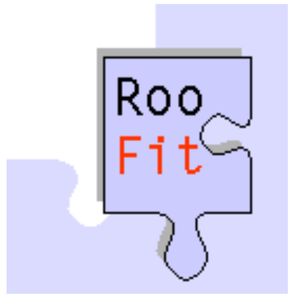
netherlands eScience center



Making fitting in RooFit faster

Automated Parallel Computation of Collaborative Statistical Models

Patrick Bos

Sarajevo, 10 Sep 2018

Physics: Wouter **Verkerke** (PI), Vince **Croft**, Carsten **Burgard**

eScience: Patrick **Bos** (yours truly), Inti **Pelupessy**, Jisk **Attema**






Nikhef

ATLAS EXPERIMENT

diana hep

netherlands eScience center

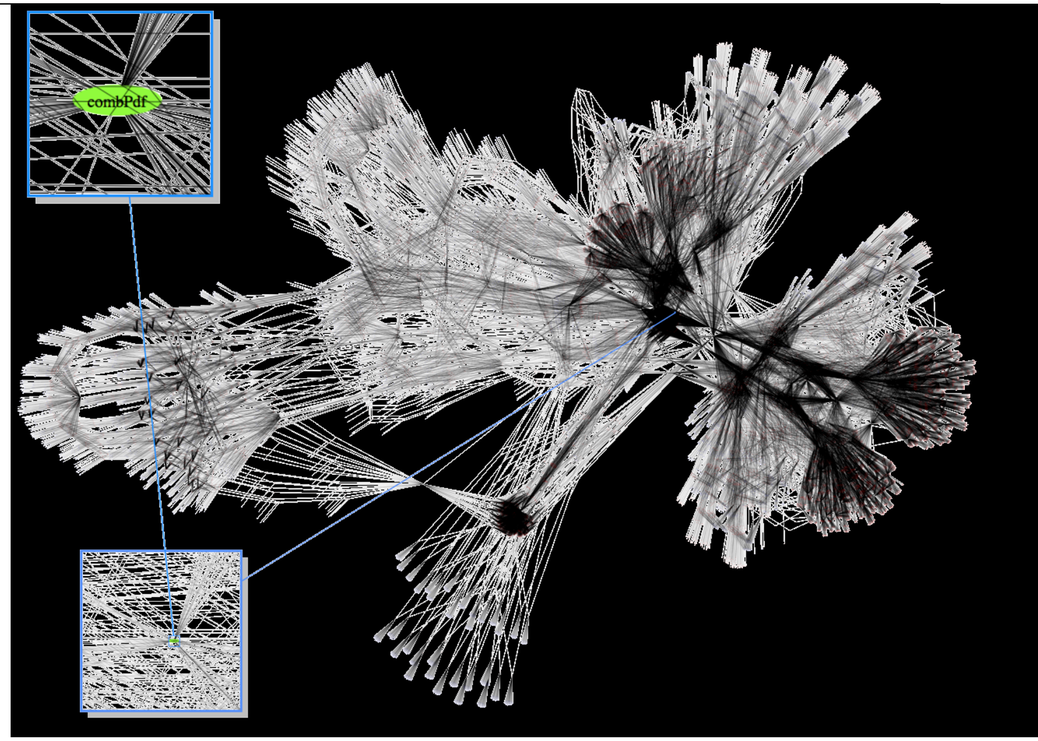
RooFit: Collaborative Statistical Modeling

Collaborative Statistical Modeling

- RooFit: build models together
  - Teams 10-100 physicists
  - Collaborations ~3000
    → ~100 teams
  - 1 goal
  - Pretty impressive to an outsider

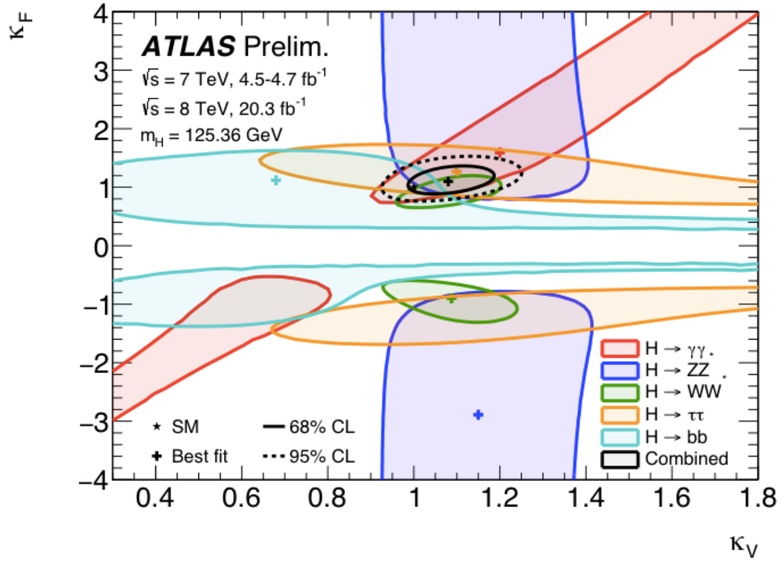
Collaborative Statistical Modeling with RooFit

Higgs @ ATLAS
20k+ nodes, 125k hours
Expression tree of C++ objects for mathematical components (variables, operators, functions, integrals, datasets, etc.)
Couple with data, event "observables"



Making RooFit faster (~30x; ~h → ~m)

- More **efficient collaboration**
  - Faster iteration/debugging
  - Faster feedback between teams
- Next level physics modeling ambitions, retaining **interactive workflow**
  1. Complex likelihood models, e.g.
     a) Higgs fit to all channels, ~200 datasets, O(1000) parameter, now O(few) hours
     b) EFT framework: again 10-100x more expensive
  2. Unbinned ML fits with very large data samples
  3. Unbinned ML fits with MC-style numeric integrals



Goals and Design: Make fitting in RooFit faster

Making fitting in RooFit faster: how?

Serial:

- benchmarks show no obvious bottlenecks
- RooFit already highly optimized (pre-calculation/memoization, MPFE)

Parallel

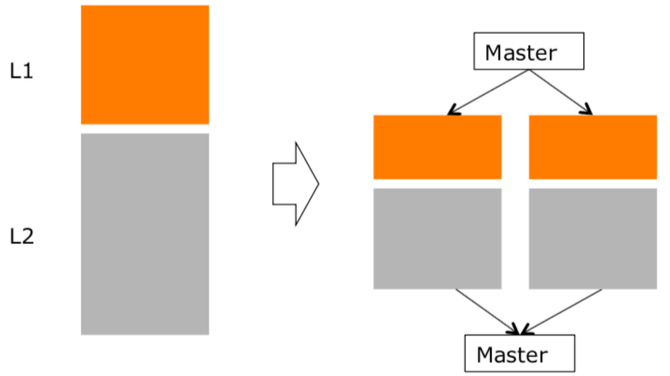
Faster fitting: (how) can we do it?

Levels of parallelism

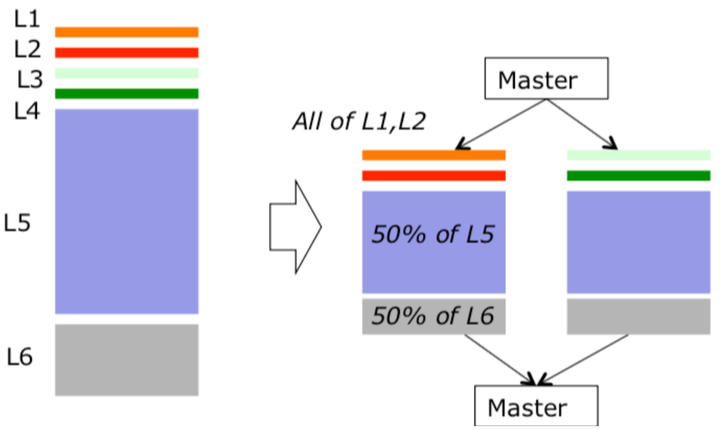
"Vector"

1. Gradient (parameter partial derivatives) in minimizer
2. Likelihood
3. Integrals (normalization) & other expensive shared components

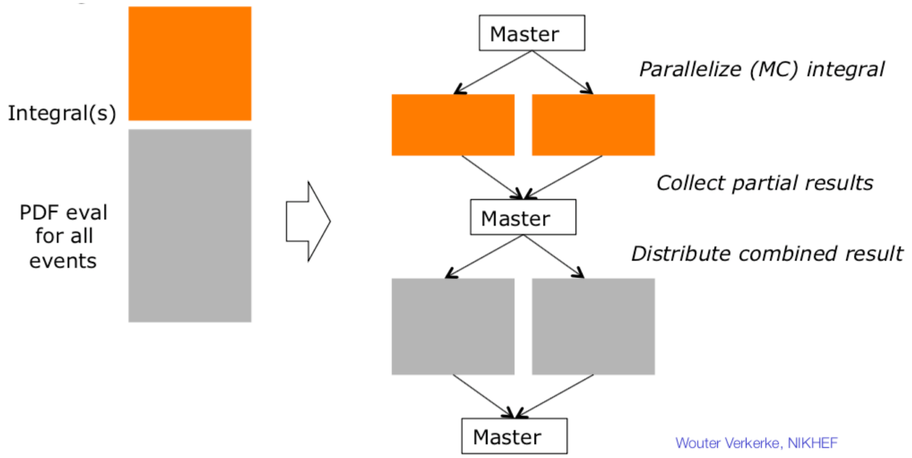
likelihood: events



likelihood: (unequal) components



integrals etc.

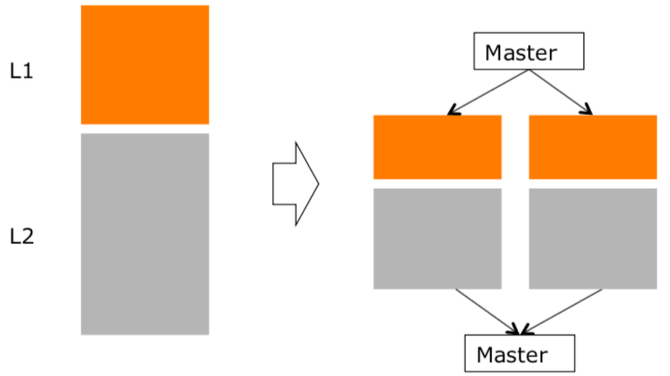


Faster fitting: (how) can we do it?

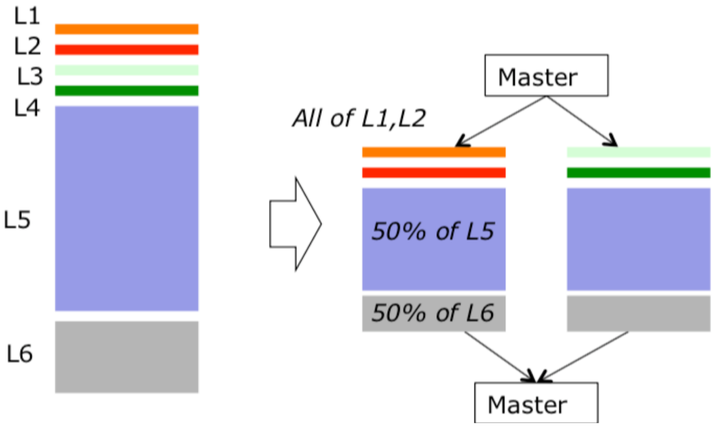
Heterogeneous: sizes, types

- Multiple strategies
- How to split up?
  - Small components → **need low latency/overhead**
  - Large components as well…
- How to divide over cores?
  - Load balancing → **task-based approach: work stealing**

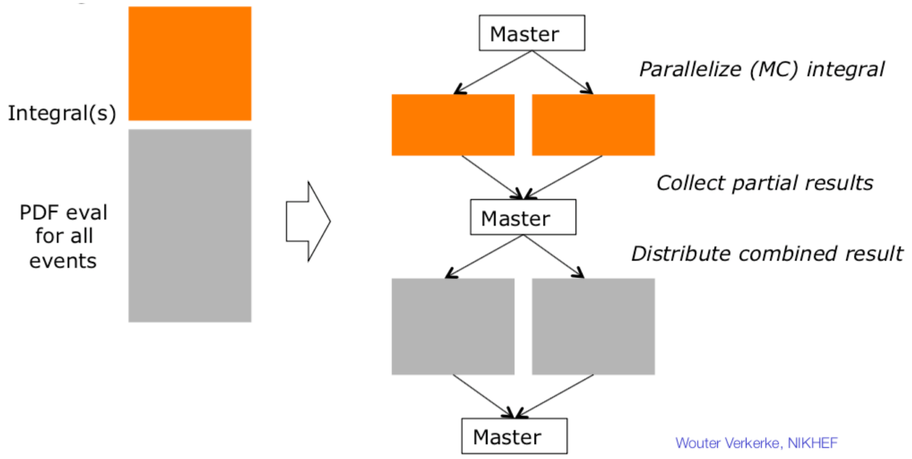
likelihood: events



likelihood: (unequal) components



integrals etc.

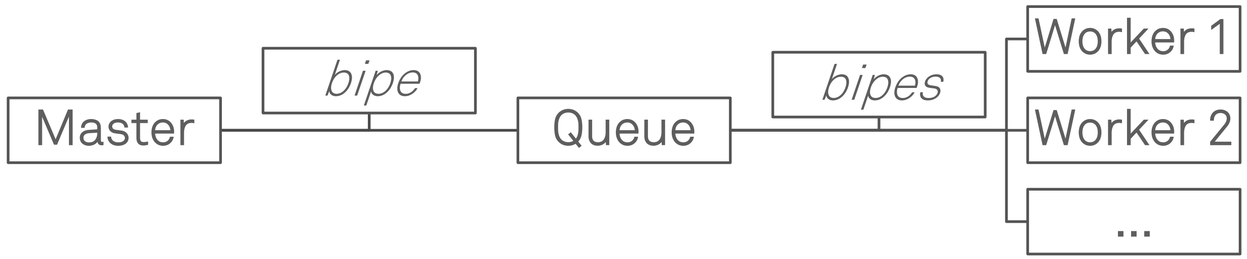


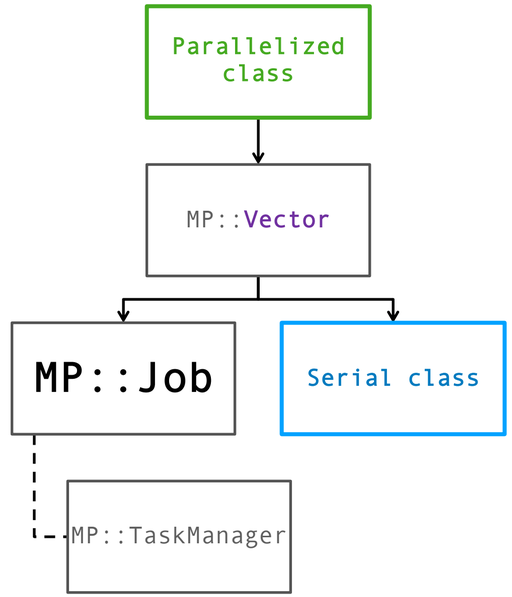
Design: MultiProcess task-stealing framework

Task-stealing, worker pool, executes Job tasks

No threads, process-based: “bipe”

(`BidirMMapPipe`) handles fork, mmap, pipes





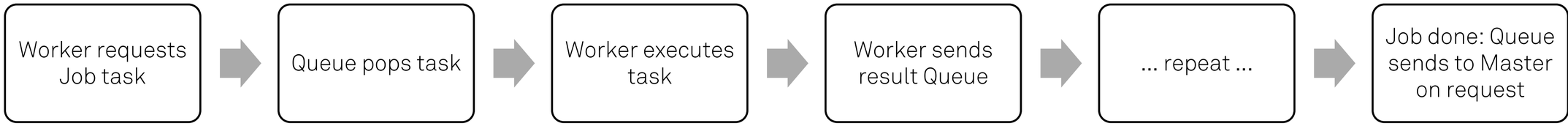
likelihood, gradient..

```
template <class T> class MP::Vector :
    public T, public MP::Job
```

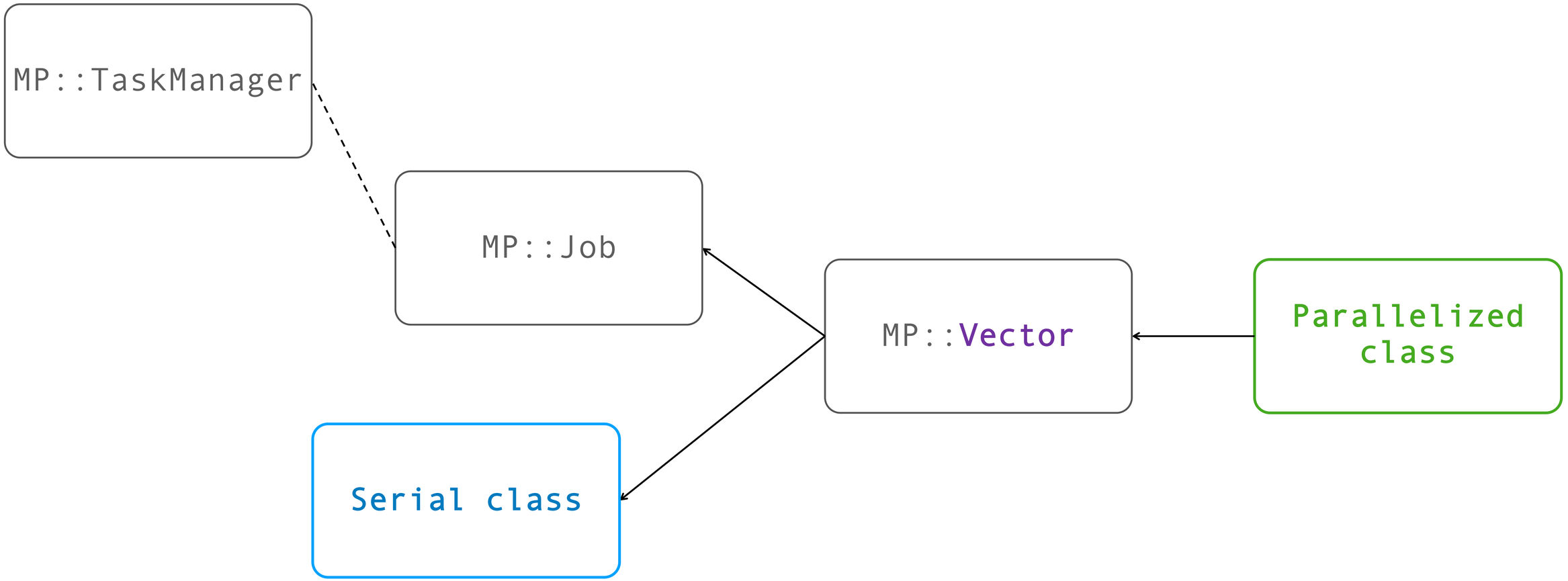
Master: main RooFit process, submits Jobs to queue, waits for results (or does other things in between)

queue loop: act on input from Master or Workers (mainly to avoid loop in Master / user code)

worker loop:



MultiProcess usage for devs




```
template <class T> class MP::Vector : public T, public MP::Job
        class Parallel : public MP:Vector<Serial>
```

MultiProcess usage for devs

```
class xSquaredSerial {
 public:
  xSquaredSerial(vector<double> x_init)
          : x(move(x_init))
          , result(x.size()) {}

  virtual void evaluate() {
   for (size_t ix = 0; ix < x.size(); ++ix) {
     x_squared[ix] = x[ix] * x[ix];
   }
  }

  vector<double> get_result() {
    evaluate();
    return x_squared;
  }

 protected:
  vector<double> x;
  vector<double> x_squared;
};
```

```
class xSquaredParallel
  : public RooFit::MultiProcess::Vector<xSquaredSerial> {
 public:
  xSquaredParallel(size_t N_workers, vector<double> x_init) :
      RooFit::MultiProcess::Vector<xSquaredSerial>(N_workers, x_init)
  {}

 private:
  void evaluate_task(size_t task) override {
    result[task] = x[task] * x[task];
  }

public:
 void evaluate() override {
   if (get_manager()->is_master()) {
     // do necessary synchronization before work_mode

     // enable work mode: workers will start stealing work from queue
     get_manager()->set_work_mode(true);

     // master fills queue with tasks
     for (size_t task_id = 0; task_id < x.size(); ++task_id) {
       get_manager()->to_queue(JobTask(id, task_id));
     }

     // wait for task results back from workers to master
     gather_worker_results();

     // end work mode
     get_manager()->set_work_mode(false);

     // put gathered results in desired container (same as used in serial class)
     for (size_t task_id = 0; task_id < x.size(); ++task_id) {
       x_squared[task_id] = results[task_id];
     }
   }
  }
};
```

```
template <class T> class MP::Vector : public T, public MP::Job
```

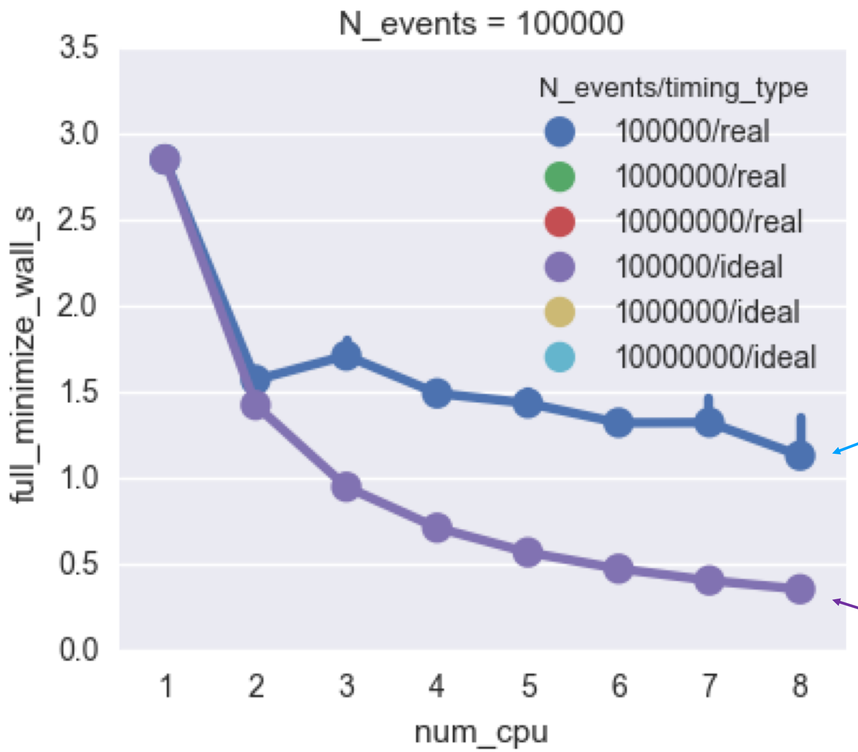
```
vector<double> x {1, 4, 5, 6.48074};

xSquaredSerial xsq_serial(x);

size_t N_workers = 4;
xSquaredParallel xsq_parallel(N_workers, x);

// get the same results, but now faster:
xsq_serial.get_result();
xsq_parallel.get_result();

// use parallelized version in your existing functions
void some_function(xSquaredSerial* xsq);

some_function(&xsq_parallel);  // no problem!
```

Parallel performance (MPFE & MP)

Likelihood fits (unbinned, binned)
Numerical integrals
Gradients

Run-time vs N(cores)

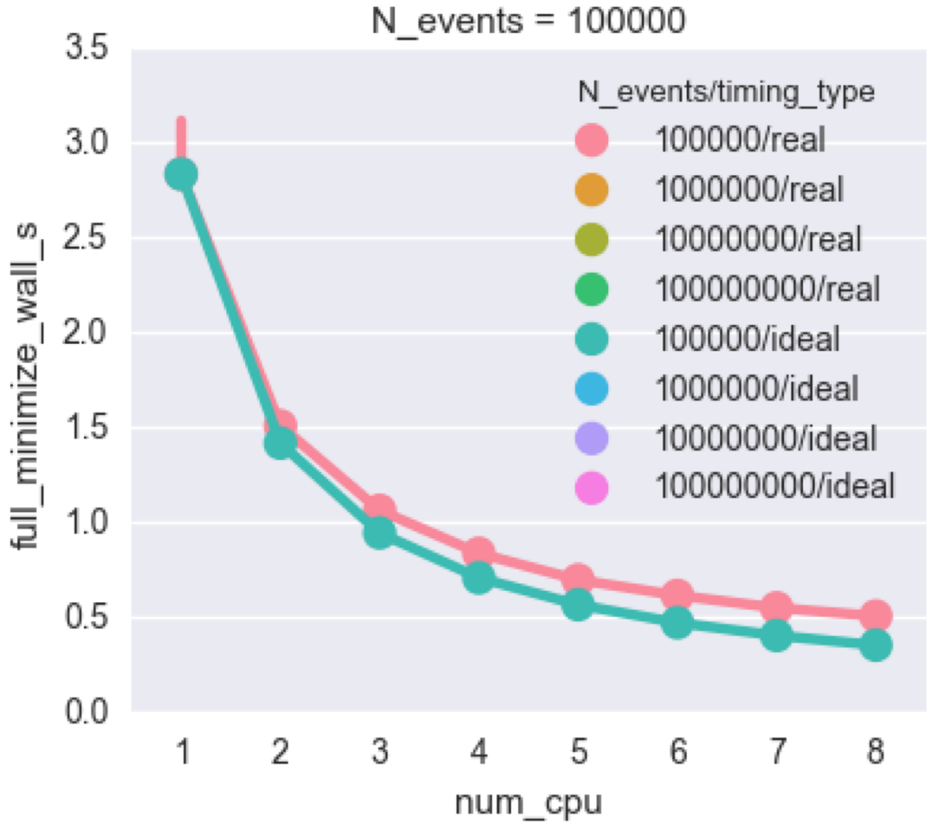
Before: max ~2x



Actual performance

Expected performance (ideal parallelization)

Now (with CPU affinity fixed): max ~20x (more for larger fits)

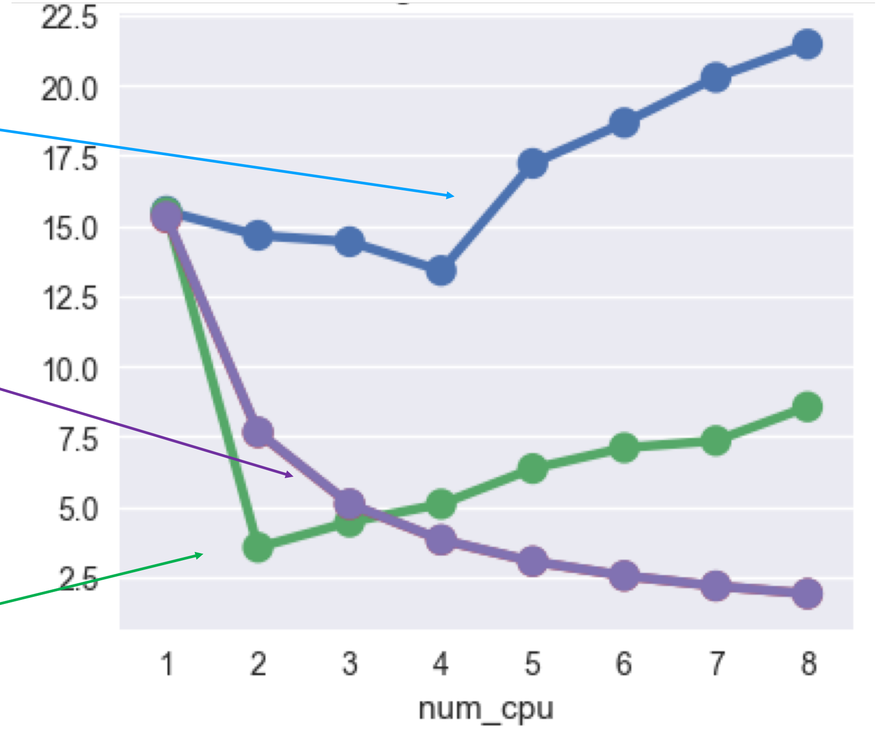


Run-time vs N(cores) in *binned fits*

Actual performance

Expected performance (ideal parallelization)

CPU time (single core)



Room for improvement

WIP

- ✓ 0th step: get Minuit to use external derivative
- ✓ 1st step: replicate Minuit2 behavior
  - NumericalDerivator (Lorenzo)
    - Modified to *exactly (floating point bit-wise)* replicate Minuit2
  - → RooGradMinimizer
- ✓ 2nd step: calculate partial derivative for each parameter in parallel

First benchmarks (yesterday):

- ggF workspace (*Carsten*), migrad fit
- scaling not perfect and erratic (+/- 5s)
  - similar as we saw for likelihoods without CPU pinning
  - probably due to too much synchronization

RooMinimizer	MultiProcess GradMinimizer					
-	1 worker	2 workers	3 workers	4 workers	6 workers	8 workers
28s	33s	20s	15s	14s	17s (...)	11s

Let’s stay in touch

+31 (0)6 10 79 58 74

p.bos@esciencecenter.nl

www.esciencecenter.nl

egpbos

linkedin.com/in/egpbos

blog.esciencecenter.nl

netherlands eScience center

Encore

Load balancing

PDF timings change dynamically due to RooFit precalculation strategies

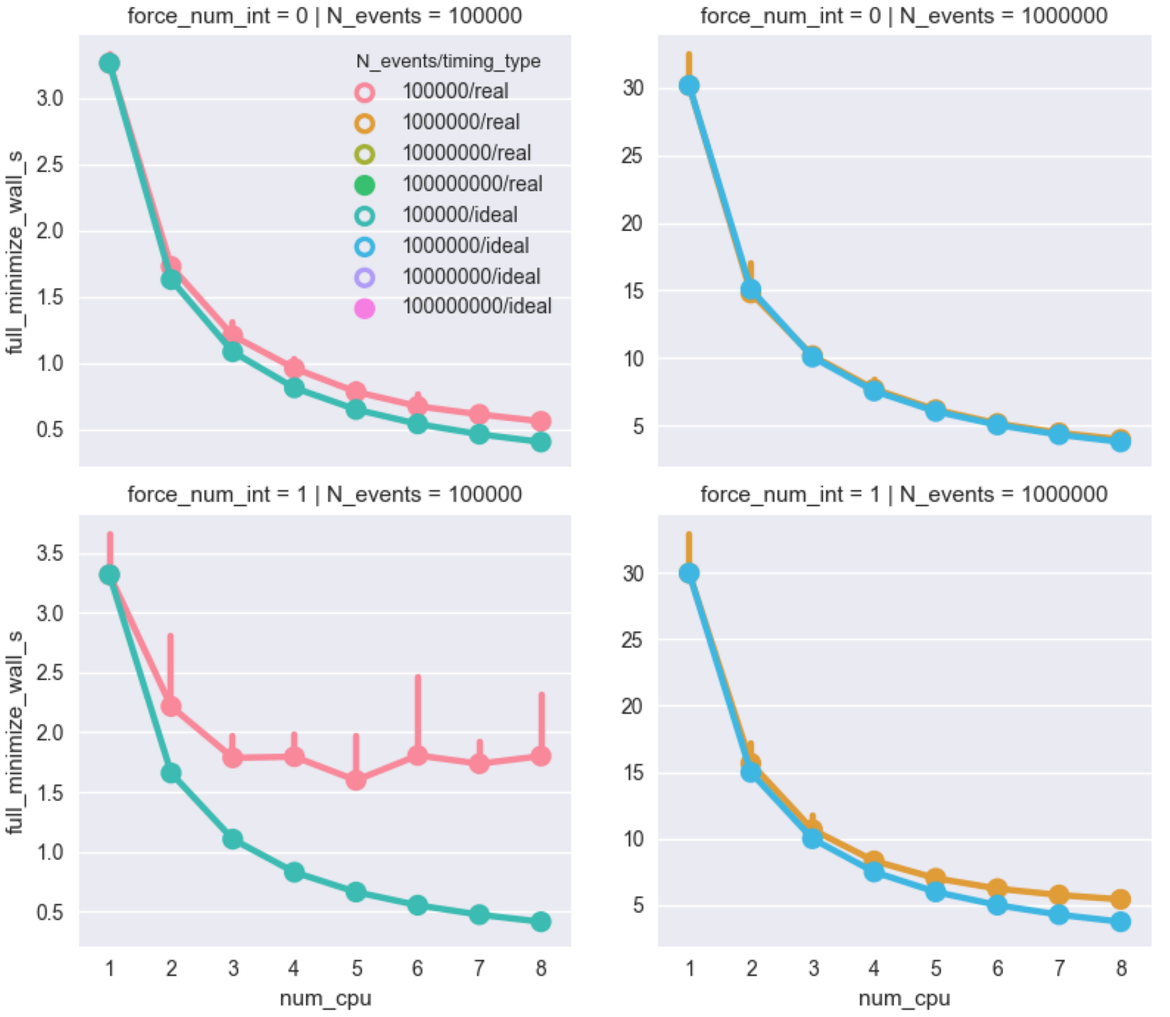
… not a problem for numerical integrals

Analytical derivatives (automated? `CLAD`)

Numerical integrals

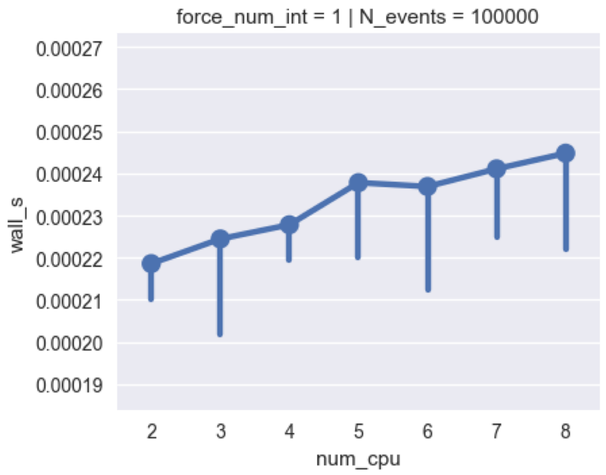
"Analytical" integrals

Forced numerical (Monte Carlo) integrals
(Higgs fits didn't have them)

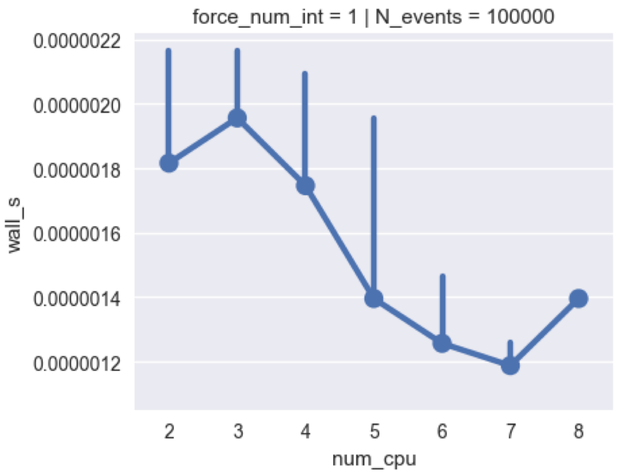


Individual NI timings (variation in runs and iterations)

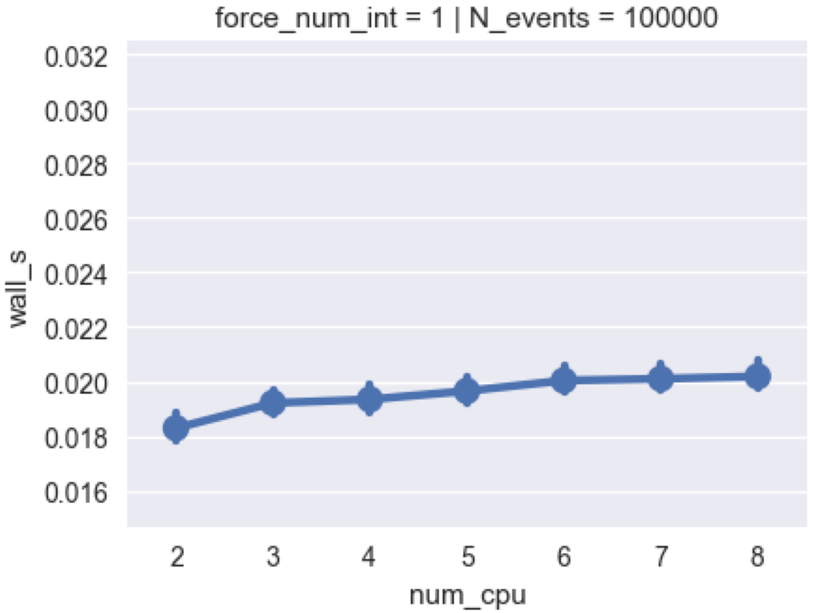
Maxima



Minima



Sum of slowest integrals/cores per iteration over the entire run



(single core total runtime: 3.2s)

`RooFit::MultiProcess::Vector<YourSerialClass>`

Serial class: likelihood (e.g. `RooNLLVar`) or gradient (Minuit)

Interface: subclass + MP

Define ”vector elements”

Group elements into tasks (to be executed in parallel)

`RooFit::MultiProcess::SharedArg<T>`

`RooFit::MultiProcess::TaskManager`

`RooFit::MultiProcess::Vector<YourSerialClass>`

`RooFit::MultiProcess::SharedArg<T>`

Normalization integrals or other shared expensive objects

Parallel task definition specific to type of object

... design in progress

`RooFit::MultiProcess::TaskManager`

`RooFit::MultiProcess::Vector<YourSerialClass>`

`RooFit::MultiProcess::SharedArg<T>`

`RooFit::MultiProcess::TaskManager`

Queue gathers tasks and communicates with worker pool

Workers steal tasks from queue

Worker pool: forked processes (`BidirMMapPipe`)

- performant and already used in RooFit
- no thread-safety concerns
- instead: communication concerns
- … flexible design, implementation can be replaced (e.g. TBB)

Single core profiling and improvements

Higgs ggf & 9 channel fits (workspaces by Lydia Brenner)

Most time spent on:

1. Memory access → RooVectorDataStore::get() (4% / 32%), 0.3% LL cache misses (expensive!)
   - Row-wise access pattern on column-wise data store (and std::vector<std::vector>)
2. Logarithms: 12%
3. Interpolation → RooStats::HistFactory::FlexibleInterpVar (10%)

Faster fitting: single core improvements

RooLinkedList::findArg: ~ 5% of memory access instructions

RooLinkedList::At took considerable time in Gaussian test fit (*Vince*)

std::vector lookup → 1.6x speedup! WIP

Faster fitting: future work

Reorder tree evaluation → CPU cache use, vectorization

Smarter fitting (stochastic minimizer, analytical gradient, CLAD)

Front-end / back-end separation (e.g. TensorFlow back-end)

Faster fitting: single core profiling meta-conclusions

profiling functions & classes

- valgrind
- gprof
- Instruments
- … etc.

profiling objects (e.g. call-trees, e.g. RooFit…)

- … DIY?

More Multi-Core

`RooRealMPFE` / `BidirMMapPipe`

Custom multi-process message passing protocol

- POSIX `fork`, `pipe`, `mmap`

Communication “overhead” (delay between sending and receiving messages): ~ 1e-4 seconds

- `serverLoop` waits for message & runs server-side code
- messages used sparingly
- data transfer over memory-mapped pipes

	RooFit (MINUIT)	TensorFlow (BFGS)
Unbinned fit	0.1s	**0.01 - 0.1s** (dep. on precision)
Binned fit	**0.7ms**	2.3ms

Fits on identical model & data (single i7 machine)

TensorFlow: No pre-calculation / caching!

Major advantage of RooFit for binned fits (e.g. morphing histograms)

(feature request for memoization https://github.com/tensorflow/tensorflow/issues/5323)

N.B.: measured before CPU affinity fixing

RooFit now even faster (but limited to running one machine)