# Parallelization of likelihood functions for data analysis

**Alfio Lazzaro**
**alfio.lazzaro@cern.ch**
**CERN openlab**


Forum on Concurrent Programming Models and Frameworks

15 February 2012

# Introduction

- Data analysis is a very important task
  - See presentation by Lorenzo at the last meeting:
    - https://indico.cern.ch/conferenceDisplay.py?confId=174781
- Important at the user level:
  - No centralized production, user based workload
  - Floating point intensive application
  - Strong scaling: go fast!
  - Possibility to be aggressive in the optimization
- The aim of the presentation is to give an overview (no technicalities) on the work we are doing at CERN openlab
  - Based on a prototype of ROOT/RooFit (~4K lines of code)
    - Data analysis model taken from *B* physics
  - Porting to ROOT is underway
    - http://root.cern.ch/svn/root/branches/dev/openlab/
- Note that at openlab we are “only” interested to parallelize the code for our evaluations in collaboration with Intel

# Maximum Likelihood Fits

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

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N} \left[ \ln \sum_{j=1}^{s} \left( n_j \prod_{v=1}^{n} \mathcal{P}_j^v(x_i^v|\hat{\theta}_j) \right) \right]$$

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

P probability density function (PDF) for n observables
s species, i.e. signals and backgrounds
$n_j$ number of events belonging to the species j

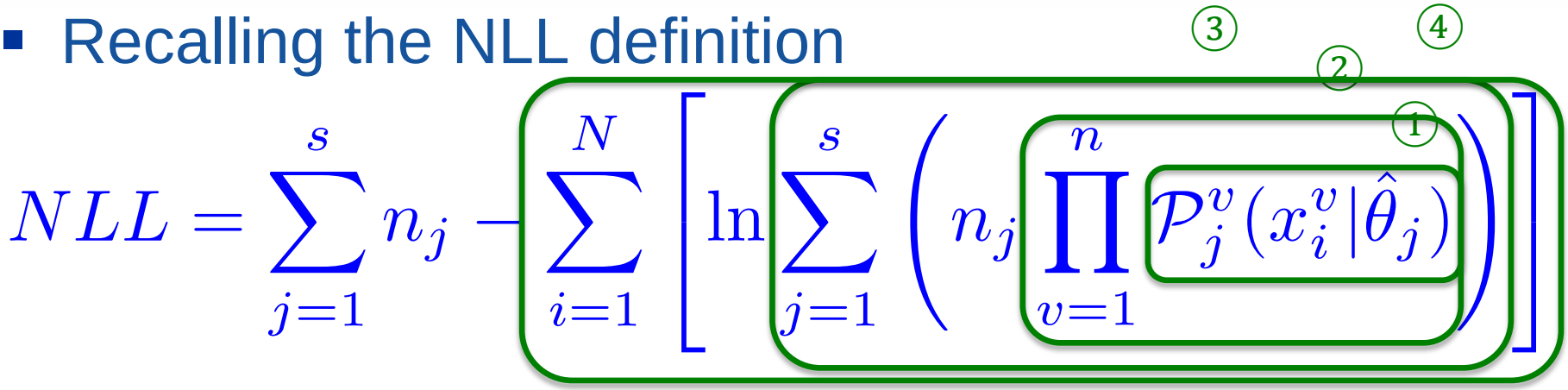
- The procedure of minimization can require several evaluation of the NLL
  - Depending on the complexity of the function, the number of observables, the number of free parameters, and the number of events, the entire procedure can require long execution time
  - Mandatory to speed-up the evaluation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

# Requirements

- The code is implemented in a library used for different users analyses
  - ROOT/RooFit in C++ code
  - All data in the calculation are in double precision floating point numbers
- Very chaotic situation: users can implement any kind of function
  - Parallelization is “encapsulated” in the library, i.e. no need to change the user code to use the different parallel implementation
  - Easy add of new PDFs
  - Intensive use of transcendental functions
  - Use a simple flag to switch between parallel implementations
- Use of commodity systems, no hardware specific
- Very important to have predictable results
  - Results should not depend on the specific parallel implementation and number of threads involved

# Algorithm Description

- Recalling the NLL definition




① Each P (Gaussian, Polynomial,…) is implemented with a corresponding class (basic PDF)
  - Virtual protected method to evaluate the function
  - Virtual public method to return the normalized value

② Product over all observables (composite PDF)

③ Sum over all species (composite PDF)

④ Reduction of all values

# Algorithm Tree

- We can visualize the NLL evaluation as a tree

$$\mathcal{P}_1^1$$

$$n_1 \prod_{v=1}^{n}$$

$$\sum_{i=1}^{N}$$

$$\ln \sum_{j=1}^{s}$$

reduction

$$\mathcal{P}_s^1$$

$$n_s \prod_{v=1}^{n}$$

Composite PDFs

Basic PDFs

# Algorithm Evaluation

- Data are organized in memory in vectors
  - A vector for each observable
  - Read-only during the NLL evaluation
- The NLL evaluation consists to traverse the entire tree, first evaluating the leaves up to the root

# Algorithm Evaluation

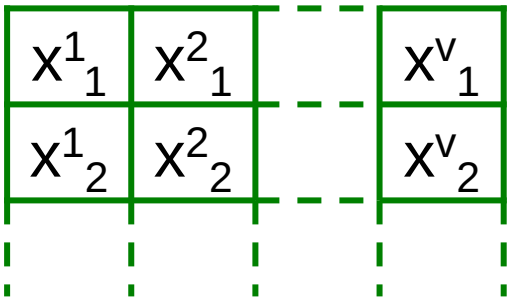
1. Read the observable values for a given event
2. Traverse the entire NLL tree
   - Do the entire evaluation for each event
3. Loop for all events and accumulate the results

$$\mathcal{P}_1^1$$

$$n_1 \prod_{v=1}^{n}$$

$$\sum_{i=1}^{N}$$

$$\ln \sum_{j=1}^{s}$$




$$\mathcal{P}_s^1$$

$$n_s \prod_{v=1}^{n}$$

# Algorithm Evaluation

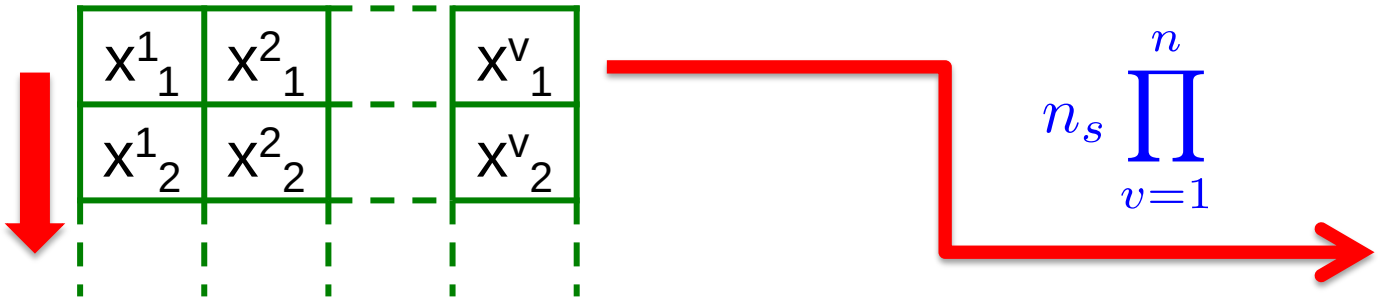
1. Read the observable values for a given event
2. Traverse the entire NLL tree
   - Do the entire evaluation for each event
3. Loop for all events and accumulate the results

$$\mathcal{P}_1^1$$

$$n_1 \prod_{v=1}^{n}$$

$$\sum_{i=1}^{N}$$

$$\ln \sum_{j=1}^{s}$$




$$\mathcal{P}_s^1$$

$$n_s \prod_{v=1}^{n}$$

# Current parallelization

Events are independent, but the reduction!

- Apply parallelization on the loop over the events (constants PDFs can be calculated once during the minimization)
- Do the final reduction and get the final value

Implemented inside ROOT/RooFit with fork

- Easy change in the code
- Good scalability: ~11x on 12 threads

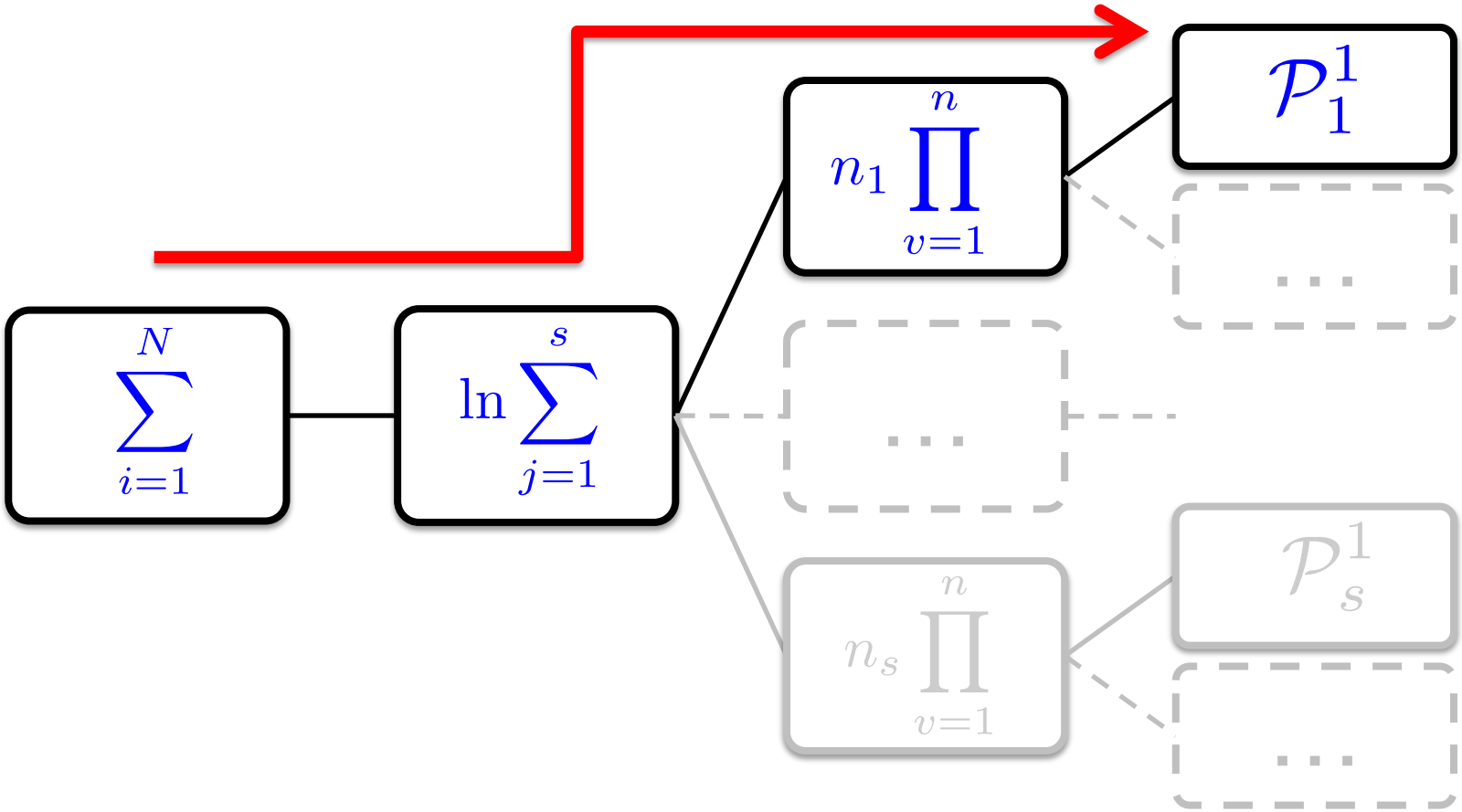
but

- Copy of everything to avoid false sharing
- At least data (read-only) can be shared!

Can we do better? Re-design the algorithm

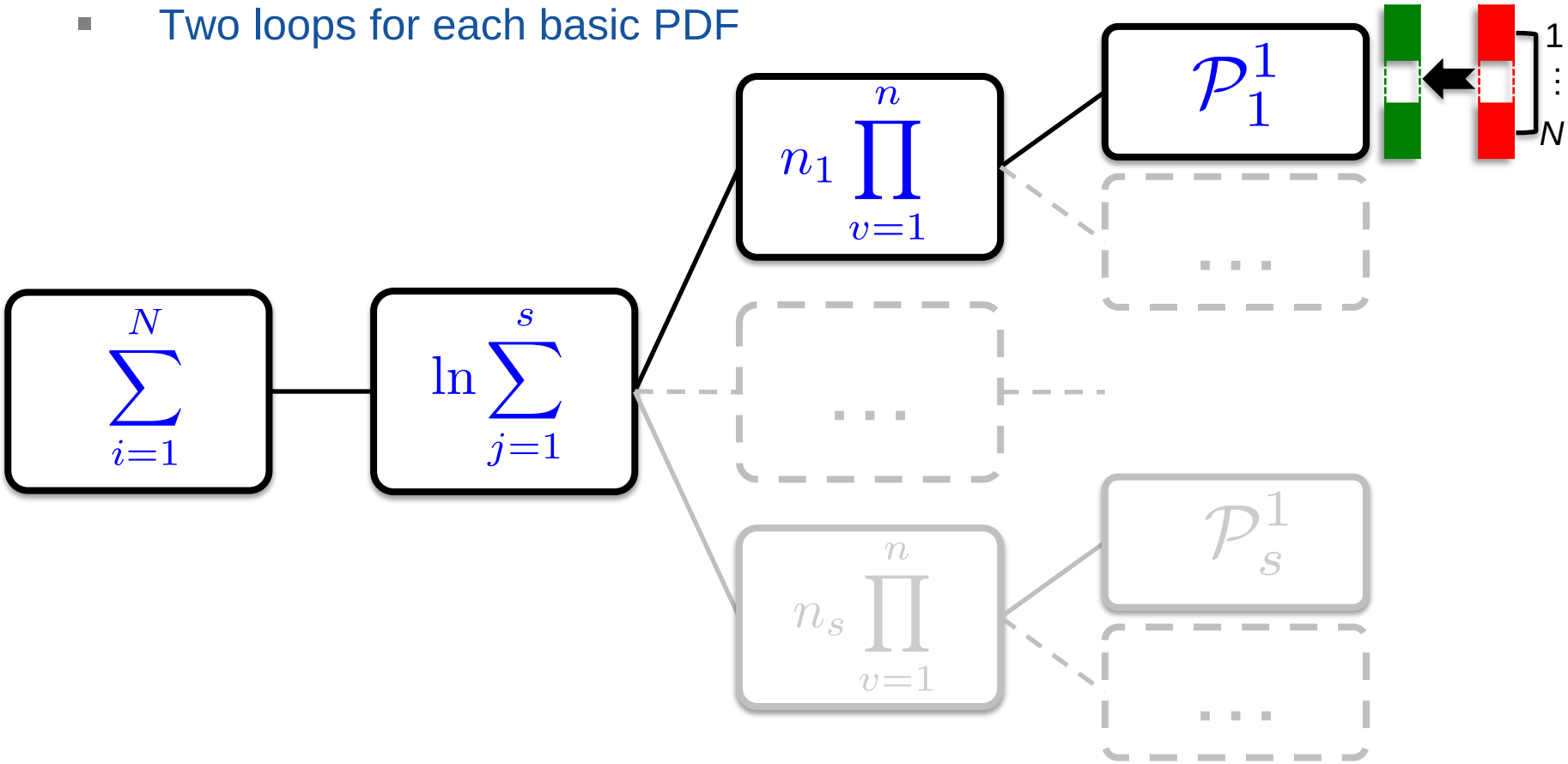
# Algorithm Evaluation

1. Traverse the NLL tree up to the first leaf
   - For each composite PDF and each basic PDF



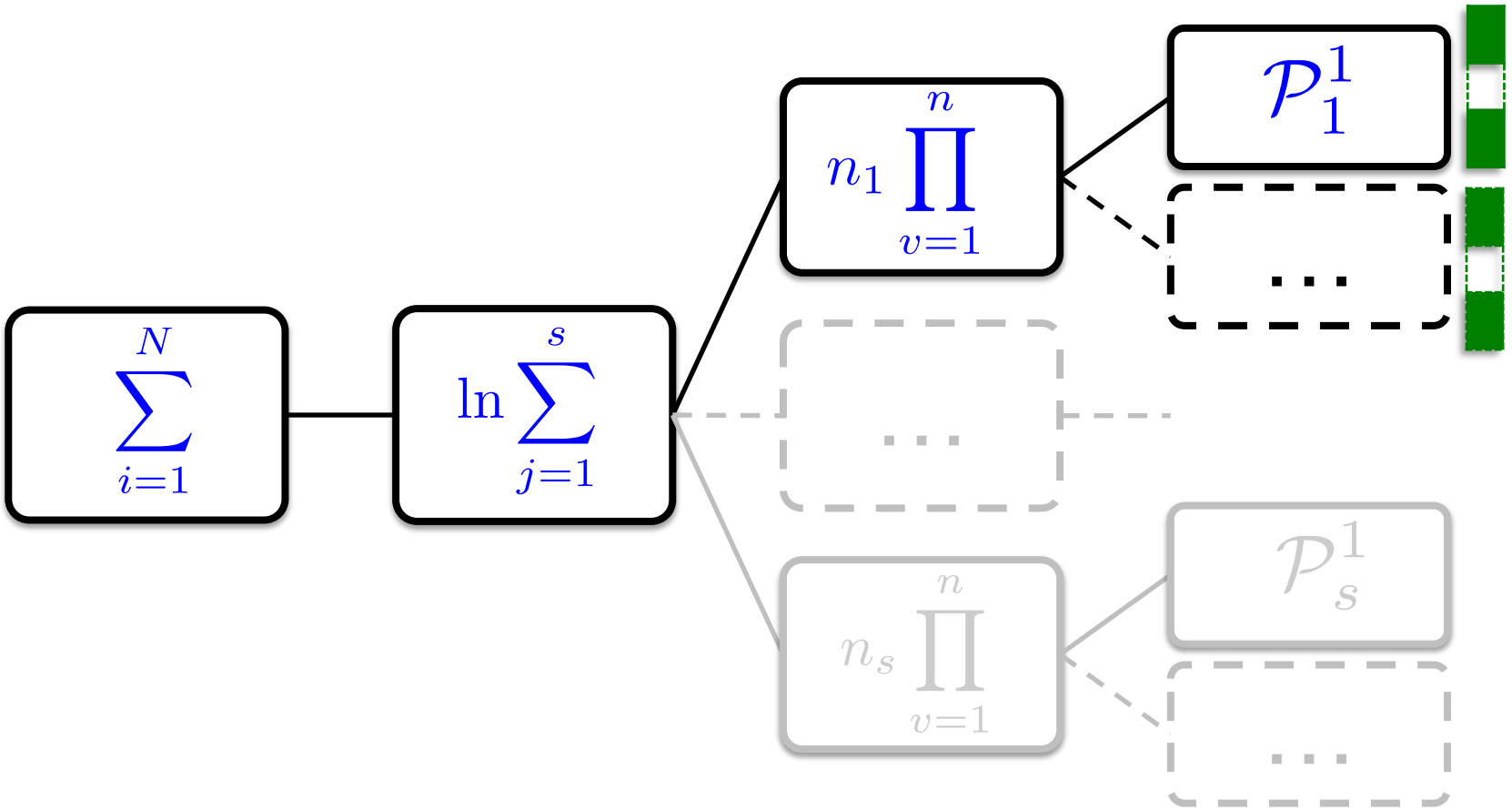
# Algorithm Evaluation

2. Loop over the *N* events and evaluate the function for each event
   - Produce a corresponding array of results
3. Loop over the results and apply the normalization
   - Two loops for each basic PDF



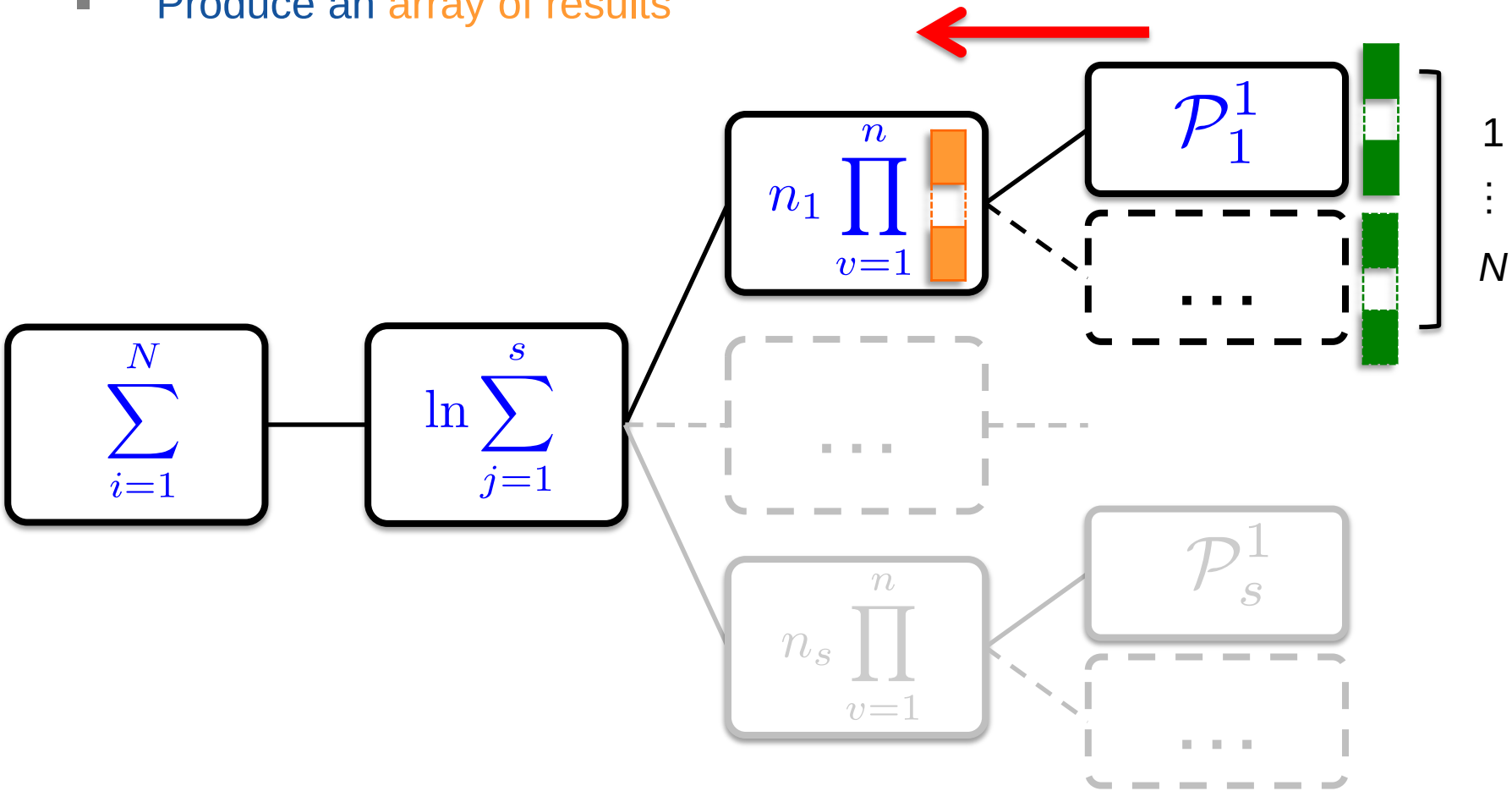
# Algorithm Evaluation

4. Repeat the evaluation for all basic PDF in a composite PDF
   - Produce an array of results for each basic PDF



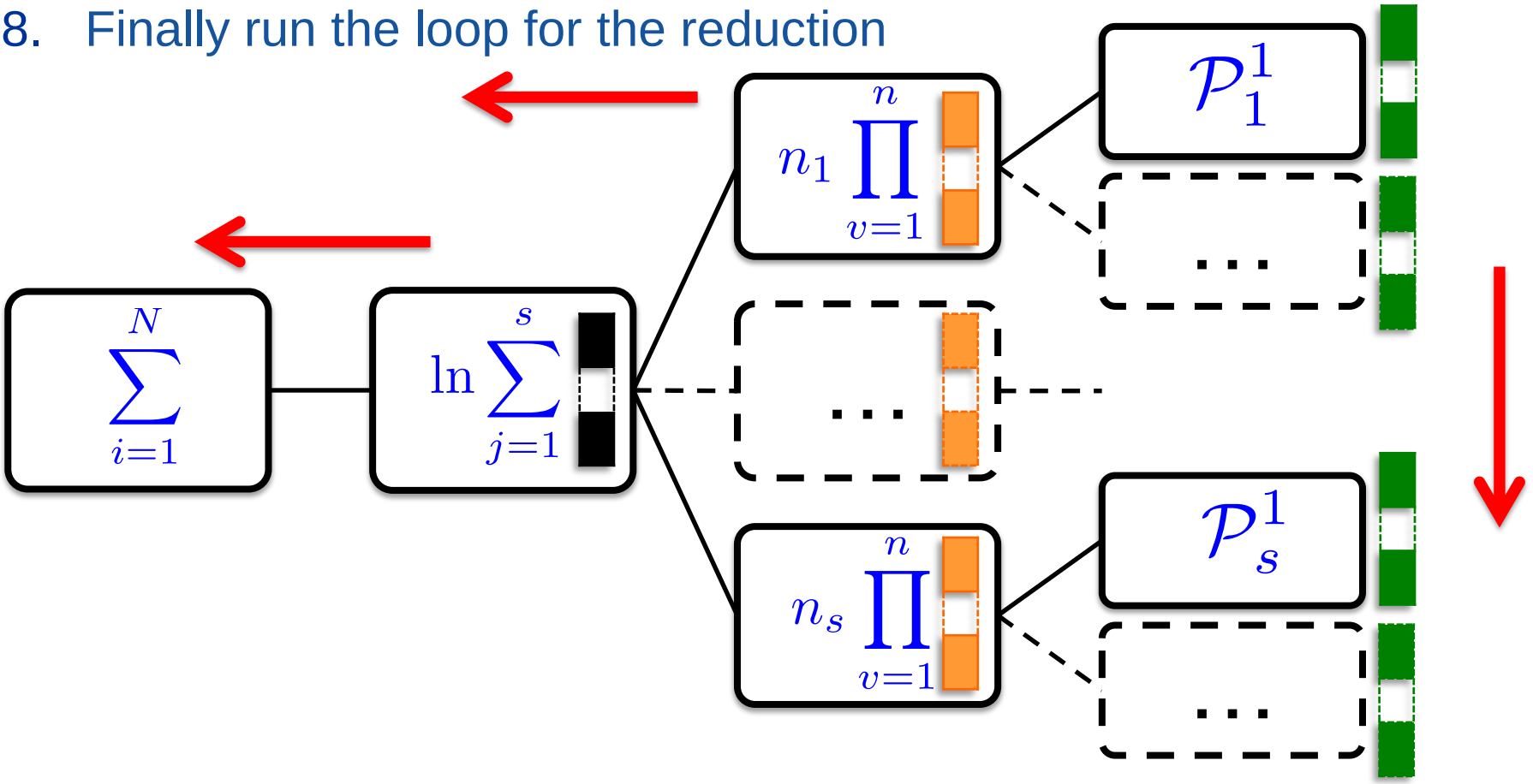
# Algorithm Evaluation

5. Combine the array of results for the composite PDF
   - Loop over the array of results of the basic PDF
   - Produce an array of results



# Algorithm Evaluation

6. Repeat for all composite PDFs
7. Loop over the array of results
   - Produce the final array of results
8. Finally run the loop for the reduction



# Implementation

```
// 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;
}
```

- Take benefit from the code optimizations
  - No virtual functions
  - Inlining of the functions
- Evaluation of functions over arrays of read-only data
  - Balanced independent iterations
- Input data are shared in memory
  - Memory footprint increases with the number of events and number of PDFs, but not with the number of threads
- Possible to exploit vectorization
  - Using Intel compiler for the auto-vectorization of the loops (using svml library by Intel)
- Very easy parallelization with OpenMP
  - Easy thread-safety, limiting the parallelization to the PDF loops

**NOTE: error checking inside the loops with output warnings will destroy vectorization and parallelization**

# Test on CPU in sequential

- Intel Westmere-based system: CPU (X5650) @ 2.67GHz, 12MB L3 cache
- Intel C++ compiler version 12.1.0
- Input data is composed by 500,000 events per 3 observables, for a total of about 12MB; results are stored in 29 vectors of 500,000 values, i.e. about 110MB
- ~85% of the execution time of the sequential code is spent in floating-point operations
- Results:
  - Original RooFit algorithm: 1226s
  - New algorithm (vectorization off): 449s
  - New algorithm (SSE vectorization): 259s

Vectorization gives a 1.7x speed-up (SSE)

- Total speed-up: 4.7x

# Parallelization: limitations

- >99% of the sequential time can be parallelized
- Testing on dual socket Intel 6-core “Westmere”-based server system, 2 threads per each socket, 2*12MB L3 cache
  - With 12 threads speed-up is 7.6x (8.9x using SMT 24 threads)
  - Well below the Amdahl’s law prediction! (>10.8x with 12 threads)

- Analysis of the bottlenecks:
  - Several independent OpenMP regions
    - OpenMP overhead becomes consistent with high number of threads
  - Performance depends on the cache memory available on the systems

# Parallelization: limitations

- Accessing the arrays of results: overlap computation and memory accesses
  - The amount of arrays to manage becomes consistent in case of complex models and large data samples
    - Crucial to have an optimal treatment of the data inside the cache memories
  - Effect particularly important for PDFs with simple function, like polynomials, and for the normalization loop (i.e. a product) and composite PDFs
    - Composite PDFs have to combine several arrays of results with just a simple operation (i.e. products and sums)
    - Fast computation, not enough time to fetch the data from memory

| /Function /Call Stack | CPU Time |
|---|---|
| __svml_exp2.N | 39.1% |
| PdfPolynomial::evaluateOpenMP | 11.5% |
| PdfArgusBG::evaluateOpenMP | 8.2% |
| PdfGaussian::evaluateOpenMP | 6.8% |
| PdfAdd::evaluateOpenMP | 6.5% |
| [libiomp5.so] | 6.1% |
| PdfProd::evaluateOpenMP | 5.4% |
| AbsPdf::GetVal | 4.3% |
| NLL::GetVal | 3.6% |
| PdfBifurGaussian::evaluateOpenMP | 2.9% |

(a) $N = 100\ 000$

| /Function /Call Stack | CPU Time |
|---|---|
| PdfProd::evaluateOpenMP | 22.8% |
| __svml_exp2.N | 18.5% |
| PdfAdd::evaluateOpenMP | 17.8% |
| PdfPolynomial::evaluateOpenMP | 11.8% |
| PdfGaussian::evaluateOpenMP | 6.6% |
| AbsPdf::GetVal | 6.6% |
| PdfBifurGaussian::evaluateOpenMP | 4.7% |
| PdfArgusBG::evaluateOpenMP | 4.4% |
| [libiomp5.so] | 2.1% |
| __svml_log2.L | 1.7% |

(b) $N = 1\ 000\ 000$

24 SMT threads

# Optimizations and results

- Start only one OpenMP parallel region at the root of the NLL tree: each thread executes the entire evaluation from the root to the leaves within its own partition only
  - Minimum OpenMP overhead, but risk of race conditions
- Block-splitting: full evaluation for small sub-groups of events, i.e. decomposition of the loop iterations
  - Reuse of data, more cache-friendly
- Results (speed-up):
  - 12 threads: 10.9x (perfectly in agreement with the prediction)
  - 24 SMT threads: 14.9x! (better reuse of data in the cache)

# Other implementations and conclusions

- TBB implementation also implemented
  - It gives more “abstraction” from the hardware
  - It gives block-splitting for free
- CUDA and OpenCL implementations
- Working on a MPI implementation

- Conclusions
  - Working on prototypes can help
  - Redesign the algorithm
  - Keep the original algorithm for comparison
  - Several hardware-related and numerical problems when moving to parallelization