

MINUIT package parallelization and applications using the RooFit package

Alfio Lazzaro^{1,2} and Lorenzo Moneta¹

¹ CERN, Geneva

² Università degli Studi and INFN, Milano

E-mail: alfio.lazzaro@mi.infn.it, lorenzo.moneta@cern.ch

Abstract. The fitting procedures are based on numerical minimization of functions. The MINUIT package is the most common package used for such procedures in High Energy Physics community. The main algorithm in this package, MIGRAD, searches the minimum of a function using the gradient information. For each minimization iteration, MIGRAD requires the calculation of the derivative for each free parameter of the function to be minimized. Minimization is required for data analysis problems based on the maximum likelihood technique. The calculation of complex likelihood functions, with several free parameters, many independent variables and large data samples, can be very CPU-time consuming. Then, the minimization process requires the calculation of the likelihood functions several times for each minimization iteration. In this paper we will show how MIGRAD algorithm and the likelihood function calculation can be easily parallelized using Message Passing Interface techniques. We will present the speed-up improvements obtained in typical physics applications such as complex maximum likelihood fits using the RooFit package.

1. Introduction

In general all the methods used in data analysis are based on optimization problems. Depending on the particular method, the evaluation of a function is required, like the maximum likelihood function or the expected prediction error function, which has to be optimized as function of several free parameters to be determined [1]. In the last years many complex techniques are being used in the High Energy Physics (HEP) community, like maximum likelihood fits, neural networks, boosted decision trees [2]. These techniques have several advantages with respect to the simple *cut and count* analysis method, such as better discrimination between signal and background events, the possibility to take in account errors with a better precisions, and to consider correlations between the discriminating variables used in the analysis.

Increasing the samples of data analyses and using complex algorithms require high CPU performance. In general the requirement on CPU-time depends on:

- number of free parameters to be determined
- number of input events in the process
- complexity of the model to be evaluated

In the last years, vendors like Intel and AMD have not incremented the performance of single CPU unit as in the past, but they are working on multi-core CPU. Currently we have up to 6 cores implemented on one single chip, but this number will increase rapidly in the near future.

This fact represents a possible revolution in the development of new programs. Indeed we can parallelize the code using a shared memory paradigm (such as OpenMP) obtaining great benefits from new multi-core architectures. That requires to reformulate some algorithms generally used for HEP data analyses. Another paradigm for the parallelization of the code is based on the Message Passing Interface (MPI), where we can also spread the execution of the code over several interconnected CPUs in a cluster. These techniques of High Performance Computing (HPC) are well established in other fields, like computational chemistry and astrophysics. In HEP community there is not such a large use, but in the future it can be an elegant solution in all the cases where the data analyses will get more and more complicated.

In the work described in this paper we focus on the parallelization of maximum likelihood fitting code, focusing on the likelihood function calculation based on the RooFit package [3], and optimization of the ML function using the MINUIT package [4]. We will present the implemented algorithms and some speed-up examples. The techniques adopted for the parallelization are based on MPI. A description of a similar work with parallelization based on OpenMP can be found in ref. [5].

2. Maximum likelihood technique

In this section we briefly introduce the maximum likelihood (ML) technique. More details can be found elsewhere [6].

We consider a random variable x (or a multidimensional random vector $\hat{x} = (x_1, \dots, x_n)$) distributed with a distribution function $\mathcal{P}(x; \theta)$. We assume $\mathcal{P}(x; \theta)$ to be well known except for the parameter θ (or parameters $\hat{\theta} = (\theta_1, \dots, \theta_p)$). So, $\mathcal{P}(x; \theta)$ expression represents, after normalizing it, the hypothesized probability density function (PDF) for the x variable. Then, we suppose to perform an experiment where a measurement has been repeated N times, supplying x_1, \dots, x_N values. The maximum likelihood technique allows to estimate the parameter value from this data sample. Defining the *likelihood function* \mathcal{L} as

$$\mathcal{L}(\theta) = \prod_{i=1}^N \mathcal{P}(x_i; \theta), \quad (1)$$

to estimate the parameter value we have to maximize this function (*i. e. maximum likelihood*). We should underline that x_i are measured and the $\mathcal{P}(x; \theta)$ function is well-known, so \mathcal{L} only depends on the parameter we want to fit on the data sample.

In general we use the ML technique to discriminate between signal and background events in a data sample. In this case we want to extract the number of events belonging to the different species, *i. e.* signals or backgrounds. Considering s different species, and defining with n_j the number of events belonging to specie j and with $\mathcal{P}_j(x_i; \theta_j)$ the PDF for the specie j , the ML function becomes:

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^s n_j}}{N!} \prod_{i=1}^N \sum_{j=1}^s n_j \mathcal{P}_j(x_i; \theta_j). \quad (2)$$

In this expression we have introduced the *extended* term to take in account that the number of observations N in the sample is itself a Poisson random variable with a mean value $\sum_{j=1}^s n_j$. So this function is called *extended likelihood function*. It has to be maximized as function of the free parameters n_j and the eventual free parameters θ_j of the PDFs.

The evaluation of maximum for \mathcal{L} as function of the unknown parameters can be done in a numeric way. Usually, it is used to minimize the equivalent function $-\ln(\mathcal{L})$, the *negative log-likelihood* (NLL), or the function $\chi^2 = -2\ln(\mathcal{L})$. So the NLL to be minimized has the

form ¹:

$$NLL = \sum_{j=1}^s n_j - \sum_{i=1}^N \left(\ln \sum_{j=1}^s n_j \mathcal{P}_j(x_i; \theta_j) \right), \quad (3)$$

that is, a sum of logarithms. The evaluation of the NLL used in this paper is done in the ROOT framework using the RooFit package [7].

3. Optimization procedure: MINUIT

There are several algorithms to find the minimum of a function [1]. Among them, the most common method used in the HEP community is based on the MIGRAD algorithm inside the MINUIT package. MIGRAD performs the minimization of a function using the *variable metric* method [8]. This method involves the calculation of the derivatives of the NLL for each free parameter. Since very often we are faced with minimizing a function for which no derivatives are provided, MIGRAD is able to estimate the derivatives of the function by finite differences. Details of the implementation of the method used in MIGRAD can be found elsewhere [4]. Here we simply say that for the first derivative we can use the formula

$$\left. \frac{\partial NLL}{\partial \hat{\theta}} \right|_{\hat{\theta}} \approx \frac{NLL(\hat{\theta} + \hat{d}) - NLL(\hat{\theta} - \hat{d})}{2\hat{d}}, \quad (4)$$

where $\hat{\theta}$ is the set of free parameters and \hat{d} is a “small” displacement (step-size \hat{d}). The value for \hat{d} should be chosen as small as possible, but still large enough so that the rounding error in the computation of NLL does not become larger than the error introduced by the approximation. This approximation requires $2p$ function calls for p free parameters to estimate the first derivative. Furthermore it is required the numerical evaluation of second derivatives during the minimization procedure; since they form a symmetric $p \times p$ matrix, there are $p(p+1)/2$ independent components. Depending of the number p of free parameters, this means a considerable number of function calls (which increases as p^2). Then, of course, the whole procedure has to be repeated for each point $\hat{\theta}$ (except the numerical evaluation of second derivatives which are considered approximately constant over small regions) of the minimization procedure until the minimum is reached. Depending of the complexity of the NLL function, this can be very time-consuming. Furthermore, we should consider that in the NLL function we use PDFs that need to be normalized for each iteration of the minimization procedure, in case the integral of the PDFs depends on the free parameters. This requires to calculate the integral of the function, which is also time-consuming. Specific examples of maximum likelihood fits can require several hours [9].

4. Parallelization strategies

Summarizing what we described in the previous sections, we can distinguish three parts of the maximum likelihood fit procedure:

- (i) NLL calculation (implemented in RooFit): the likelihood function is calculated over the input events. From formula 3, this is a sum of terms over the N events (scaling as N);
- (ii) PDFs normalization (implemented in RooFit): it depends on the complexity of the function, in particular the calculation can be very slow if we do not have an analytical expression of the integral;

¹ We omit the $N!$ term in the expression.

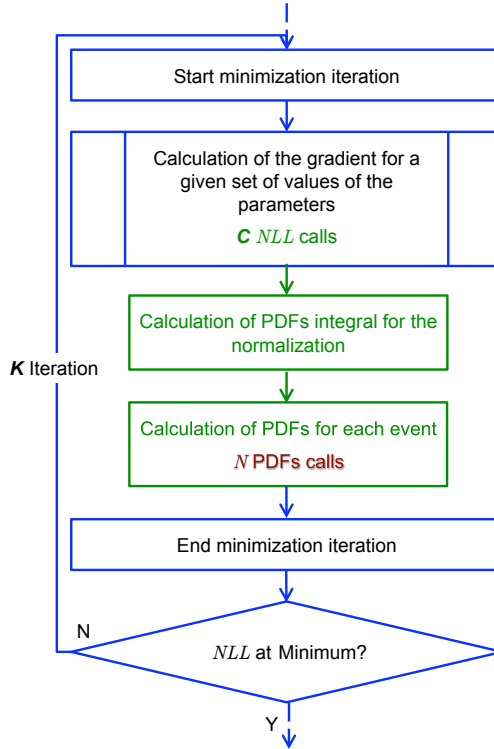


Figure 1. Schema of the minimization procedure: K is the number of iteration, C the number of NLL calls for each iteration, N the dimension of data sample to fit. See text for more details.

- (iii) minimization procedure (implemented in MIGRAD): it requires the gradient calculation for each iteration of the minimization procedure, *i. e.* the calculation of the first derivatives for several different free parameters values, which depends on the number of free parameters p to be determined (scaling as $2p$). Then it requires the calculation of the second derivatives (which increases as p^2).

A schema of the minimization iteration which links the three parts described above is shown in figure 1. Denoting with K the number of iteration, C the number of NLL calls for each iteration, N the dimension of data sample to fit, then the total number of PDFs calls without parallelization is

$$\#\text{Calls(PDFs; serial)} = K \times C \times N. \quad (5)$$

The goal of the parallelization is to reduce this number.

In principle the three parts mentioned above can be simply parallelized. The work presented in this paper concerns the points i and iii, using MPI solutions on multiple nodes of a cluster. Currently there is no implementation of parallelization of the point ii.

4.1. NLL function calculation parallelization

The parallelization of the NLL function calculation can be easily achieved splitting the sum term over the N events (see formula 3). This is already implemented in RooFit using *fork* calls. It allows a sensible reduction of the duration of maximum likelihood fits in case we run on multi-core machines, requiring one core for each thread. In this way the speed-up scales almost with a factor proportional to the number of threads. However, this parallelization used in RooFit is not

a shared-memory solution and is therefore limited by the total dimension of memory available in the multi-core machine, and then by the number of available cores in a single node.

We implemented a similar strategy for parallelization using MPI. This resolves the limitations due to the dimension of memory and the number of cores in a single node. Each MPI process does the same work, except for the calculation of the *NLL* function. In this case the sum is split in partial sums for each MPI process. At the end of the partial sums calculation, a `MPI_Allreduce` operation is done, so that each process has the result of the complete sum. This parallelization is done for each *NLL* function call. In this case the possible number of MPI processes depends on the complexity of the function and the number of events in the data sample. Normally the speed-up scales almost with a factor proportional to the number of processes in complex ML fits, before the execution time for serial parts, for communications and synchronizations become significant. The total numbers of MPI calls and PDFs calls for each process are

$$\begin{aligned}\#\text{Calls}(\text{MPI}; NLL) &= K \times C, \\ \#\text{Calls}(\text{PDFs}; NLL) &= K \times C \times \frac{N}{P},\end{aligned}\tag{6}$$

where P is the number of MPI processes.

Note that this parallelization can suffer some rounding approximation between the single sum without parallelization and the sum of the partial sums in case of parallelization.

4.2. Minimization procedure parallelization

The parallelization is done splitting the derivative calculation over several processes, balancing the number of derivatives (which are equal to the number of free parameters in the *NLL* function) for each process. This means that the maximum number of processes is equal to the number of free parameters p . For example with 10 free parameters and 3 processes, we have four derivatives for one process, and three each for the other two. Each process has a common serial initialization part. Then we have the splitting of the derivative calculation, where each process takes care of his group of derivatives. At the end of this stage, all results are scattered between the processes, using a `MPI_Allgather` operation, so that they can conclude the minimization iteration (also this part is in common between all the processes, *i. e.* each process proceeds in the same way in the minimization). This represents an iteration of the minimization procedure, which is repeated until the minimum of the function is reached. Figure 2 shows an illustrative schema of this procedure. The total numbers of MPI calls and PDFs calls for each process are

$$\begin{aligned}\#\text{Calls}(\text{MPI}; \text{Minimization}) &= K, \\ \#\text{Calls}(\text{PDFs}; \text{Minimization}) &= K \times \left(\frac{2p}{P} + C_2 \right) \times N,\end{aligned}\tag{7}$$

where C_2 is the number of *NLL* function calls not used for the first derivative calculations (with $C = 2p + C_2$). Indeed, the implemented parallelization does not regard the calculation of the second derivatives, whose number of *NLL* function calls usually represent a small fraction of total calls in complex fits. However, this fact limits the scalability of the parallelization.

It is important to note that the time spent for each iteration depends on the slowest process, *i. e.* the process with the most derivatives to calculate. This fact suggests the maximum number of processes where we can see an effective speed-up. For example, with 10 free parameters, there is at least one process with 2 parameters when we require between 5 and 9 processes in the parallelization. Of course increasing the number of processes means overhead for the synchronization and communication between the processes. Therefore a good scalability can be reached in case of larger number of free parameters than number of processes.

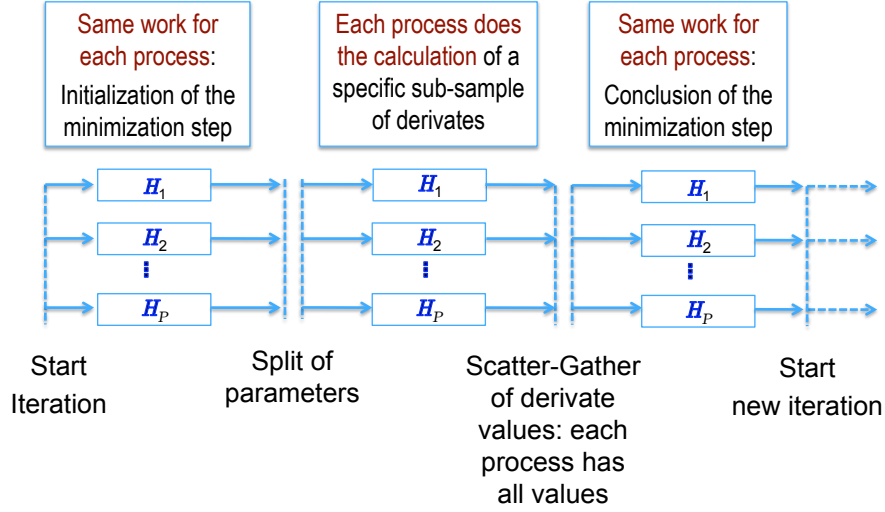


Figure 2. Schema of the parallelization of the minimization procedure on P processes (see text for details).

4.3. Hybrid parallelization

The two implementations described above give the opportunity of a third implementation, which is a hybrid of the two. Assuming $P = H_x \times H_y$ processes, we reserve the H_x processes for the minimization procedure (first derivative calculations) and H_y processes for the NLL function calculation. We declare in this way a MPI Cartesian topology communicator, *i. e.* a matrix of $H_x \times H_y$ dimension. We extract the MPI sub-communicators from the columns of the matrix (dimension is H_x) for the minimization parallelization, and from the rows (dimension is H_y) for the NLL function calculation. For the second derivatives and all other NLL function calls we parallelize the NLL function calculation using all P processes. The total number of MPI calls and PDFs calls for each process are

$$\begin{aligned} \#\text{Calls(MPI; Hybrid)} &= K \times \left(\frac{2p}{H_x} + C_2 + 1 \right), \\ \#\text{Calls(PDFs; Hybrid)} &= K \times C \times \frac{N}{P}. \end{aligned} \quad (8)$$

Comparing these numbers with those of the parallelization of NLL function (formulas 6), we can note that in case of the hybrid parallelization we have less MPI calls with the same number of PDFs calls. In other words, the hybrid solution scales as the parallelization of NLL function, but with less MPI calls.

5. Speed-up example

In the following reported example, we used MINUIT2, which is just the C++ object-oriented version of the original MINUIT [10], and RooFit implemented in ROOT v5.22. We performed all tests running at CNAF batch system cluster (INFN Computing Center, Bologna, Italy). Each node used for this work have 2 quad-core CPUs (Intel® Xeon® CPU E5420, 2.50 GHz), *i. e.* 8 cores per node, with 16 GByte/node of shared memory. The internal network for the communication is provided by Gigabit Ethernet. We ran our test in the worst condition in terms of MPI communications, reserving always one process per node (other slots per node being used for other jobs in the cluster).

All the MPI calls are declared in a specific class of the MINUIT2 package, which supplies an interface to MPI routines. We use OpenMPI v1.2.5 [11] and TAU Performance System for performance analysis [12].

The likelihood function is defined with 2 species and 3 variables, denoted as x_1, x_2, x_3 . For each variable of the specie 1 we use a double Gaussian PDF, defined as:

$$DG_t \equiv DG(x_t; \mu_t^A, \sigma_t^A, \mu_t^B, \sigma_t^B, f_t) = f_t G_A(x_t; \mu_t^A, \sigma_t^A) + (1 - f_t) G_B(x_t; \mu_t^B, \sigma_t^B), \quad (9)$$

where G_A and G_B are Gaussian PDFs, with parameters μ and σ , and f is the corresponding fraction between them. For each variable of the specie 2 we use a second order polynomial PDF, $P_t \equiv P(x_t; a_t, b_t) = a_t x_t^2 + b_t x_t + 1$ ². The total PDF for each specie is the product of the PDF of each variable:

$$\begin{aligned} \mathcal{P}_1 &= DG_1 \times DG_2 \times DG_3, \\ \mathcal{P}_2 &= P_1 \times P_2 \times P_3. \end{aligned} \quad (10)$$

So the total number of parameters to be fitted in the NLL function (formula 3) is 23. The data sample is composed by 1,050,000 events.

Running the fit without parallelization takes about 1 hour and 50 minutes (real time). We check that the fit ends with convergence status. The total number of NLL calls is 1489, where 292 of them do not regard the first derivative calculation ($C_2 = 292$). We use this case as reference for the speed-up plots.

In figure 3 we show the speed-up results for different configurations of the parallelization. We note that the case of only minimization parallelization strategy does not scale as well as when we apply the NLL function parallelization strategy. The reason, already mentioned in section 4.2, is due to the value of C_2 which is not negligible in this example with respect to the total number of NLL function calls. Same considerations can be extracted from the plots in figure 4, where we compare the speed-up plots for different parts of the fitting program.

We compare in the figure 5 the fraction of exclusive time spent for MPI calls with respect to the total execution time of the fitting program. We observe a constant increase of this fraction. Breakdown of the fractions of exclusive time spent for some parts of the fitting program are shown in figure 6. In the figure 7 we show the exclusive time for the same parts, and we put it in correlation with the total execution time. From all these plots we can conclude that the main limitation to the scalability in the considered example is due to the time spent for MPI calls. However, we should consider that the likelihood function used in this example has a simple expression (based on Gaussians and polynomials). In complex fits (like for Dalitz Plot analysis [13]) the likelihood function has more variables and more complicated expressions with more free parameters. For such cases we have more scalability before the time spent for MPI calls becomes a limitation.

6. Conclusion

The solutions adopted for parallelization of ML fitting programs give acceptable results, reducing the total execution time. The main limitation to the scalability is given by time spent for communications between MPI calls, which can be reduced using faster networks. The work presented in this paper have required changes in the RooFit and MINUIT2 packages, which will be part of the future release of ROOT. The parallelization of the minimization procedure implemented in MINUIT2 can be used in all domains where it is required such a procedure, *i. e.* not only NLL fits, hence in general in data analysis code based on MINUIT.

² We are omitting the normalization factors in all PDFs. The normalization integrals are calculated using analytical expressions.

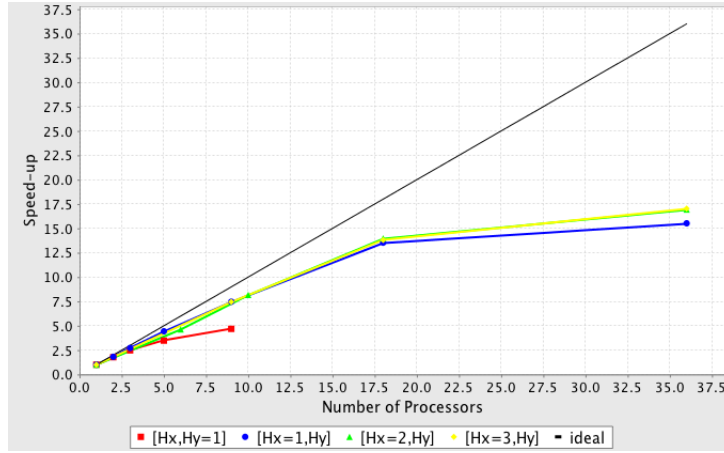


Figure 3. Speed-up plots for different parallel configurations of the the fit described in the text. The black line without marks refers to ideal speed-up. Other lines refer to our tests: red line with square marks is the case of only minimization parallelization strategy (label $[H_x, H_y=1]$); blue line with circle marks is the case of only NLL function parallelization strategy ($[H_x=1, H_y]$); other two lines are for hybrid parallelization strategy with $H_x = 2$ (green line with triangle marks, $[H_x=2, H_y]$) and $H_x = 3$ (yellow line with diamond marks, $[H_x=3, H_y]$), respectively.

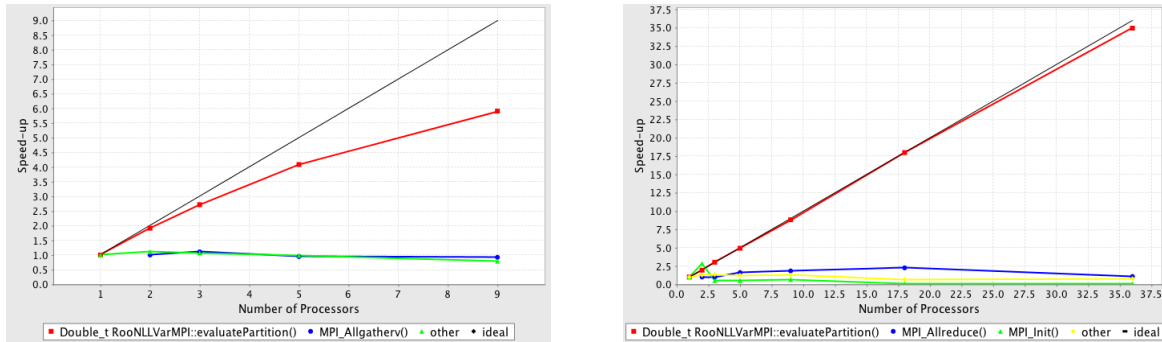


Figure 4. Speed-up plots for different parts of the fitting program. The `Double_t RooNLLVarMPI::evaluatePartition()` function does the calculation of the NLL function. On the left plot we have the case of only minimization parallelization strategy and on the right plot the NLL function parallelization strategy. Clearly we can see how the calculation of the NLL function scales better in the right plot. Similar consideration are valid for the hybrid parallelization strategy.

Acknowledgments

Authors thank CNAF support group, in particular Armando Fella, for their kind support during the tests done on his cluster. Alfio Lazzaro thanks Filippo Spiga for his suggestions about the MPI parallelization.

References

- [1] Press W H, Teukolsky S A, Vetterling W T and Flannery B P 2007 *Numerical Recipes: The Art of Scientific Computing* 3rd ed (Cambridge: Cambridge University Press)

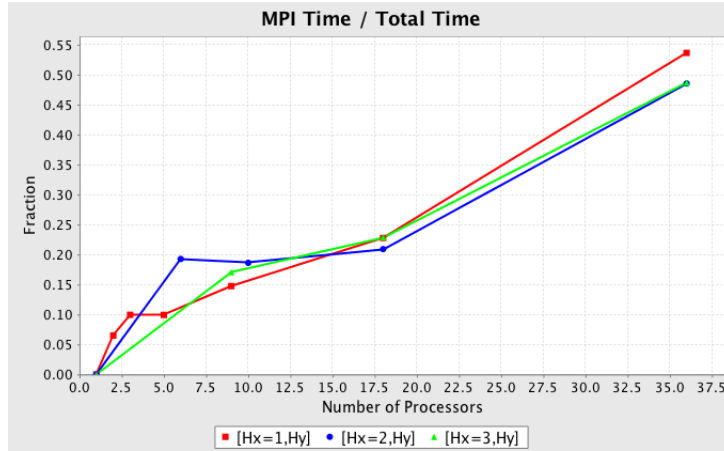


Figure 5. Fraction of time spent for MPI calls with respect to the total execution time of the fitting program. Lines refer to our tests: red line with square marks is the case of only *NLL* function parallelization strategy ($[H_x=1, H_y]$); other two lines are for hybrid parallelization strategy with $H_x = 2$ (blue line with circle marks, $[H_x=2, H_y]$) and $H_x = 3$ (green line with triangle marks, $[H_x=3, H_y]$), respectively.

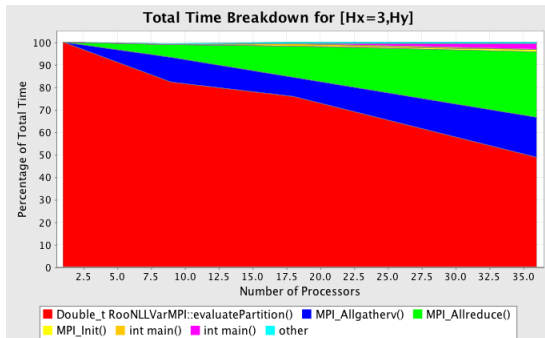


Figure 6. Percentages of exclusive time for different parts of the fitting programs with respect to total execution time. The plot refers to the test with hybrid parallelization ($H_x = 3$). The `Double_t RooNLLVarMPI::evaluatePartition()` function does the calculation of the *NLL* function.

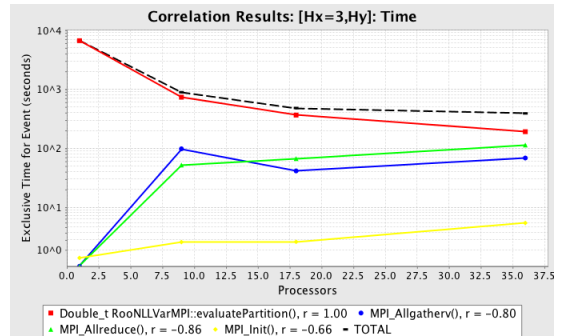


Figure 7. Exclusive time for different parts of the fitting program with respect to total execution time. The plot refers to the test with hybrid parallelization ($H_x = 3$). The `Double_t RooNLLVarMPI::evaluatePartition()` function does the calculation of the *NLL* function. In the legend we report the correlation values with respect to the total time.

- [2] Friedman J, Hastie T and Tibshirani R 2001 *The Elements of Statistical Learning* (Berlin: Springer)
- [3] Verkerke W and Kirkby D see the web page of RooFit package: URL <http://roofit.sourceforge.net>
- [4] James F 1972 *MINUIT - Function Minimization and Error Analysis* CERN Program Library Long Writeup D506
- [5] Lazzaro A and Moneta L 2009 *Proceedings of Science* PoS(ACAT08)083
- [6] Cowan G 1998 *Statistical Data Analysis* (Oxford: Clarendon Press)
- [7] See the web page of ROOT project: URL <http://root.cern.ch/>
- [8] Davidon W C 1991 *SIAM J. Optim.* **1** 1–17
- [9] Aubert B *et al.* (BABAR Collaboration) 2007 *Phys. Rev. Lett.* **98** 211802
- [10] Hatlo M *et al.* 2005 *IEEE Transactions on Nuclear Science* **52-6** 2818
- [11] See the web page of OpenMPI: URL <http://www.open-mpi.org/>
- [12] See the web page of TAU: URL <http://www.cs.uoregon.edu/research/tau/home.php>
- [13] Aubert B *et al.* (BABAR Collaboration) 2009 *Phys. Rev. D* **79** 032003