Faster RooFitting
Automated Parallel Computation of Collaborative Statistical Models

Carsten D. Burgard

Patrick Bos, Stephan Hageböck, Vince Croft, Wouter Verkerke, Inti Pelupessy, Jisk Attema

4th of November
CHEP 2019
Adelaide
RooFit is the statistical modeling & fitting package of ROOT used for statistical inference in many experiments. Extract parameters via Likelihood maximization. Separate model building & fitting from the technical implementation and optimization.

**Parallelization**
- CPU time dominated by repeated calculations of $-\log L$
- RooFit internally parallelizes fitting
- User interface via NumCPU(..) argument
Collaborating & combining

- central container object RooWorkspace
  - store models (p.d.f. modeling the data)
  - store datasets (binned or unbinned)
  - made persistent via ROOT streamer
- measurements can be easily combined by combining workspaces (both p.d.f and data)
RooFit: Existing Optimizations

- constant-term optimization/caching
- likelihood parallelization by event/bin

Faster RooFitting

- further parallelization (this project)
- vectorization of calculations (Stephan Hageböck)
Challenges

- Many components too small to efficiently distribute over many cores ⇒ Relatively large overhead
- Other components very large (binned regions), not splitting efficiently over multiple workers

Solutions

1. Increase chunk sizes
2. Dynamic load balancing
Solution 1: Partial derivatives

- Most CPU time spent in derivative calculations inside minimizer MINUIT
  - 1 Iteration $\sim$ Gradient + line-search
  - gradient for $N$ parameters $p$: $\frac{df}{dp} \approx \frac{f(p-dp)-f(p)}{dp} \Rightarrow 2N$ calls of $f$
  - line-search: descend along gradient direction $\Rightarrow O(1)$ calls of $f$
- Focus on partial derivatives as *calculation chunk* instead of component likelihood
  - Required changes in MINUIT
  - made sure outputs stay exactly same
old custom BidirMMapPipe handles fork, mmap, pipes

new ØMQ for communication between forks

<table>
<thead>
<tr>
<th>Master</th>
<th>Queue</th>
<th>Worker 1</th>
<th>Worker 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job: $n$ tasks</td>
<td>Task 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Task 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>…</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_results</td>
<td>Result 1</td>
<td></td>
<td>Result 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>
Benchmarks for MIGRAD fit / minimization

<table>
<thead>
<tr>
<th>label</th>
<th>fast</th>
<th>big</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>ATLAS $H \rightarrow WW$ fit</td>
<td>ATLAS Higgs combination Moriond 2019</td>
</tr>
<tr>
<td>components</td>
<td>13795</td>
<td>126883</td>
</tr>
<tr>
<td>parameters</td>
<td>265</td>
<td>1487</td>
</tr>
<tr>
<td>approx. timing</td>
<td>20 s</td>
<td>10 min – few hours</td>
</tr>
<tr>
<td>comment</td>
<td>not main target audience, but used for fast benchmarking</td>
<td>depending on starting point</td>
</tr>
</tbody>
</table>
Small model
update slow, but at least constant  
- fast model not "main target"  
- mainly want to speed up runs that take hours  
- this constant part becomes insignificant  
- rest ignored for now, focusing on long runs
Big model

![Bar chart with time in seconds on Y-axis and workers on X-axis. The chart shows the time taken for gradient, line search, update, terminate, and rest tasks with error bars for each worker count from 1 to 32.](chart.png)
Big model

![Graph showing speedup with different timing types and number of workers.](image)

- **Timing Types**:
  - Gradient
  - Line search
  - Update
  - Terminate
  - Rest
  - Total

**x-axis**: Number of workers
**y-axis**: Speedup
Big model

- gradient not scaling well
  - mainly due to first partial derivative on each node taking long due to expensive precalculation

- big rest term caused by long synchronization step in serial part (master node) between roofit and minuit
- specifically the many constant terms in this model
Conclusions

▶ effective collaboration requires short wall times
  ▶ minutes rather than hours
  ▶ Parallelization can deliver this
▶ improved scaling of existing likelihood-level parallelization
▶ introduced new flexible framework
  ▶ multi-level parallelization: likelihood, gradient
▶ gradient-level parallelization scales for large (> 1 h) fits
  ▶ main goal achieved!

Related poster

Stephan Hageböck: *A faster, more accessible RooFit*
Outlook

- include new infrastructure into ROOT/RooFit
- for the adventurous: development version
  - [github.com/roofit-dev/root](https://github.com/roofit-dev/root)
- investigate imperfect scaling observed in gradient calculation
- redesign core test statistic classes for future-proof interface
- allows to plug in any new types of calculation strategy
  - e.g. analytical derivatives
- HESSE can be parallelized in similar way
- With combination of all techniques expect speedups of factor 20 for fits $> 1$ hour.
What to expect in the future

Existing ROOT classes

ROOT::Fit::Fitter
- fcn: RooMinimizerFcn
- _theFitter: ROOT::Fit::Fitter *
+ RooMinimizer(RooAbsL *)
+ RooMinimizer(RooRealL *)

ROOT::Math::IMultiGenFunction
- DoEval(const double* x) const: double
- NDim() const: unsigned int
- Clone() const: pointer to new object

ROOT::Math::IMultiGradFunction
- DoDerivative(const double *, unsigned int): double
- DoSecondDerivative(const double *, unsigned int): double
- DoStepSize(const double *, unsigned int): double

ROOT: MultiProcess
+ is_master/_worker(): bool
+ init_/term_processes()

Messenger
- zmq_sockets
+ send/receive(T)

Queue
- task_queue: std::queue<size_t>
+ add_task(size_t)
+ pop_task(): size_t
+ loop()

JobManager
- jobs: std::vector<Job *>
- process_manager_ptr
- messenger_ptr
- queue_ptr
+ activate() (start loops)

Job
- current_strategy: size_t
+ evaluate_task(size_t): void

LikelihoodJob
+ task_results: vector<pair<double>>
+ result: double

GradientJob

HessianJob

The Job strategy cannot be implemented as enum at abstract base class level, but it must be at base level to be able to implement syncing it in worker loops. In general, keep things that must be synced and take its simple.

JobManager, ProcessManager, Messenger and Queues used to be together in one big jumbled up mess called the TaskManager class (still untangling it all). The worker would be in the Job class.

In which library to put the Job implementations? They depend on the wrappers which are probably in roofitcore, but roofitcore will depend on MultiProcess, so circular dependency... Or should the wrappers also be a separate library on which roofitcore then depends?

Existing roofitcore classes

RooAbsReal
+ getValV(RooArgSet&): double

RooAbsPdf
+ createNLL(RooAbsData&, ... so circular dependency...
Or should the wrappers also be a separate
library on which roofitcore then depends?

RooAbsL
- pdf: RooAbsPdf*
- data: RooAbsData*
- optimizer: RooPDFOptimizer
+ evaluate_partition(int, int, int): double

Likelihood wrappers

LikelihoodWrapper
likelyhood: shared_ptr<RooAbsL>
+ get_value(parameters): double

LikelihoodGradientWrapper
likelihood: shared_ptr<RooAbsL>
+ get_value(parameters, size_t): double

LikelihoodHessianWrapper
likelihood: shared_ptr<RooAbsL>
+ get_value(parameters, size_t, size_t): double

Likelihood wrapper
- should optimizer have to be member? Does it have state?
- If not, just a function instead of class

Stephan: The optimization could take an AbsL, and yield another (AbsL). It’s not necessarily a function of the AbsL itself.

It’s unclear how to implement a Hessian wrapper. Minuit implements its own Hessian calculation, no callback like with FCN and GRAD.

For users that need the value of a likelihood for other reasons than minimization...