Faster RooFitting

Automated parallel calculation of collaborative statistical models

E G Patrick Bos[1,*], *Carsten D* Burgard[2,**], *Vincent A* Croft[3], *Stephan* Hageboeck[4], *Lorenzo* Moneta[4], *Inti* Pelupessy[1], *Jisk J* Attema[1], and *Wouter* Verkerke[2]

[1]*Netherlands eScience Center*, Amsterdam, Netherlands
[2]*ATLAS group*, *Nikhef*, Amsterdam, Netherlands
[3]*Dept. of Physics and Astronomy*, *Tufts University*, Medford, Massachusetts 02155, USA
[4]*ROOT Development Team*, *CERN*, Geneva, Switzerland

Abstract. RooFit [1, 2] is the main statistical modeling and fitting package used to extract physical parameters from reduced particle collision data, e.g. the Higgs boson experiments at the LHC [3, 4]. RooFit aims to separate particle physics model building and fitting (the users' goals) from their technical implementation and optimization in the back-end. In this paper, we outline our efforts to further optimize this back-end by automatically running parts of user models in parallel on multi-core machines. A major challenge is that RooFit allows users to define many different types of models, with different types of computational bottlenecks. Our automatic parallelization framework must then be flexible, while still reducing run time by at least an order of magnitude, preferably more. We have performed extensive benchmarks and identified at least three bottlenecks that will benefit from parallelization. We designed a parallelization framework that allows us to parallelize likelihood minimization with high performance by splitting over partial derivatives in the minimizer. The basis of the framework is a task queue approach. Preliminary results show speed-ups of factor 2 to 20, depending on the exact model and parallelization strategy.

1 Introduction

RooFit is a tool that is used in large collaborations of hundreds of physicists to fit large statistical models to data coming from detectors at collider experiments. Streamlining the model fitting process is crucial for increasing collaboration productivity. When a model takes only minutes to verify instead of hours, the user can remain focused on the physics of the measurement. Moreover, faster run times would allow fitting models with much more parameters to larger datasets — necessary to investigate the next generation of particle physics models, e.g. Effective Field Theory models of the Higgs boson — yielding more precise results, or even completely new findings, like models of dark matter or super-symmetry.

2 RooFit performance bottlenecks

To gauge current performance of RooFit and to identify the most promising optimization targets, we ran a benchmark on both realistic particle physics models and a set of representative

[*]e-mail: p.bos@esciencecenter.nl
[**]e-mail: c.burgard@nikhef.nl

toy models[1]. Apart from two key serial optimization opportunities, namely vectorization and memory access pattern optimization [5], no obvious further optimization target was identified without parallelization. In particular, we identified three major bottlenecks that could benefit greatly from parallelization:

1. Gradient calculation (partial derivatives with respect to the parameters) in the Minuit2 [6] `Migrad` minimizer [7];

2. Likelihood evaluation, which is a sum over PDF components evaluated for events; parallelization can happen both over events, scaling with data volumes, and over (unequal) components, scaling with model parameters;

3. Integrals (normalization) and other expensive shared components.

Which of these bottlenecks are actually relevant depends very much on the user's specific model. In some cases, parallelization of one type of "bottleneck" may lead to slower run times due to increased overhead. This calls for the implementation of multiple strategies that can be activated or deactivated depending on the model at hand.

In this paper, we focus on our implementation of gradient-level parallelization. This strategy speeds up fits of likelihoods (or other test statistics) with a large number of parameters, as e.g. the ATLAS and CMS Higgs combination fits [3, 4, 8]. `Migrad` requires a numerical partial derivative for each fit parameter, and these partial derivatives can be calculated in parallel. In this way, we speed up the most time consuming part of the `Migrad` minimization procedure [7], the gradient step. For N parameters, this step involves $2N$ test statistic evaluations. The second most expensive item, the line-search step between gradient steps, takes only a few test statistic evaluations. Note that speeding up the test statistic would speed up both steps. However, this is much more complicated due to the wide range of possible test statistics. In contrast, given a sufficient number of model parameters (sufficient being a multiple of the number of available CPU cores), the strategy of parallelizing the gradient in the minimizer will always yield performance improvements. This is why we chose to initially focus on this strategy.

3 Parallel design

In order to support multiple strategies for the parallelization of RooFit models, we designed a generic framework, `RooFit::MultiProcess`, that we expect to be close to optimal, flexible and automatic by default. The basis of the framework is a task queue approach. For each parallelizable job, a number of sub-tasks is defined and sent to a queue process that handles bookkeeping of these tasks. A pool of workers subsequently requests tasks from the queue process. Each worker only gets one task at a time and returns the result to the queue when it has completed. Then the worker will request a new task, until the queue runs out of tasks. This system automatically balances the unequal loads that heterogeneous jobs like composite likelihood calculations or partial derivatives create. Communication between processes is done by message passing using ZeroMQ [9].

Our design of the `MultiProcess` classes dealing with task management will inevitably also impact user facing classes, in particular the existing likelihood calculation classes `RooNLLVar`, `RooAbsOptTestStatistic` and `RooAbsTestStatistic`. These classes currently contain all the logic for both optimization, parallelization and different types of likelihoods, making it hard to maintain and modify. We therefore took this opportunity to also

[1] The benchmarks can be found in our GitHub repositories at https://github.com/roofit-dev/parallel-roofit-scripts and https://github.com/roofit-dev/rootbench.

redesign these likelihood classes. New user-facing classes will represent statistical concepts: binned and unbinned likelihoods and combinations thereof. Internally, optimization will be handled in a separate class as well. Parallelization will be handled by `MultiProcess`. The entire class design and its relation to existing classes is illustrated in figure 5.

We aim to provide a smooth transition for users by ensuring that all algorithms implemented in `RooFit::MultiProcess` produce the exact bit-wise identical results as the previous algorithms. One example is the transformation to `Minuit2` internal parameters, which involves trigonometric functions that cause rounding differences. For more design details, we refer to [10].

4 Results

We next present the results of benchmarks run using our implementation of a gradient-level parallelization strategy in the new `RooFit::MultiProcess` framework. This method was benchmarked on two realistic models:

1. *Fast* model: a gluon gluon fusion Higgs boson production model on an Asimov data set [11]. This has 13795 likelihood components and 265 parameters. A fit on this model runs in about 20 seconds – our main target is to speed up longer running benchmarks, but we used this model for getting good statistics on the timing data, which inevitably varies due to external influences, like operating system or other background activity.

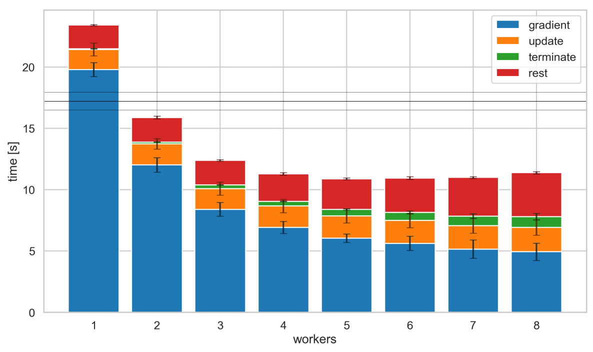
2. *Big* model: ATLAS Higgs combination fit [12]. This model has 126883 likelihood components and 1487 parameters. In a realistic scenario, where the starting point of the fitter is not close to the actual minimum, this model fits in a few hours.

We ran the benchmarks on a CentOS 7 node of the Nikhef HPC cluster. The node runs on an AMD EPYC 7551P 32-core CPU, with 256 GB RAM, which is more than sufficient for our purposes. No other users could use the node at the same time, so the impact of concurrently run programs is minimized to only processes run by the OS.

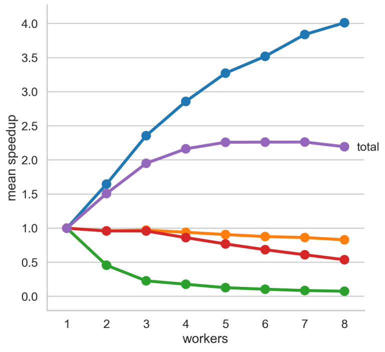
As per our design (previous section), our fit results using the new parallel framework are exactly the same as those that come out of using the serial RooFit routines. For further physics validation of the models we refer to the respective cited references.

4.1 Fast model results

The fast model fit runs in about 17 seconds with the old `RooFit::RooMinimizer` class that just runs serially in a single process, indicated by the black horizontal line in figure 1a. As figure 1 further shows, the single worker `MultiProcess` run is slower, averaging at 23 seconds. This is in part due to communication (the orange “update” component), which is not required for the `RooMinimizer`, but also clearly the gradient calculation itself was slower in our benchmarks, since it is slower than the entire minimization. The cause of the differences remains under study. A possible cause is that also the rest term (i.e. mainly the line search step) runs slightly faster in the old situation compared to the single worker situation. We found that this is largely due to the fact that RooFit caches partial results. However, these cached values are not synchronized between the workers and the master process. Since the main process does the line-search step and the workers do the gradient steps, and parameters change in between these steps, the cache is effectively thrown away each time the work load switches from the master process to the workers and the other way around. Compared to the old `RooMinimizer`, this causes a slight delay both in the master process and in the workers at the start of each step. This especially affects fast-fitting models. In fact, beyond 8 cores,



(a) Wall clock time of runs in seconds.



(b) Speed-up of the runs compared to a single worker run.

Figure 1: Fast model wall clock run times for runs with increasing number of workers on the horizontal axis. For each number of workers, the fit was repeated 10 times to get both mean run time — indicated by the height of the bars — and standard deviation — indicated by the black error bars on each histogram bar. Separately measured components of the run time are colored as indicated in the legend: *gradient* calculation time, *update* time of parameters between processes, *terminate* time at the end of a run (shutting down ZeroMQ sockets and context and the forked processes) and the rest of the run time (in these runs this includes the line-search phase). For reference, the black horizontal line at about 17 seconds indicates the mean run time of the old `RooFit::RooMinimizer` class, while the surrounding two grey lines indicate those runs' standard deviation.

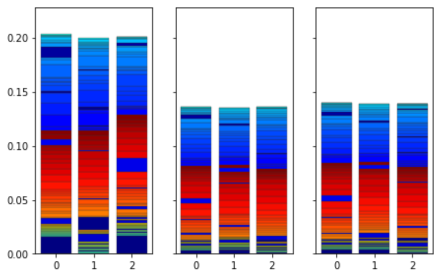
the lack of further scaling, but growth of the rest term, leads to anti-scaling, i.e. slower wall clock times with increasing number of workers. This can be seen most clearly in figure 1b, specifically in the purple line that represents the speed-up for the total run with respect to the single worker run. Despite this, a speed-up of a factor 2.5 can be achieved with 4 workers on a "fast" run like this.

4.2 Load balancing

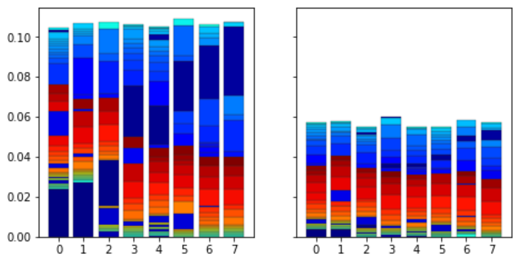
One might suspect that waiting time in-between partial derivative calculations on the workers could be a delaying factor as well, but we confirmed that this was not the case in any significant way. In addition, we investigated whether a sub-optimal load balancing of the partial derivatives over the workers could be causing the sub-optimal scaling. This analysis for a single fast model run is illustrated in figure 2. We see that for three workers (panel 2a), the load for each gradient is, in fact, very well balanced over the workers. In the case of the eight workers (panel 2b), the idle times of some workers that are waiting for the slowest worker becomes more noticeable. We measured that on average this costs about 2% of the run time with 8 workers on the big model run. All in all, we can conclude that the dynamic load balancing of our task queue approach is efficient.

4.3 Big model results

Figure 3 shows the main timing results on the big model. Due to time constraints we ran this model with initial starting parameters very close to the actual minimum, leading to only 10

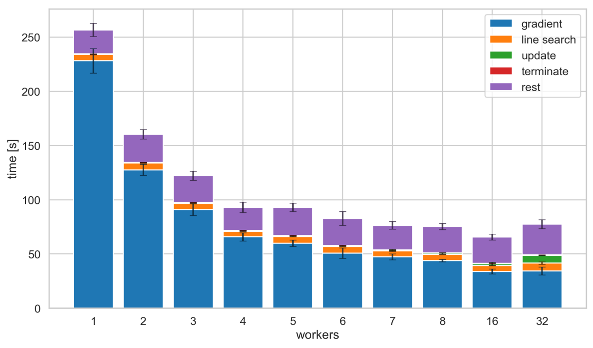


(a) Three iterations of a three worker run.

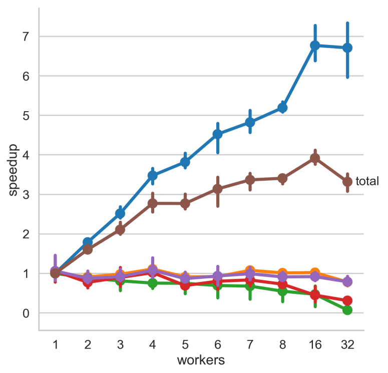


(b) Two iterations of an eight worker run.

Figure 2: Load balancing of our task queue algorithm. Each panel represents one gradient calculation. Each gradient calculation consists of 265 partial derivatives, each of which is shown as a differently colored stacked "sub"-bar. The three or eight main bars each represent work done on one of the workers used in that run. Vertical axis shows wall clock time in seconds.



(a) Timing



(b) Speed-up

Figure 3: Big model benchmark results. In this run, for each number of workers, the fit was repeated only 3 times and we additionally measured the line-search phase separately. See the caption of figure 1 for further details.

gradient steps per minimization run. We find in this case that a speed-up of a factor 4 can be achieved with 7 workers. The update and termination times seem to have become insignificant in these longer runs. The rest term, on the other hand, plays a major role in keeping the model from scaling. Further analysis revealed, however, that this component happens only once at the beginning of a minimization run. It it caused by the high number of constant terms in this model and their initial evaluation performed before the starting of the minimization run. Apart from this, the line-search step, which here we do measure independently of the rest term, turns out not to be insignificant either, although at least it remains constant, since it is calculated independently of the workers.

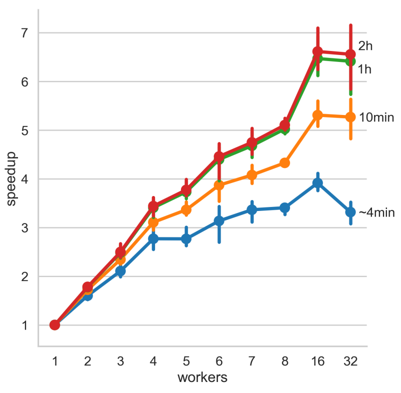


Figure 4: Big model benchmark results for performance models of longer total fitting run times.

In a typical realistic fit, many more gradient steps would be performed, since the initial guesses of the parameters will not be close to the optimal point. To see how our above results generalize to such a more realistic scenario, we also ran several times with starting guesses further from the minimum. We found that the rest and terminate components stay constant within the expected variance. Since these are only one time costs that do not scale with the number of steps (whereas the gradient, update and line search components do), we can easily construct a performance model for longer, more realistic runs. In figure 4 we show these performance model results for three run times: 10 minutes, 1 hour and 2 hours. The latter two are, in fact, the actual realistic range of single core run time using real Run 2 data [12]. We show that using 16 workers (possibly less, since we did not measure any amount of workers between 8 and 16) one can achieve an average total run time speed-up of a factor 6.5.

5 Discussion

Our parallelized gradient method achieves a factor seven speed-up on our main target of big models. The speed-up varies slightly due to communication between the processes, since we currently synchronize all parameters from and to all nodes after each run, amounting to ~ 1000 numbers being transferred between N processes for each gradient call. This is necessary because the gradient algorithm self-adapts its precision based on the minimizer's search progress. We could reduce the required communication by two orders of magnitude by pinning partial derivatives to specific workers, since the adaptive precision for each derivative component only depends on that component itself. This trade-off of flexibility in dynamic load balancing (which would be lost when pinning gradient components to specific workers) versus reduced communication could be implemented as an alternative strategy. Both strategies may prove useful in different cases.

The framework is currently available in the ROOT fork in the RooFit development GitHub page at https://github.com/roofit-dev/root/tree/MP_ZeroMQ. We warn that it should not be considered production-ready. Once ready, it will be included in an upcoming official ROOT release. The authors are in close contact with the ROOT developers team to coordinate this effort.

Acknowledgments

This project was funded by the Netherlands eScience Center. The benchmark results were analyzed and visualized using Python modules NumPy [13], Matplotlib [14], Seaborn [15] and Pandas [16] in Jupyter notebooks [17].

References

[1] W. Verkerke, D. Kirkby, ArXiv Physics e-prints (2003), `physics/0306116`

[2] L. Moneta, K. Cranmer, G. Schott, W. Verkerke, *The RooStats project*, in *Proceedings of the 13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research. February 22-27, 2010, Jaipur, India.* (2010), p. 57, `1009.1003`

[3] ATLAS and CMS Collaborations, Phys. Rev. Lett. **114**, 191803 (2015), `1503.07589`

[4] ATLAS and CMS Collaborations, JHEP **08**, 045 (2016), `1606.02266`

[5] S. Hageböck, *Making RooFit Ready for Run 3*, in *J. Phys. Conf. Ser.: ACAT 2019* (2019)

[6] M. Hatlo, F. James, P. Mato, L. Moneta, M. Winkler, A. Zsenei, IEEE Trans. Nucl. Sci. **52**, 2818 (2005)

[7] F. James, M. Roos, Computer Physics Communications **10**, 343 (1975)

[8] Tech. Rep. ATLAS-CONF-2019-005, CERN, Geneva (2019), `https://cds.cern.ch/record/2668375`

[9] P. Hintjens, *ZeroMQ: Messaging for Many Applications* (O'Reilly Media, 2013)

[10] E.G.P. Bos, I. Pelupessy, V.A. Croft, W. Verkerke, C.D. Burgard, *Automated Parallel Calculation of Collaborative Statistical Models in RooFit*, in *2018 IEEE 14th International Conference on e-Science (e-Science)* (2018), pp. 345–346

[11] M. Aaboud et al., Physics Letters B **789**, 508 (2019)

[12] ATLAS Collaboration, Tech. Rep. ATLAS-CONF-2019-005, CERN, Geneva (2019), `https://cds.cern.ch/record/2668375`

[13] S. van der Walt, S.C. Colbert, G. Varoquaux, Computing in Science Engineering **13**, 22 (2011)

[14] J.D. Hunter, Computing in Science Engineering **9**, 90 (2007)

[15] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, J. Ostblom, S. Lukauskas, D.C. Gemperline, T. Augspurger, Y. Halchenko, J.B. Cole et al., *mwaskom/seaborn: v0.9.0 (july 2018)* (2018), `https://doi.org/10.5281/zenodo.1313201`

[16] W. McKinney, *Data Structures for Statistical Computing in Python*, in *Proceedings of the 9th Python in Science Conference* (2010), pp. 51–56

[17] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay et al., *Jupyter Notebooks – a publishing format for reproducible computational workflows*, in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, edited by F. Loizides, B. Schmidt (IOS Press, 2016), pp. 87 – 90

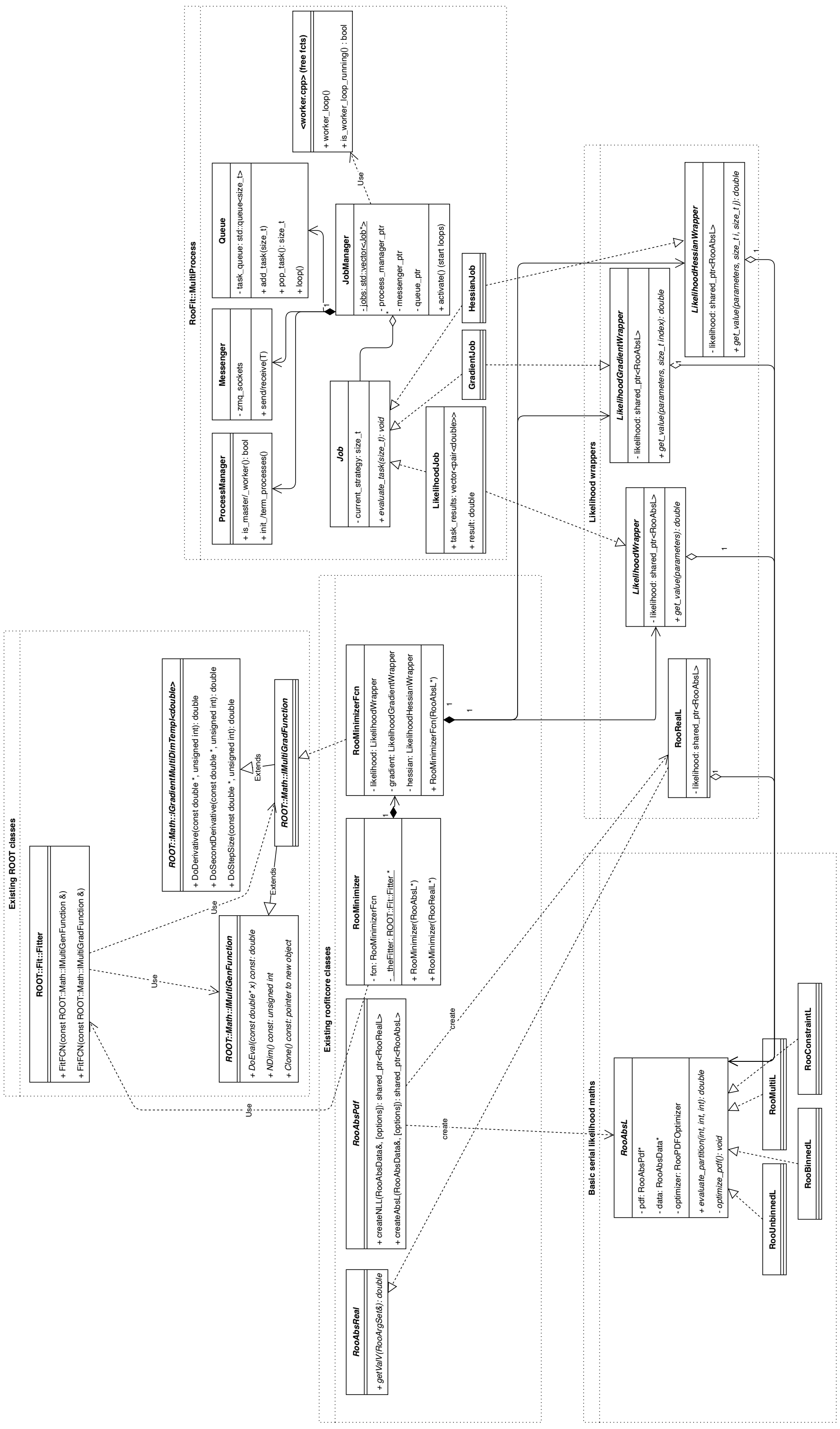

RooFit::MultiProcess
ProcessManager
+ is_master/_worker(): bool
+ init_/term_processes()
Messenger
- zmq_sockets
+ send/receive(T)
Queue
- task_queue: std::queue<size_t>
+ add_task(size_t)
+ pop_task(): size_t
+ loop()
<worker.cpp> (free fcts)
+ worker_loop()
+ is_worker_loop_running() : bool
JobManager
- 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
Use
Likelihood wrappers
LikelihoodWrapper
- likelihood: shared_ptr<RooAbsL>
+ get_value(parameters): double
LikelihoodGradientWrapper
- likelihood: shared_ptr<RooAbsL>
+ get_value(parameters, size_t index): double
LikelihoodHessianWrapper
- likelihood: shared_ptr<RooAbsL>
+ get_value(parameters, size_t i, size_t j): double
RooRealL
- likelihood: shared_ptr<RooAbsL>
Existing ROOT classes
ROOT::Fit::Fitter
+ FitFCN(const ROOT::Math::IMultiGenFunction &)
+ FitFCN(const ROOT::Math::IMultiGradFunction &)
ROOT::Math::IGradientMultiDimTempl<double>
+ DoDerivative(const double *, unsigned int): double
+ DoSecondDerivative(const double *, unsigned int): double
+ DoStepSize(const double *, unsigned int): double
ROOT::Math::IMultiGradFunction
ROOT::Math::IMultiGenFunction
+ DoEval(const double* x) const: double
+ NDim() const: unsigned int
+ Clone() const: pointer to new object
Extends
Existing roofitcore classes
RooMinimizerFcn
- likelihood: LikelihoodWrapper
- gradient: LikelihoodGradientWrapper
- hessian: LikelihoodHessianWrapper
+ RooMinimizerFcn(RooAbsL*)
RooMinimizer
- fcn: RooMinimizerFcn
- _theFitter: ROOT::Fit::Fitter *
+ RooMinimizer(RooAbsL*)
+ RooMinimizer(RooRealL*)
RooAbsPdf
+ createNLL(RooAbsData&, [options]): shared_ptr<RooRealL>
+ createAbsL(RooAbsData&, [options]): shared_ptr<RooAbsL>
RooAbsReal
+ getValV(RooArgSet&): double
create
Basic serial likelihood maths
RooAbsL
- pdf: RooAbsPdf*
- data: RooAbsData*
- optimizer: RooPDFOptimizer
+ evaluate_partition(int, int, int): double
- optimize_pdf(): void
RooUnbinnedL
RooBinnedL
RooMultiL
RooConstraintL
1

Figure 5: UML class diagram of the `MultiProcess` classes and the new likelihood classes and their connections to existing classes.