Implementing parallel algorithms for data analysis in ROOT/RooFit

Sverre Jarp, *Alfio Lazzaro*, Julien Leduc,
Yngve Sneen Lindal, Andrzej Nowak
European Organization for Nuclear Research (CERN), Geneva, Switzerland

Workshop on Future Computing in Particle Physics, e-Science Institute, Edinburgh (UK)
June 15th–17th, 2011

CERN openlab

CERN openlab

www.cern.ch/openlab

PARTNERS

invent

CERN openlab

- CERN openlab is the only large-scale structure at CERN for developing industrial R&D partnerships
  - www.cern.ch/openlab-about
- Divided in competence centers
  - HP: wireless networking
  - Intel: advanced hardware and software evaluations and integrations
  - Oracle: database and storage
  - Siemens: automating control systems

Introduction (1)

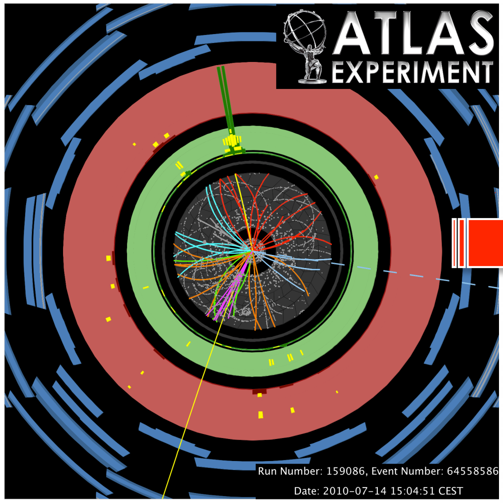
- Part of our activity is to develop new benchmarks that are representative of the computing applications used at CERN
  - Simulation, reconstruction, data analysis
  - Collaboration with the physics community
  - We use these applications for evaluating the performance of new Intel platforms, working closing with Intel experts
- In this and in next presentation we will present what we are doing for data analysis applications
  - Biased from my experience in the Babar and Atlas experiments. However, data analysis is not our goal, so we don’t focus on any specific analysis
    - Strong collaboration with physics collaborators to have wide coverage of different analyses

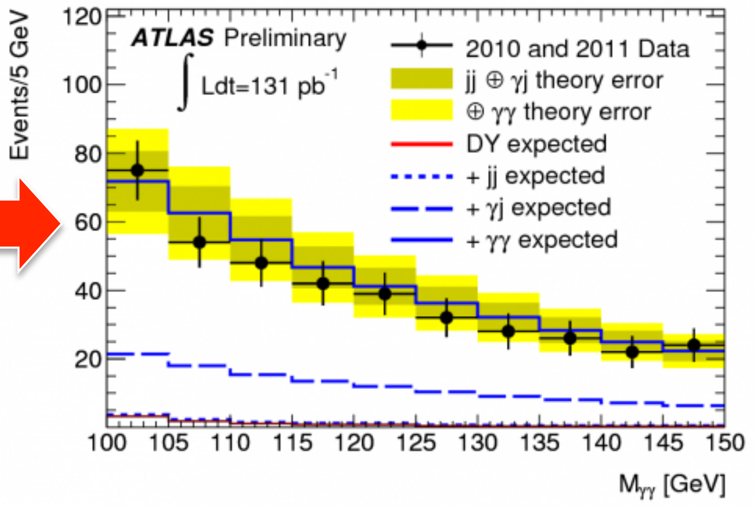
Introduction (2)

- Our way to proceed:
  - Understanding the current version of the algorithm
  - Rewriting the algorithm so that we can improve it
    - Optimizations, vectorization, numerical accuracy
  - Apply parallelization
  - Porting the algorithm on accelerators
- We will focus on the problem we have encountered and on the solutions we have adopted, rather than showing results
  - Most technical details, useful in the context of a workshop
  - In my presentation I will introduce the application and the parallelization on the CPU, while in the next presentation Yngve will show the porting to the GPU

Data analysis

- Huge quantity of data collected, but most of events are due to well-know physics processes
  - New physics effects expected in a tiny fraction of the total events: few tens
- Crucial to have a good discrimination between interesting (signal) events and the rest (background)
  - Data analysis techniques play a crucial role in this “war”


ATLAS EXPERIMENT
Run Number: 159086, Event Number: 64558586
Date: 2010-07-14 15:04:51 CEST



Likelihood-based techniques

- Data are a collection of independent events
  - an event consists of the measurement of a set of variables (energies, masses, spatial and angular variables...) recorded in a brief span of time by the physics detectors
- Introducing the concept of probability $\mathcal{P}$ (= Probability Density Function, PDF) for a given event to be signal or background, we can combine this information for all events in the *likelihood function*

$$\mathcal{L} = \prod_{i=1}^{N} \mathcal{P}(\hat{x}_i|\hat{\theta})$$

N number of events
$\hat{x}_i$ set of variables for the event i
$\hat{\theta}$ set of parameters

- Several data analysis techniques requires the evaluation of $\mathcal{L}$ to discriminate signal versus background events

Maximum Likelihood Fits

- It allows to estimate free parameters over a data sample, by minimizing the corresponding Negative Log-Likelihood (NLL) function (extended likelihood)

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N} \left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j(\hat{x}_i|\hat{\theta}_j) \right)$$

s species, i.e. signals and backgrounds
n_j number of events belonging to the species j

- The procedure of minimization can require several evaluation of the NLL
  - Depending on the complexity of the function, the number of observables, the number of free parameters, and the number of events, the entire procedure can require long execution time
  - Mandatory to speed-up the execution

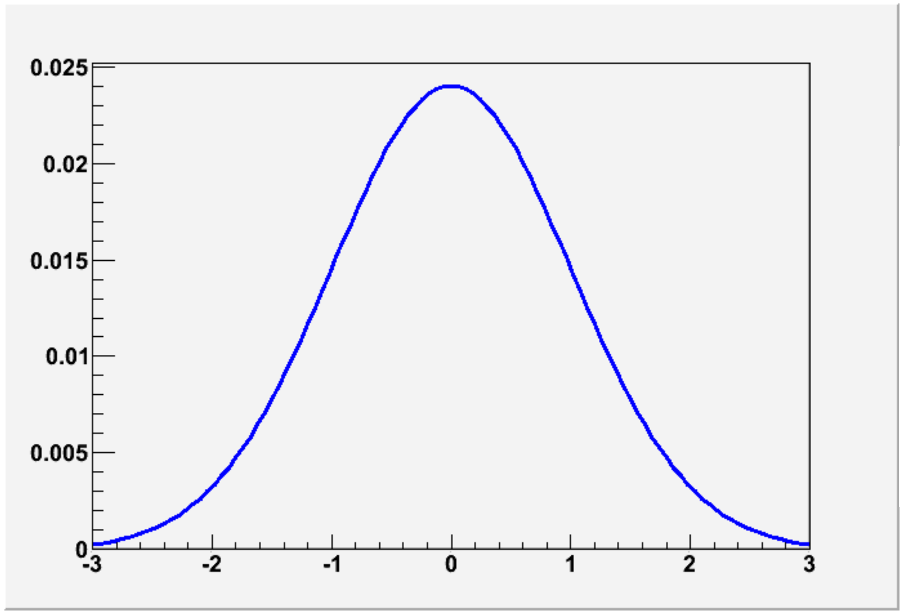
Examples

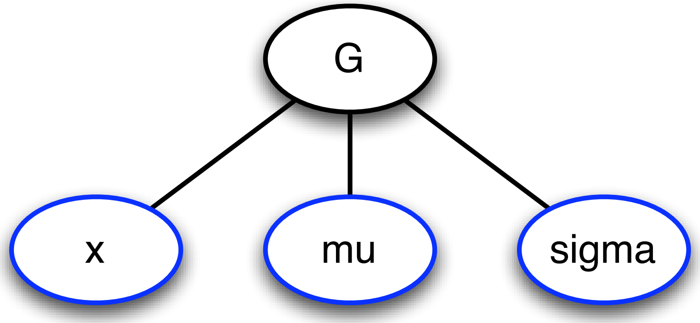
- In most cases PDFs can be factorized as product of the n PDFs of each variable (i.e. case of uncorrelated variables)

$$\mathcal{P}_j(\hat{x}_i|\hat{\theta}_j) = \prod_{v=1}^{n} \mathcal{P}_j^v(x_i^v|\hat{\theta}_j)$$

Gaussian

$G(x|\mu,\sigma)$



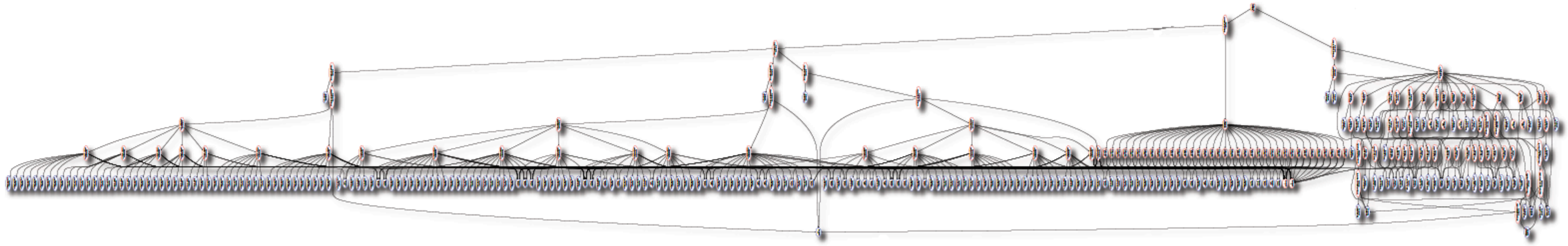


Examples

- In most cases PDFs can be factorized as product of the n PDFs of each variable (i.e. case of uncorrelated variables)

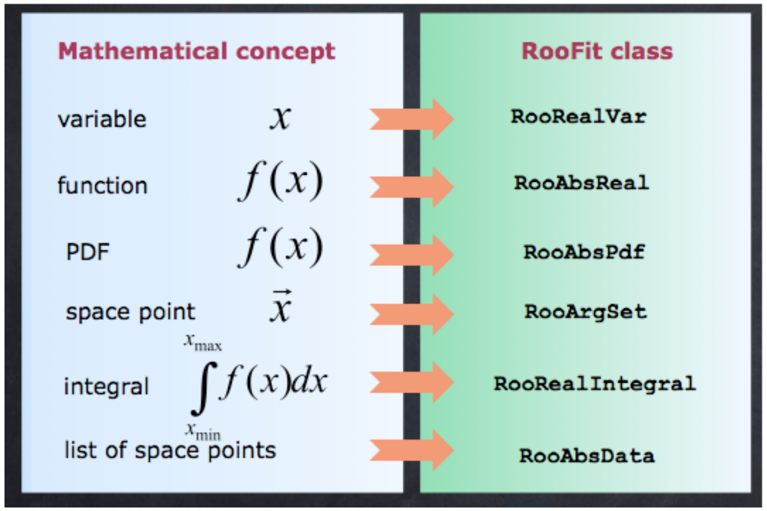
$$\mathcal{P}_j(\hat{x}_i|\hat{\theta}_j) = \prod_{v=1}^{n} \mathcal{P}_j^v(x_i^v|\hat{\theta}_j)$$

Combined Atlas & CMS Higgs analysis:
12 variables
50 free parameters

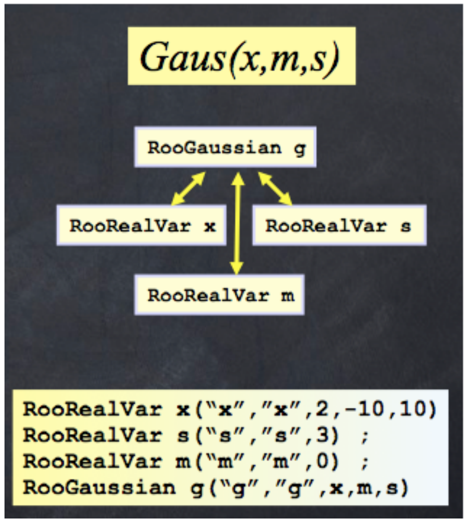


Building models: RooFit

- RooFit is commonly used in High Energy Physics experiments to define the likelihood functions (W. Verkerke and D. Kirkby)
  - Details at http://root.cern.ch/drupal/content/roofit
  - Mathematical concepts are represented as C++ objects



Mathematical concept		RooFit class
variable	x	RooRealVar
function	$f(x)$	RooAbsReal
PDF	$f(x)$	RooAbsPdf
space point	$\vec{x}$	RooArgSet
integral	$\int_{x_{min}}^{x_{max}} f(x)dx$	RooRealIntegral
list of space points		RooAbsData



```
RooRealVar x("x","x",2,-10,10)
RooRealVar s("s","s",3) ;
RooRealVar m("m","m",0) ;
RooGaussian g("g","g",x,m,s)
```

- On top of RooFit developed another package for advanced data analysis techniques, RooStats
  - Limits and intervals on Higgs mass and New Physics effects

MINUIT

- Numerical minimization of the *NLL* using MINUIT (F. James, Minuit, Function Minimization and Error Analysis, CERN long write-up D506, 1970)
- MINUIT uses the gradient of the function to find local minimum (MIGRAD), requiring
  - The calculation of the gradient of the function for each free parameter, naively

$$\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{\boxed{NLL(\hat{\theta}_0 + \hat{\mathrm{d}})} - \boxed{NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}}{2\hat{\mathrm{d}}}$$

2 function calls per each parameter

  - The calculation of the covariance matrix of the free parameters, i.e. evaluation of the second order derivatives
- The minimization is done in several steps moving in the Newton direction: each step requires the calculation of the gradient

  ➪ Several calls to the *NLL*

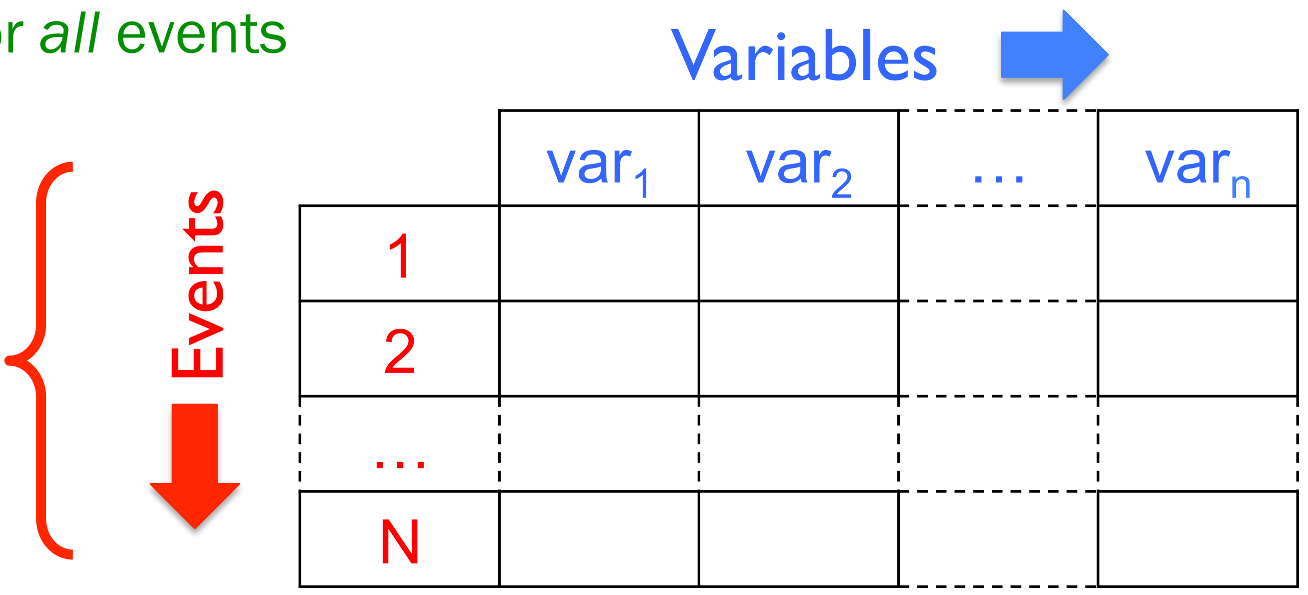
Caveats

- We developed a new algorithm for the likelihood function evaluation to be added in RooFit
  - We don't replace the current RooFit algorithm, which is used for results checking
  - Very chaotic situation: users can implement any kind of model
  - No need to change the user code to use the new implementation, i.e. same interface (use a simple flag to switch to the new algorithm)
- The new algorithm is optimized to run on the CPU
  - Used as reference for the GPU implementation: "fair" comparison
- All data in the calculation are in double precision floating point numbers
- Our target is to use commodity systems (e.g. laptops or desktops), easily accessible to data analysts
  - Of course we tests also on server systems

Likelihood Function evaluation in RooFit (1)

1. Read the values of the variables for each event
2. Make the calculation of PDFs for each event
   - Each PDF has a common interface declared inside the class RooAbsPdf with a virtual method which defines the function
   - Automatic calculation of the normalization integrals for each PDF
   - Calculation of composite PDFs: sums, products, extendend PDFs
3. Loop on all events and make the calculation of the *NLL*
   - A *single* loop for *all* events

Parallel execution over the events (***by fork***), with final reduction of the contributions



Likelihood Function evaluation in RooFit (2)

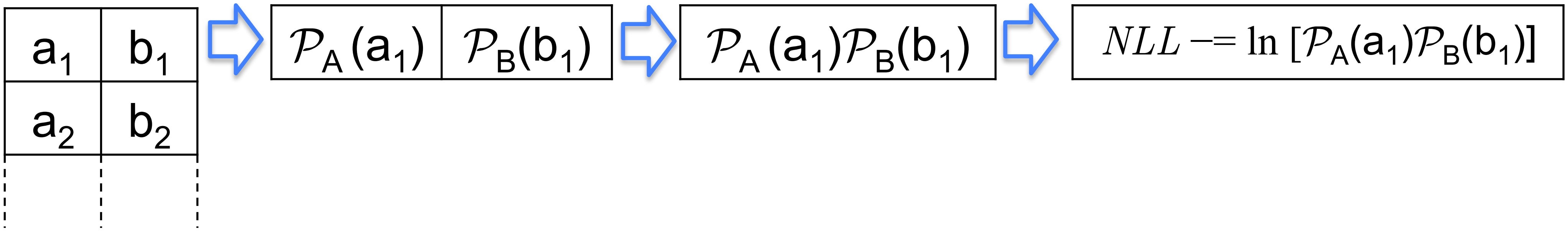
Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$

$$\boxed{NLL = 0}$$

a_1	b_1
a_2	b_2

Likelihood Function evaluation in RooFit (2)

Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$


a_1
b_1
a_2
b_2
$\mathcal{P}_A(a_1)$
$\mathcal{P}_B(b_1)$
$\mathcal{P}_A(a_1)\mathcal{P}_B(b_1)$
$NLL -= \ln[\mathcal{P}_A(a_1)\mathcal{P}_B(b_1)]$

Likelihood Function evaluation in RooFit (2)

Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$

$$\begin{tikzcd} \begin{array}{|c|c|}\hline a_1 & b_1 \\ \hline a_2 & b_2 \\ \hline \vdots & \vdots \end{array} \arrow[r] & \boxed{\mathcal{P}_A(a_2) \mid \mathcal{P}_B(b_2)} \arrow[r] & \boxed{\mathcal{P}_A(a_2)\mathcal{P}_B(b_2)} \arrow[r] & \boxed{NLL \mathrel{-}= \ln\,[\mathcal{P}_A(a_2)\mathcal{P}_B(b_2)]} \end{tikzcd}$$

Likelihood Function evaluation in RooFit (3)

Looping over all events and do the accumulation on *NLL*

- Data are stored in something like ROOT TTree (RooTreeDataStore)
  - Very inefficient. At then our variables are simple float/double/int values
  - It breaks any possible vectorization
  - No thread safe, parallelization done with a fork, i.e. no shared memory
- In the C++ OO spirit, there is a common interface (RooAbsReal) and then virtual methods in all derivate classes
  - Each PDF calls virtual methods to access parameters, the observables, the integral value for the normalization, calculation of the ln's, …
  - In case of composite PDFs (e.g. sums, products) it requires the call to virtual method of corresponding PDFs
  - A lot of virtual function calls!
- If the PDF doesn't change in the minimization, they are precalculated for all events and stored as a standard variable in the dataset
  - Not efficient way for caching the values of the PDFs
  - It doesn't take in account caching of constant values of the PDF inside a single minimization iteration

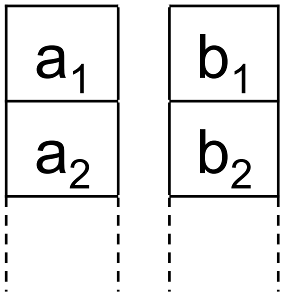
Likelihood Function evaluation in RooFit (4)

- PDFs are considered as independent entities, i.e. a PDFs doesn’t know if it is called inside a minimization process, from a mother composite PDF, or with a direct call
  - A PDF is not responsible to read the corresponding data
  - The PDF provides a single result for a given values of the data and parameters
  - In case of calculation which gives errors (e.g. negative probability), we get a warning message for the given values of the data and parameters
- Parallelization with a fork increases the memory footprint with the number of threads, but data are read-only!
  - Still it is easy to implement and it gives good scalability
- At the end, we are doing the evaluation of functions (PDFs) over a vector of read-only data!
  - Suitable for loop parallelism (note functions can be very complex!)

New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

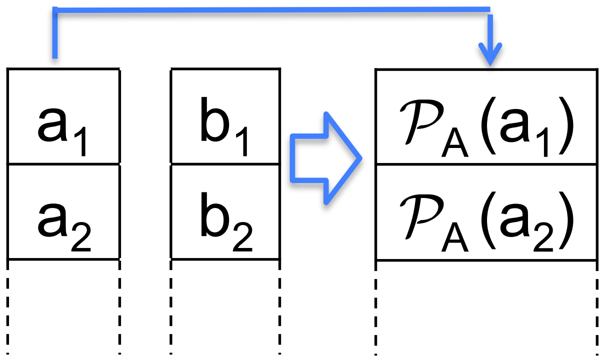
Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$



New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

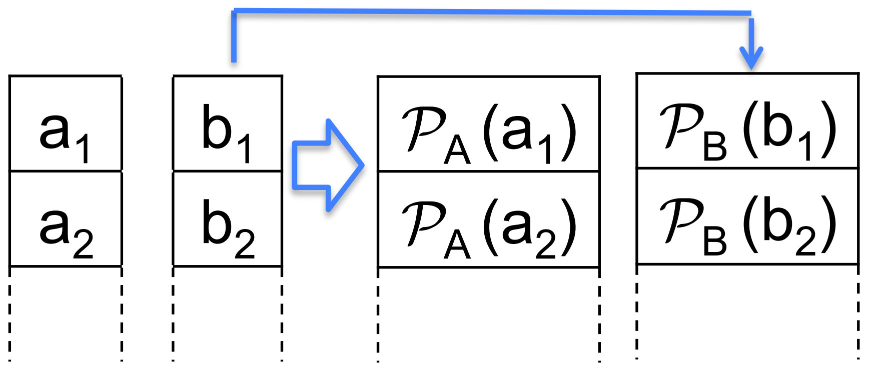
Ex: $\mathcal{P} = \mathcal{P}_\mathsf{A}(\mathsf{a}_i)\ \mathcal{P}_\mathsf{B}(\mathsf{b}_i)$



New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

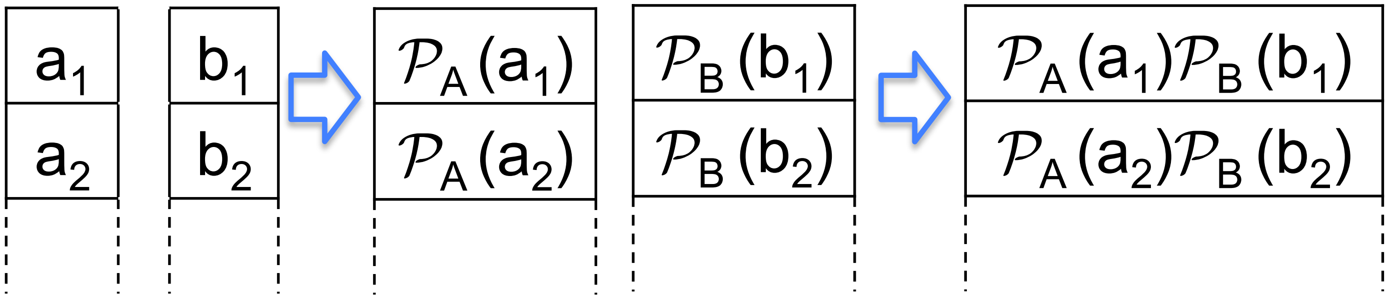
Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$



New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

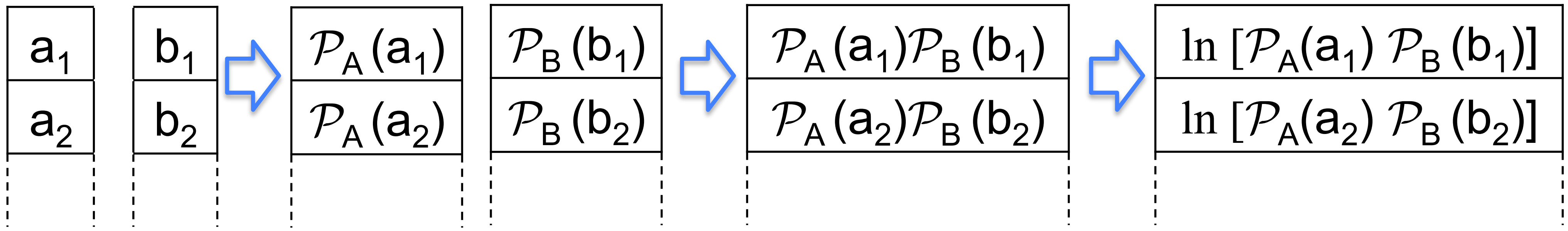
Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$



New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$

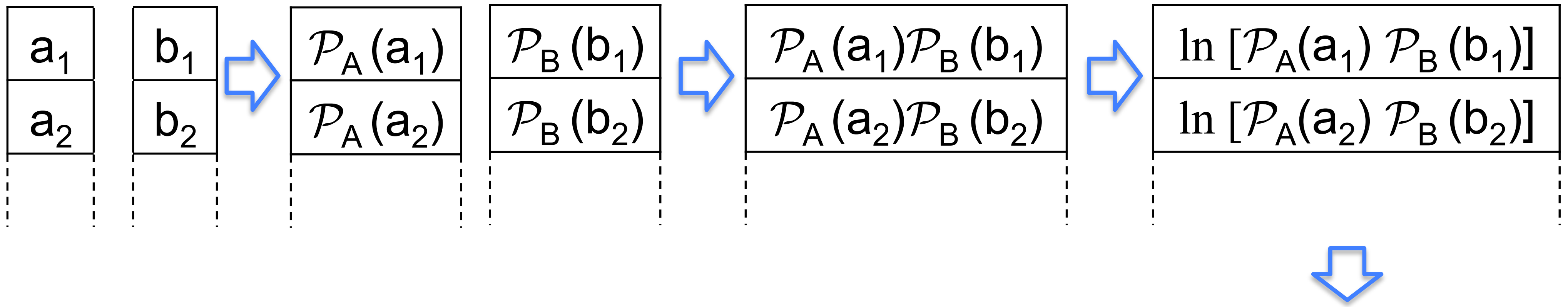


a_1
a_2
b_1
b_2
$\mathcal{P}_A(a_1)$
$\mathcal{P}_A(a_2)$
$\mathcal{P}_B(b_1)$
$\mathcal{P}_B(b_2)$
$\mathcal{P}_A(a_1)\mathcal{P}_B(b_1)$
$\mathcal{P}_A(a_2)\mathcal{P}_B(b_2)$
$\ln[\mathcal{P}_A(a_1)\ \mathcal{P}_B(b_1)]$
$\ln[\mathcal{P}_A(a_2)\ \mathcal{P}_B(b_2)]$

New algorithm and parallelization (I)

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF
3. Combine the arrays of results (composite PDFs)
4. Loop over the final array of results to calculate NLL (final reduction)

Ex: $\mathcal{P} = \mathcal{P}_A(a_i)\ \mathcal{P}_B(b_i)$



a_1
a_2
b_1
b_2
$\mathcal{P}_A(a_1)$
$\mathcal{P}_A(a_2)$
$\mathcal{P}_B(b_1)$
$\mathcal{P}_B(b_2)$
$\mathcal{P}_A(a_1)\mathcal{P}_B(b_1)$
$\mathcal{P}_A(a_2)\mathcal{P}_B(b_2)$
$\ln[\mathcal{P}_A(a_1)\ \mathcal{P}_B(b_1)]$
$\ln[\mathcal{P}_A(a_2)\ \mathcal{P}_B(b_2)]$

Final reduction in NLL

New algorithm and parallelization (2)

- Parallelization splitting calculation of each PDF over the events (data parallelism) and over the independent PDFs (task parallelism)
- Data are organized in vector, which are shared in memory
  - Perfect for vectorization
- Call the PDFs once for all events
  - Reduce dramatically the number of virtual function calls!
  - Perfect for caching values over the iterations during the minimization
- Drawbacks
  - Require to handle arrays of temporary results: 1 value per each event and PDF
  - Memory footprint increases with the number of events and number of PDFs, but not with the number of threads!
  - Due to the vectorization, we cannot have warning messages for a given event, but only at the end of the loop for the calculation over all events

Implementation in RooFit

- First of all we added a new class to manage the data as vectors (based on map of std::vector's, where the key is the name of the observable)
- We added a class to take in account the array of results (based on std::vector)
- The loop parallelism is implemented using OpenMP
  - An OpenMP pragma loop for each loop used in the evaluation of the function
- Added new methods to the PDF interface
  - Still the old interface is working
- Using Intel compiler for the auto-vectorization of the loops (using svml library by Intel)
  - GNU compiler cannot auto-vectorize complex functions (like exp's), unless you use intrinsics…

OpenMP parallelization

```cpp
// Inline method for the Gaussian PDF calculation,
// defined inside the class RooGaussian
inline double evaluateLocal(const double x,
                            const double mu,
                            const double sigma) const
{
  return std::exp(-0.5*std::pow((x-mu)/sigma,2));
}

// Virtual method for the calculation of the
// Gaussian PDF on a single event
// (this is the original RooFit algorithm)
virtual double evaluate() const
{
  return evaluateLocal(x,mu,sigma);
}


// Virtual method for the calculation of the
// Gaussian PDF on all events
// (new implemented algorithm)
virtual bool evaluate(const RooAbsData& data)
{
  // retrive the data array of values for the variable
  const double *dataArray = data.GetDataArray(x.arg());
  // check if there is an array for the variable
  if (dataArray==0)
    return false;

  // retrive the number of events
  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  double *resultsArray = GetResultsArray();
  double m_mu = mu;
  double m_sigma = sigma;

  // loop over the events to calculate the Gaussian
#pragma omp parallel for
  for (int idx = 0; idx<nEvents; ++idx) {
    resultsArray[idx] = evaluateLocal(dataArray[idx],
                                      m_mu,m_sigma);
  }

  return true;
}
```

- Very easy parallelization with OpenMP
- Take benefit from the code optimizations
  - Inlining of the functions, no virtual functions
  - Data organized in C arrays, perfect for vectorization
- Easily avoid race conditions, keep the parallel region limited inside each PDF

Parallel reduction

- The final reduction for the NLL evaluation done in parallel using block-wise algorithm
  - Numerical approximation w.r.t. sequential reduction, which are number of threads dependent
  - Minuit is very sensitive to these approximation
    - Of course differences are negligible, but still they can worry people (and they can be non deterministic)
- We implemented a parallel reduction based on double-double algorithm which reduces the approximations (Y. He and C. H. Q. Ding, The Journal of Supercomputing, 18, 259–277, 2001; P. Kornerup *at al.*, IEEE Transactions on Computers, 01 Feb. 2011)
  - We need to switch off any compiler optimization inside the reduction, using pragmas
- Now the results are identical up to $10^{-6,}$ no matter how many threads you are running

Complex Model Test

$$n_a[f_{1,a}G_{1,a}(x) + (1 - f_{1,a})G_{2,a}(x)]AG_{1,a}(y)AG_{2,a}(z)+$$
$$n_bG_{1,b}(x)BW_{1,b}(y)G_{2,b}(z)+$$
$$n_cAR_{1,c}(x)P_{1,c}(y)P_{2,c}(z)+$$
$$n_dP_{1,d}(x)G_{1,d}(y)AG_{1,d}(z)$$

Model from B. Aubert *et. al.*,
Phys. Rev. Lett. 98, 031801, 2007

17 PDFs in total, 3 variables, 4 components, 35 parameters

- G: Gaussian
- AG: Asymmetric Gaussian
- BW: Breit-Wigner
- AR: Argus function
- P: Polynomial

40% of the execution time is spent in exp's calculation

Note: all PDFs have analytical normalization integral, i.e. >98% of the sequential portion can be parallelized

Test on CPU in sequential

- Dual socket Intel Westmere-based system: CPU (L5640) @ 2.27GHz (12 physical cores, 24 hardware threads in total), 10x4096MB DDR3 memory @ 1333MHz
- Linux 64bit, Intel C++ compiler version 12.0.2

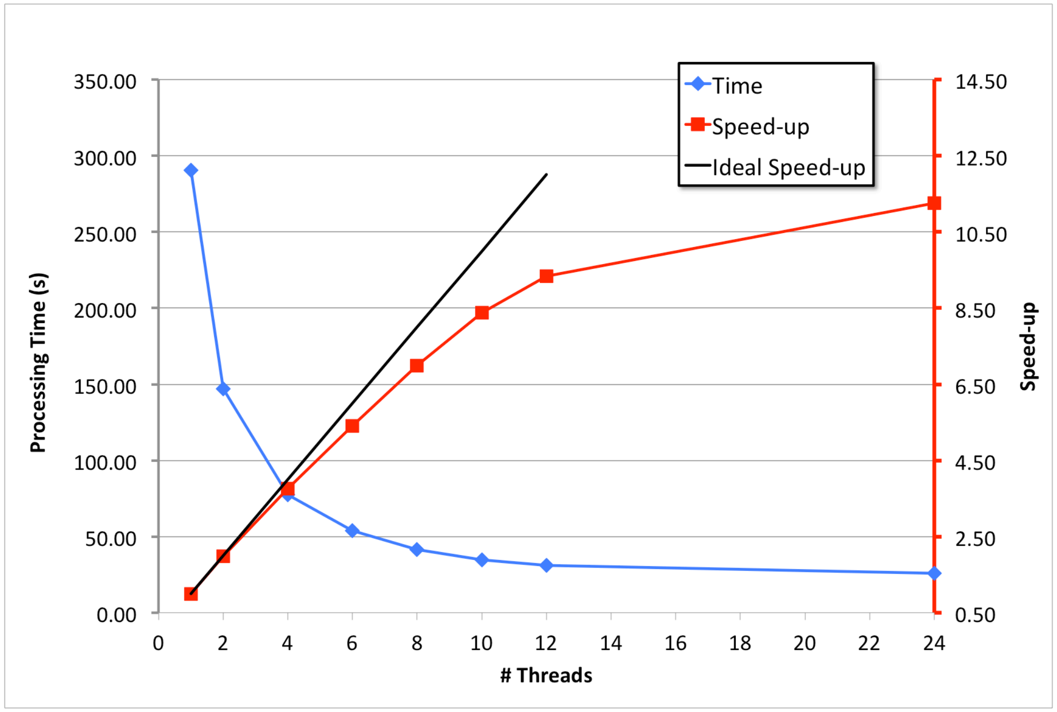
# Events	10,000	25,000	50,000	100,000
RooFit				
# *NLL* evaluations	15810	14540	19041	12834
Time (s)	826.0	1889.0	5192.9	6778.9
Time per *NLL* evaluation (ms)	52.25	129.92	272.72	528.19
OpenMP (w/o vectorization)				
# *NLL* evaluations	15237	17671	15761	11396
Time (s)	315.1	916.0	1642.6	2397.3
Time per *NLL* evaluation (ms)	20.68	51.84	104.22	210.36
w.r.t. RooFit	2.5x	2.5x	2.6x	2.5x
OpenMP (w/ vectorization)				
# *NLL* evaluations	15304	17163	15331	12665
Time (s)	178.8	492.1	924.2	1536.9
Time per *NLL* evaluation (ms)	11.68	28.67	60.28	121.35
w.r.t. RooFit	4.5x	4.5x	4.4x	4.4x

4.5x faster!

Vectorization gives a 1.8x speed-up (SSE). Additional 12% using AVX on Intel Sandy Bridge

Test on CPU in parallel

- Dual socket Intel Westmere-based system: CPU @ 2.67GHz (12 physical cores, 24 hardware threads in total), Turbo Mode ON, 10x4096MB DDR3 memory @ 1333MHz
- Linux 64bit, Intel C++ compiler version 12.0.2
- 100,000 events
- Data is shared, i.e. no significant increase in the memory footprint
  - Possibility to use Hyper-threading (about 20% improvement)
- Limited by the sequential part, OpenMP overhead, and memory access to data



Improvements

- Scalability is limited by accessing the array of results
  - In particular the effect becomes important for PDFs with simple function, like polynomials and composite PDFs (add and prod)
  - We do pinning of the threads to the physical cores, taking in account the NUMA effect
  - However the performance depends on the cache memory available on the systems
    - Testing on a 4 core i7 desktop system (8 MB L3 cache) we reach a factor ~2x with 8 threads (using SMT)
- We solve this problem with different techniques
  - Merge the number of OpenMP parallel region and reuse the data (in particular for composite PDFs)
  - Do block-splitting, i.e. do full evaluation for small sub-groups of events
- Doing this optimization we are able to reach 4.6x on the 4 core i7 desktop system (8 threads with SMT)

Conclusion (1)

- Implementation of the algorithm in OpenMP required not so drastic changes in the existing RooFit code
  - In any case we added our implementation, so that users can use the original implementation for reference
- Optimization gives a great speed-up: ~5x
- Note that our target is running at the user-level of small systems (laptops, desktops), i.e. with small number of CPU cores
- Very important to take under control numerical accuracy
  - We would like to try single precision in case of PDF evaluation, moving to double precision for the final reduction
    - Reduce memory footprint (half space for results)
    - Gain a factor possible 2x from vectorization

Conclusion (2)

- Try the code on LHC analyses
  - Dalitz analysis
  - Working with RooStats authors
- We are also evaluating Intel MIC platform, which looks very promising as accelerator system (very easy to use it)
  - x86 instruction set accelerator
  - 512-bit SIMD units
  - More than >50 cores
- There will a workshop at CERN discussing “Future Challenges in Tracking and Trigger Concepts”: http://indico.cern.ch/event/tracking2011