MINUIT Package Parallelization And Applications Using The RooFit Package

Alfio Lazzaro[1,2] and Lorenzo Moneta[1]

[1] CERN, Geneva
[2] Università degli Studi and INFN, Milano

The case: Data Analysis

- In general all methods are based on optimization problems: find a maximum (for example in case of Statistical Significance Maximization or Maximum Likelihood) or a minimum (Expected Prediction Error) of a function
- This is (in general) done by numerical algorithms
  - The most used require the calculation of the gradient of function (calculation of several derivatives)
  - Some example in Genetic Algorithm

Execution Time Issue

- The optimization process can be very slow, depending on:
  - number of free parameters to be determined
  - number of input events in the process
  - complexity of the model
- Standard philosophy in physics:
  - in general you can simplify your problem to have reasonable execution time and statistical errors on your results
  - Data dimensions scale (roughly) with the performance of CPU (gain time for free)

Execution Time Issue

- The optimization process can be very slow, depending on:
  - number of free parameters to be determined
  - number of input events in the process
  - complexity of the model
- Standard philosophy in physics:
  - in general you can simplify your pr… e reasonable execution time and stati… results
  - Data di… oughly) with the performance of CPU (gain tin… free)

Not valid anymore

Execution Time Issue

- The optimization process can be very slow, depending on:
  - number of free parameters to be determined
  - number of input events in the process
  - complexity of the model
- Standard philosophy in physics:
  - in general you can simplify your problem to have reasonable execution time and statistical errors on your results
  - Data dimensions scale (roughly) with the performance of CPU (gain time for free)

Execution Time Issue

- The optimization process can be very slow, depending on:
  - number of free parameters to be determined
  - number of input events in the process
  - complexity of the model
- Standard philosophy in physics:
  - in general you can simpli[...] execution time and statis[...]
  - Data dimensions scale (roughly) with the performance of CPU (gain time for free)

Not valid with current experiments: need to "squeeze" as much as possible from data

Execution Time Issue

- The optimization process can be very slow, depending on:
  - number of free parameters to be determined
  - number of input events in the process
  - complexity of the model
- Standard philosophy in physics:
  - in general you can simpli
    execution time and statis
  - Data dimensions scale (r U
    (gain time for free)

Not valid with current experiments: need to "squeeze" as much as possible from data

Not valid for the current CPUs!!! (multi/many-cores)

Think Parallel!

- Possibility to parallelize algorithms so we can benefit from new multi(many) cores architectures or cluster
  - OpenMP: shared memory
  - pthreads
  - Message Passing Interface (MPI): distributed memory on different nodes

All used in this work

- Remember the Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

  - S → speedup
  - P → portion of code which is parallelized
  - N → number of simultaneous process
- Need to find good algorithms to be parallelized!

$$S(N \to \infty) = \frac{1}{(1-P)}$$

Maximum Likelihood Fits

- In Maximum Likelihood fits we have to maximize the likelihood function

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^{s} n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{s} n_j \mathcal{P}_j^i.$$

j species (signals, backgrounds)
n_j number of events for specie j
P_j probability
N number total of events to fit

- In general we minimize the Negative Log-Likelihood Function

$$-\ln \mathcal{L} \equiv NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

- The minimization is performed as function of free parameters: n_j number of events, parameters of P_j

GOOD example of algorithm for easy parallelization

Minimization

- The most largely used algorithm for minimization is MINUIT
- 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 per derivative

  - The calculation of the covariance matrix of the free parameters (which means the second order derivatives)
- The minimization is done in several steps moving in the Newton direction: each step requires the calculation of the gradient (see the yesterday Lorenzo's talk for more details)

Minimization

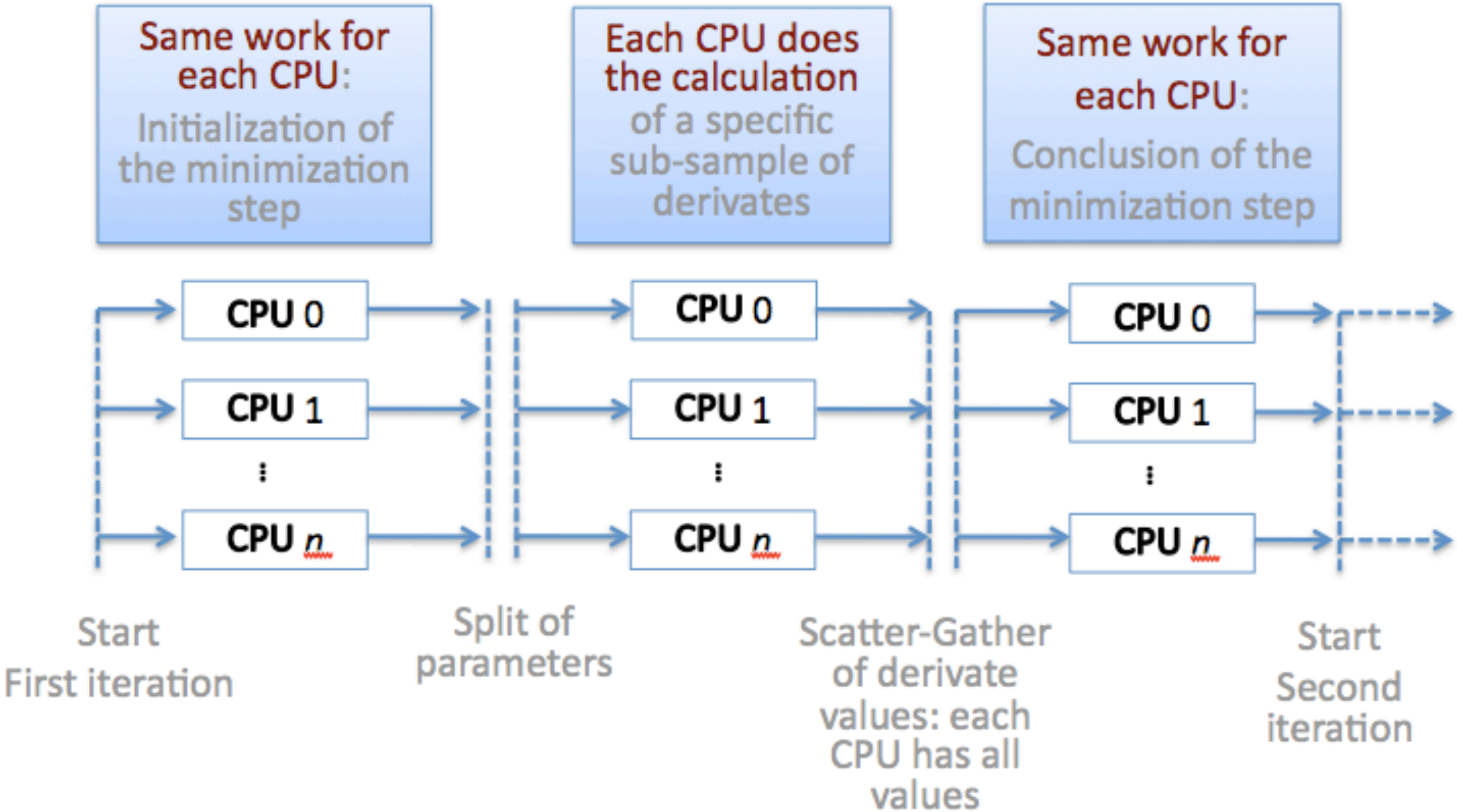
- In case of NLL function, it requires the calculation of the function for each free parameter in each minimization step
  1. Many free parameters means slow calculation
  2. Remember the definition of *NLL*

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

     The computational cost scales with the *N* number of events in the input sample
  3. Note, also, that P_j needs to be normalized (calculation of the integral) for each iteration, which can be a very slow procedure if we don't have an analytical function
- Complex fits take several hours (or days)! (example with 2M events, 4 variables, ~100 free parameters, like Dalitz plot analysis of *D* mesons)
  - Usually you have to run several fits for your tests

Parallelization

- RooFit (Maximum Likelihood fit package) implements the possibility to split the likelihood calculation over different threads (point 2)
  - Likelihood calculation is done on sub-samples
  - Then the results are collected and summed
  - You gain a lot using multi-cores architecture over large data samples, scaling almost with a factor proportional to the number of threads
- However, if you have a lot of free parameters, the bottleneck become the minimization procedure (point 1)
  - Split the derivative calculation over several OpenMP/MPI processes
  - There is not an official implementation of such a algorithm, but some tests done by people in BaBar



"Trivial" splitting of the NLL function calculation for NPAR parameters over NCPU CPUs (threads): NPAR/NCPU parameters for each CPU

Example:

NPAR = 10, NCPU = 3

- CPU 1 = 4 pars
- CPU 2 = 3 pars
- CPU 3 = 3 pars

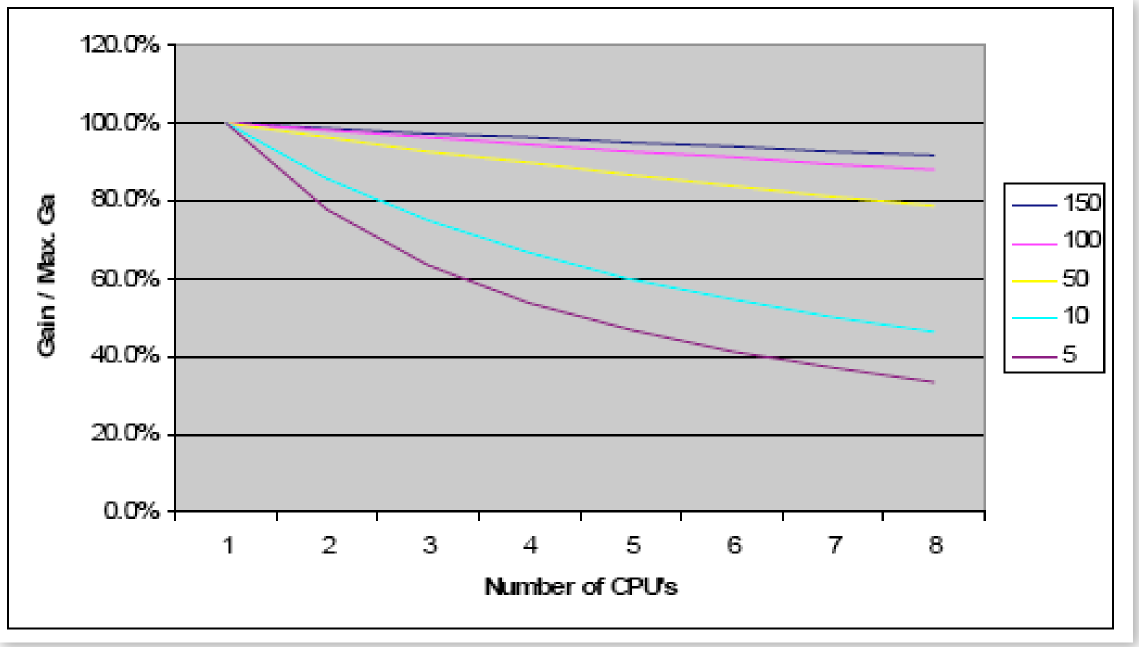
Max threads = NPAR

Parallelization

It works well in case of large number of parameters

Gain ~ NCPU*(NPAR + 2) / (NPAR + 2*NCPU)

Max. Gain = NCPU



From Brian Meadows talk at RooFit Mini Workshop @ SLAC (December 2007):
http://www.slac.stanford.edu/BFROOT/www/doc/Workshops/2007/BaBar_RooFit/Agenda.html

MINUIT2

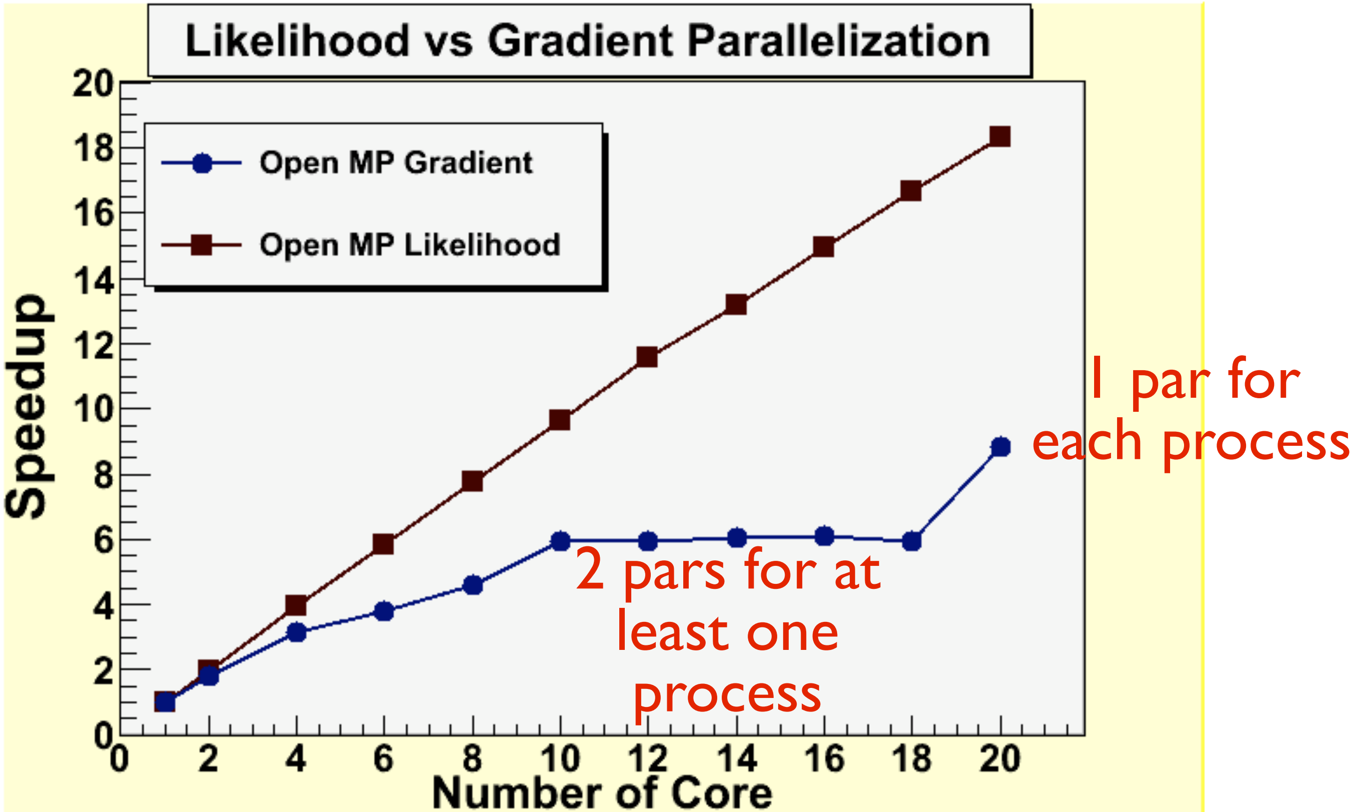
- MINUIT2 is a C++ OO version of the original MINUIT (F. James, "Minuit, Function Minimization and Error Analysis", CERN long writeup D506, 1970)
  - The most used algorithm in HEP community for Maximum Likelihood and χ^2 fits (www.cern.ch/minuit)
  - Based on the gradient method for the minimization (MIGRAD), but other algorithms implemented (SIMPLEX, SCAN)
  - Errors require Covariance Matrix calculation (HESSE)
    - More accurate calculation of the errors (to take in account non-linearities) require several steps of minimization (MINOS): VERY SLOW, non always used
- Used latest version in this work (ROOT >=5.20)
- Two strategies for the parallelization of the Minimization and NLL calculation:
  1. Minimization **or** NLL calculation on the same multi-cores node (OpenMP, pthreads)
  2. Minimization process on different nodes (MPI) **and** NLL calculation for each multi-cores node (OpenMP, pthreads): **hybrid solution**

Parallelization - OpenMP

- Shared Memory solution for
  - Parallelization of Minimization

    or
  - Parallelization of the NLL
- Data are in common, need special copy constructor
- Test done with 20 free parameters, 50 Gaussian uncorrelated variables, 10K events
- Running on 24 cores with 64Gb shared memory @ CERN (thanks to OpenLab people!)

Parallelization - OpenMP



- Better response in case on NLL parallelization
  - Note that Minimization process has more sequential part, so result as expected (Remember: it scales better with more parameters)
- In this specific case there is an analytical integral, no numerical integral: fast calculation of the NLL

Parallelization - MPI

- Work done using resources at CINECA HPC Center in Italy (Bologna): https://hpc.cineca.it/

System Architecture

```
Model: IBM BCX/5120
Architecture: eServer e326 Cluster Opteron
Processor Type: Opteron Dual Core 2.6 GHz
Number of Nodes: 1280 (4 cores per node)
Number of Processors/cores: 2560/5120
Memory: 8 GB/node
Internal Network: Infiniband (5Gb/s)
Disk Space: 100 TB + SAN
Operating System: Red Hat RHEL4
Peak Performance: 26.6 TFlop/s
Available compilers: Fortran F90, C, C++
Parallel libraries: MPI, OpenMP
```

Details -- Ctor

- Added a new class in the MINUIT2 package which deals with MPI: MPIProcess
- Inserted define directives to switch off MPI at level of compilation

MPI

```
unsigned int MPIProcess::_size = 1; // number max of threads
unsigned int MPIProcess::_rank = 0; // local rank

MPIProcess::MPIProcess(int npars) :
  _npars(npars)
{
#ifdef MPIPROC

  if (!(MPI::Is_initialized())) { // ask if MPI is already in place
    MPI::Init();
    std::cout << "MPIProcess:: Start MPI on #"
              << MPI::COMM_WORLD.Get_rank() << " processor"
              << std::endl;
  }

  _size = MPI::COMM_WORLD.Get_size();
  _rank = MPI::COMM_WORLD.Get_rank();
#endif

  if (_rank>_npars) {
    std::cerr << "Error: more processors than parameters!" << std::endl;
    exit(-1);
  }

  _numPars4JobIn = _npars/_size; // number of parameters per node
  _numPars4JobOut = _npars%_size; // number of parameters to redistribute

}
```

Pars Distribution per Node

- Results for derivatives organized in a vector of doubles
- Index of the vector per node:

```
inline unsigned int NumPars4JobIn() const { return _numPars4JobIn; }
inline unsigned int NumPars4JobOut() const { return _numPars4JobOut; }

// Number of pars per node
inline unsigned int NumPars4Job(unsigned int rank = _rank) const
{ return NumPars4JobIn()+((rank<NumPars4JobOut()) ? 1 : 0); }

// First index of pars vector
inline unsigned int StartParIndex() const
{ return ((_rank<NumPars4JobOut()) ? (_rank*NumPars4Job()) :
          (_npars-(_size-_rank)*NumPars4Job())); }

// Last index of pars vector
inline unsigned int EndParIndex() const
{ return StartParIndex()+NumPars4Job(); }
```

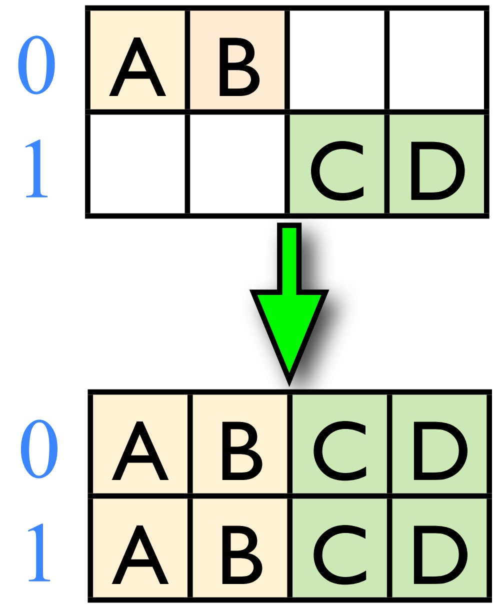
Synchronization

- Each node calculates his group of derivatives
- Each node scatters the results to all other nodes
  - At the end all nodes have the results for all derivatives

```
void MPIProcess::MPISyncVector(double *ivector, int svector, double *ovector)
// ivector --> elements to scatter
// svector --> number of elements to scatter
// ovector --> all elements gathered
{
  // Required by the Allgatherv method
  int offsets[_size];
  int nconts[_size];
  nconts[0] = NumPars4Job(0);
  offsets[0] = 0;
  for (unsigned int i = 1; i<_size; i++) {
    nconts[i] = NumPars4Job(i);
    offsets[i] = nconts[i-1] + offsets[i-1];
  }
  //

  MPI::COMM_WORLD.Allgatherv(ivector,svector,MPI::DOUBLE,
                             ovector,nconts,offsets,MPI::DOUBLE);

}
```



Call in MINUIT2

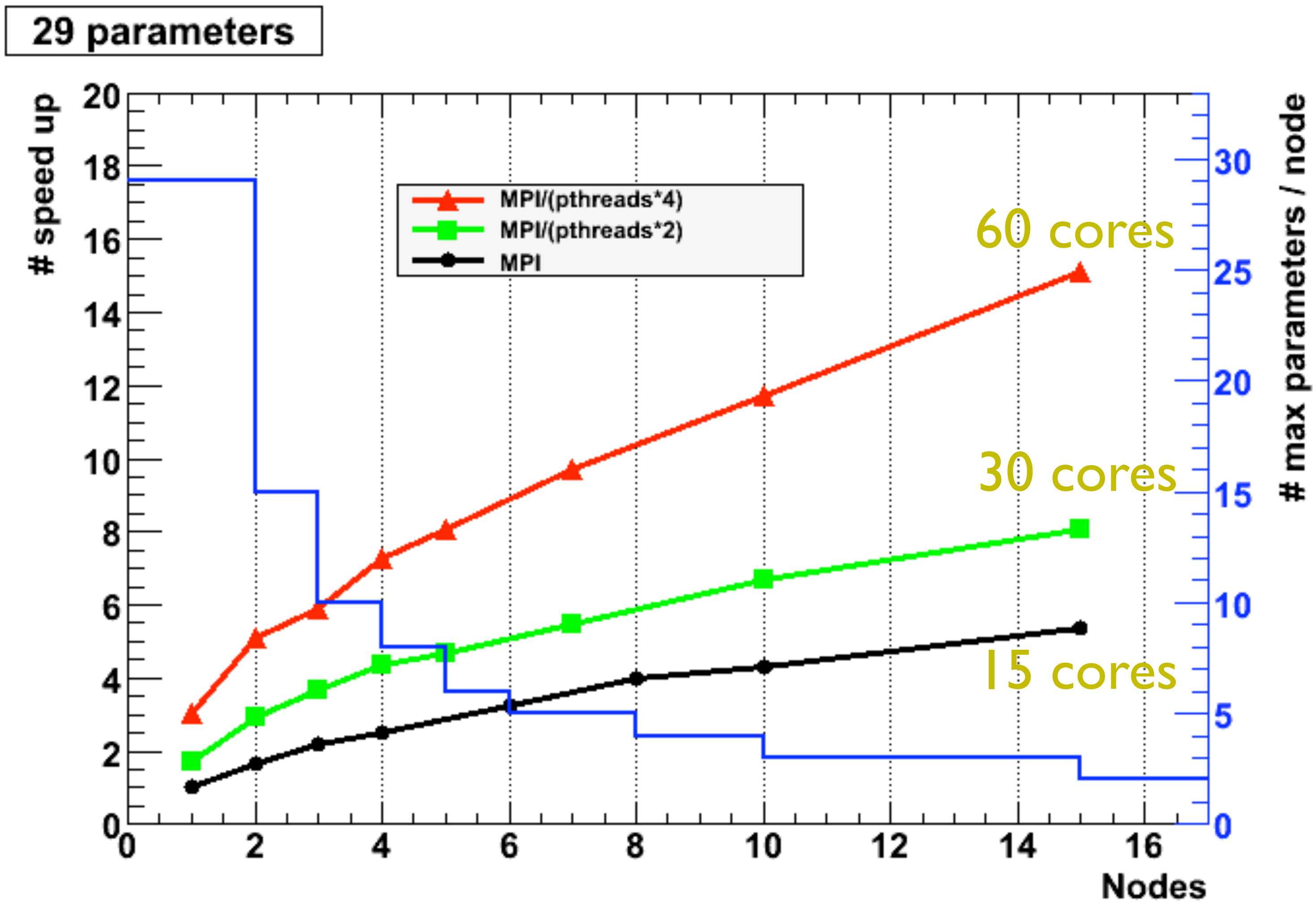
- MPIProcess declared in the class for Gradient calculation (Numerical2PGradientCalculator)
- Each node calculates his group of derivatives

```
// n --> total number parameters
MPIProcess mpiproc(n);

// loop for derivate calculations (calls to NLL function)
// i --> index od the
for (unsigned int i = mpiproc.StartParIndex(); i<mpiproc.EndParIndex(); i++) {
  // i-th derivate calculation...
```

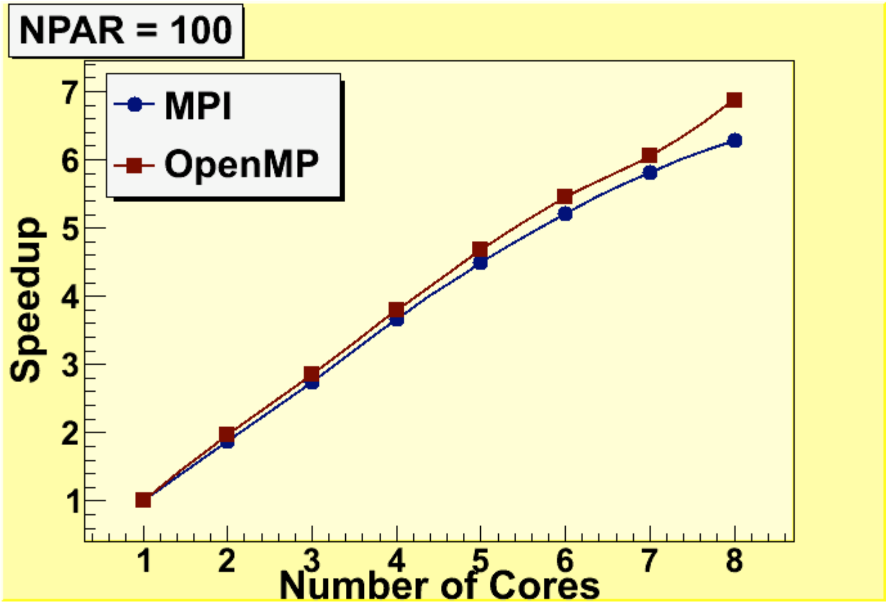
- Synchronization: At end each node has all derivative results and can do the remaining part of the (serial) code
  - all nodes do the same serial part
- The full procedure is repeated for each step in the minimization processes

Parallelization - Example



MPI vs OpenMP

- Test done with 100 free parameters, 50 Gaussian uncorrelated variables, 10K events, on a 8 cores machine



- Reasonable results: MPI needs some time for the initialization (no thread-safe)
  - no a good solution for a multi-core machine

Parallelization - To do

- In some case the bottleneck is the integral calculation (in Dalitz plot analysis we have integral in several variables, which is very slow to compute)
  - Normalization Integral to be calculated for before each step on the minimization process: VERY slow procedure with complicated integrals
  - There is not parallel implementation of the normalization integral calculation
  - Needs a general infrastructure in RooFit
- Possibility to split also the likelihood calculation using MPI (partial implementation using a cartesian topology):
  - MPI for NLL calculation (in case of many events) and MPI for Minimization (of course if you have enought resorcuses):
    - Split in 2 processes for Minimizzation, each one in 20 processes for NLL calculation: 40 processes

Conclusions

- Work is ongoing for NLL parallelization:
  - require changes in RooFit and MINUIT2
  - Some checks to do before releasing a first version of the code
    - Next MINUIT2 version (ROOT 5.22) will contain the possibility to use OpenMP
  - Other possible improvements to be included
- HEP is entering in the precise measurements era:
  - Huge quantity of data available (BaBar, Belle)
  - More data will be available soon from LHC experiments
- Most of the data analysis techniques are very CPU-time consuming
  - We can benefit using parallel version of the code
  - In most of the case easy to implement
  - Parallel and more parallel for the future (see tomorrow Sverre's talk)

Backup Slides