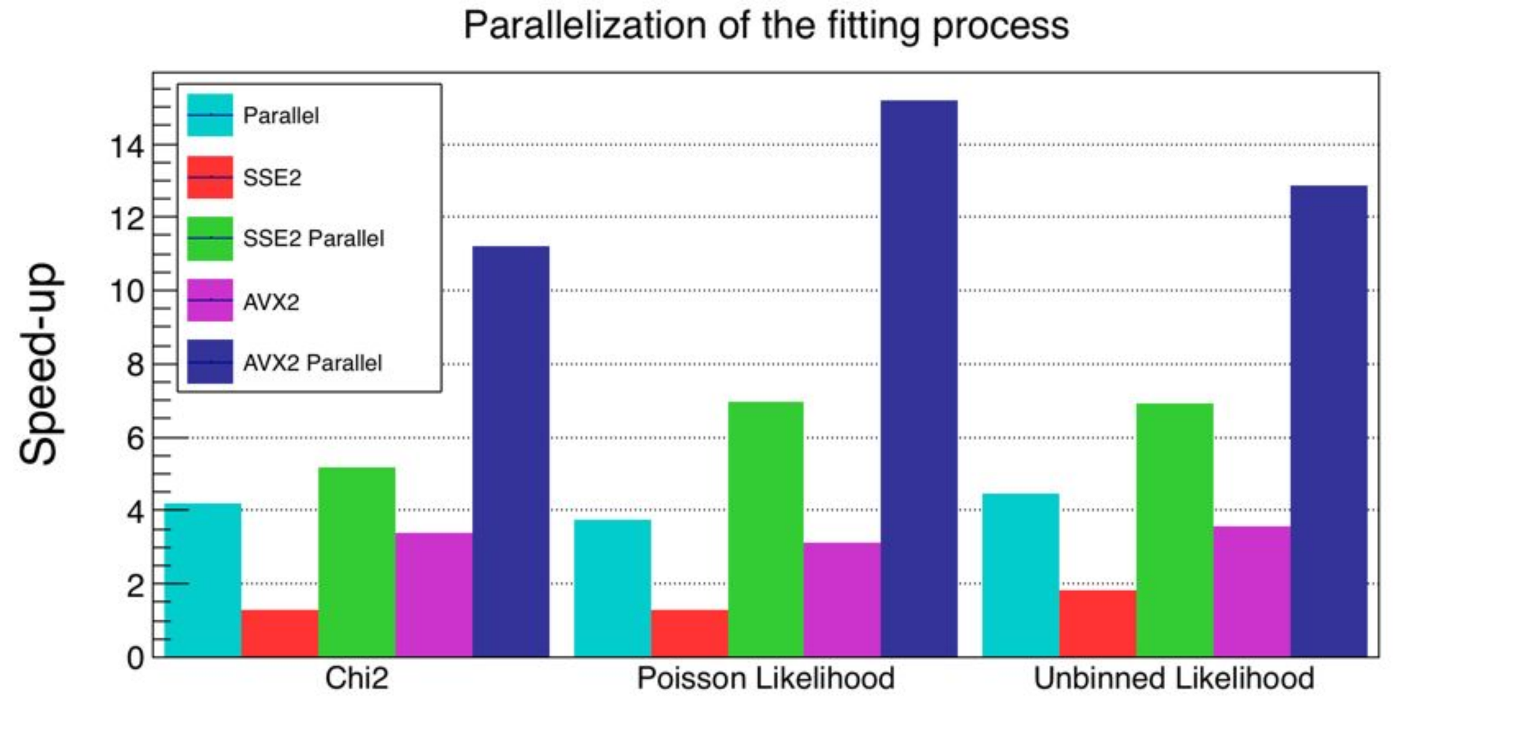
```
//Example Fit: Implementation of the vectorized function
ROOT::Double_v func(const ROOT::Double_v *data, const double *params)
{
   return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
          params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
          ((*data) * (0.01))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

// Enable implicit parallelization
ROOT::EnableImplicitMT();

//This code is totally backwards compatible
auto f = TF1("fvCore", func, 100, 200, 4);
f.SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f("h1f", "Test random numbers", 12800, 100, 200);
h1f.FillRandom("fvCore", 1000000);
h1f.Fit(&f);
```

Performance

fit of the diphoton invariant mass distribution resulting from a Higgs boson



Performance



Xavier Valls, ROOT workshop 2018

Performance

Objective function/Model	Time (s)	Speed up
χ^2/Sequential	12.11	1
χ^2/Parallel	3.20	4.19
χ^2/SSE	8.54	1.29
χ^2/SSE2 parallel	4.74	5.18
χ^2/AVX2	4.18	3.41
χ^2/AVX2 parallel	1.60	11.23
Poisson Likelihood/Sequential	16.05	1
Poisson Likelihood/Parallel	4.24	3.74
Poisson Likelihood/SSE	15.25	1.3
Poisson Likelihood/SSE2 parallel	2.33	6.95
Poisson Likelihood/AVX2	6.57	3.11
Poisson Likelihood/AVX2 parallel	1.40	15.18
Unbinned Likelihood/Sequential	3.09	1
Unbinned Likelihood/Parallel	0.82	4.48
Unbinned Likelihood/SSE	1.69	1.82
Unbinned Likelihood/SSE2 parallel	0.37	6.90
Unbinned Likelihood/AVX2	0.86	3.59
Unbinned Likelihood/AVX2 parallel	0.23	12.86

Times and speed up of the fit (4-cores + 4 hyperthreading Broadwell, 8GB RAM) for different pairs of objective function and execution policy. Evaluated over 120k bins in the binned fits and 120k points in the unbinned fit.

Wrapping up

- Check the executors! They are really convenient, flexible and provide great performance gains.
- Vectorization support is improving but you can already try it.
- See you! It's been a pleasure!