Parallelization Opportunities in Math Libraries

Forum on Concurrent Programming Models and Frameworks,

Wednesday 1/02/2012

Lorenzo Moneta

Problem Domain

- Mathematical libraries
  - mathematical functions
  - geometry (3D) and physics vectors (4D)
  - matrix and vector classes and linear algebra
  - random number generations
  - Numerical algorithms
    - numerical integration
    - minimization
- Statistics libraries
  - fitting (parameter estimation)
  - Toy MC generations (sampling from probability distribution)
  - multivariate analysis tools
  - advance tools for interval (limit) and significance estimation

- *What are the most important applications using these libraries ?*
- *Which opportunities do we have to parallelize the libraries to speed up these applications ?*

Data Analysis Applications

- Statistical techniques all based on the likelihood function
  - each event is described by a probability density function (PDF)

$$P(x|\theta)$$

Likelihood: $$L(x|\theta) = \prod_i P(x_i|\theta)$$

  - all methods require evaluation of the likelihood
    - parameter estimation (maximum likelihood fit)
    - interval estimation (e.g. mass limits in particle searches)
    - hypothesis tests (significance of discovery for new particles)
  - Bayesian Methods:
    - based on integrating the likelihood

$$\int L(x|\mu,\nu)\Pi(\mu,\nu)d\nu$$

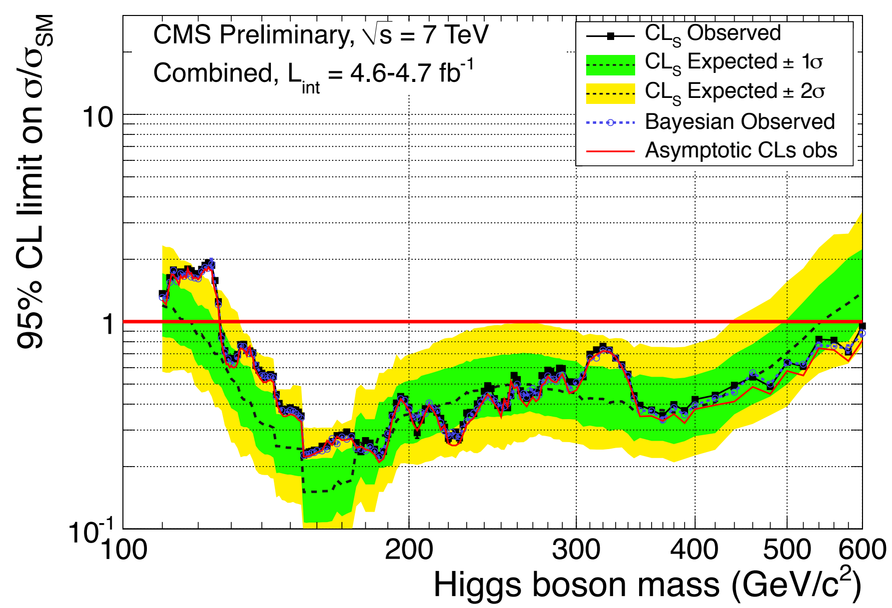
  - Frequentist methods
    - require distribution of a test statistic
      - e.g. profiled likelihood ratio

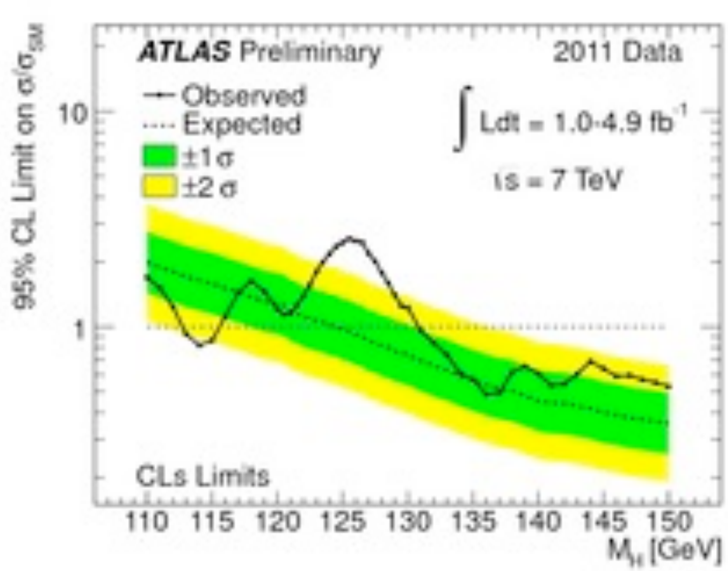
$$\lambda(\mu) = \frac{L(x|\mu,\hat{\hat{\nu}})}{L(x|\hat{\mu},\hat{\nu})}$$

    - require repeated generation and fitting of pseudo data (toys)

Data Analysis Parallelization

- Use typically RooFit for building complex PDF and RooStats for running statistical analysis
  - models with many PDF, many observables and a lot of parameters
    - e.g. Higgs combination (more than 200 parameters and several channels)
- Possible various level of parallelizations:
  - PDF evaluation
  - Loop on events for computing log-likelihood
  - Algorithms (e.g Minuit) require multiple likelihood evaluations
  - Loop on toy data analysis (on various likelihood minimization)
  - Repetition of same analysis on different inputs (analysis points)





Parallelization of Minuit

- Parallelization of MIGRAD algorithm (presented ACAT 2008)
  A. Lazzaro & L.M.
  - Each Migrad iteration consists of:
    - computing function value and gradient to find Newton direction
    - computing step by searching for minimum along the Newton direction
    - if satisfactory improve calculation of Hessian matrix, *H*
    - invert to get new matrix $V = H^{-1}$
    - repeat iteration until expected distance from minimum smaller than tolerance
  - In case of many parameters (> 10) and complex function evaluation, gradient calculation dominates the process:

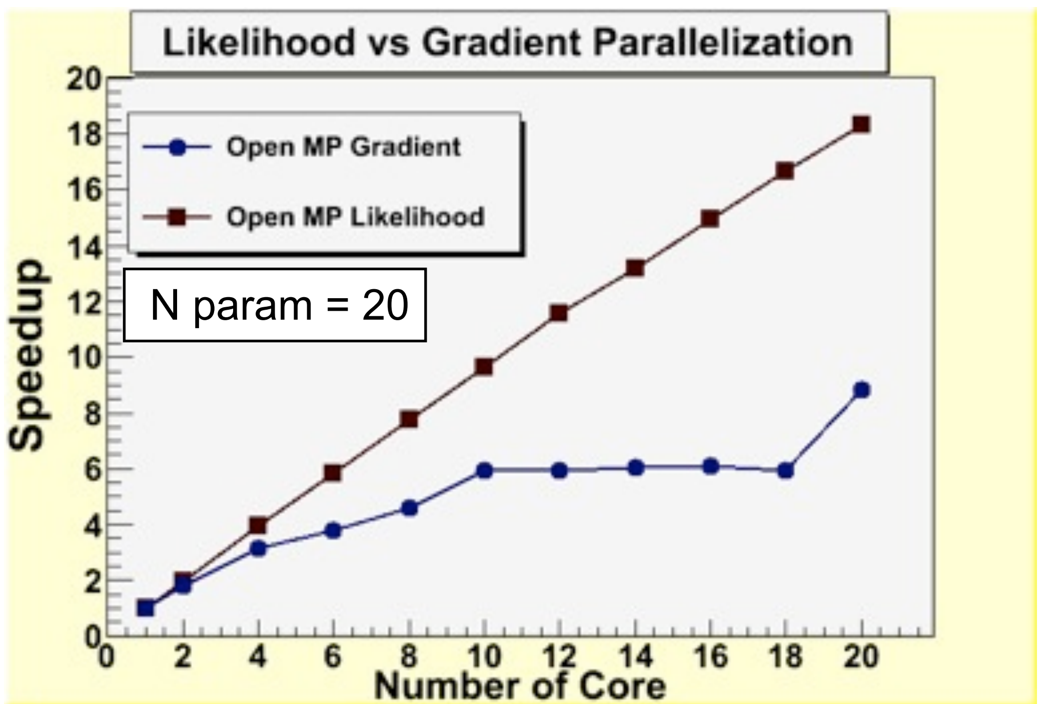
$$\nabla_i(x) = \frac{\partial f}{\partial x_i} \approx \frac{f(x_i + \delta x_i) - f(x_i - \delta x_i)}{2\delta x_i}$$

    - al least 2 * NDIM function evaluation are needed
  - Parallelize calculations by using a thread for each partial derivative
  - Use OpenMP (multi-thread) or MPI (multi-processes)
  - Available in ROOT for Minuit2 since version 5.22 (3 years ago)

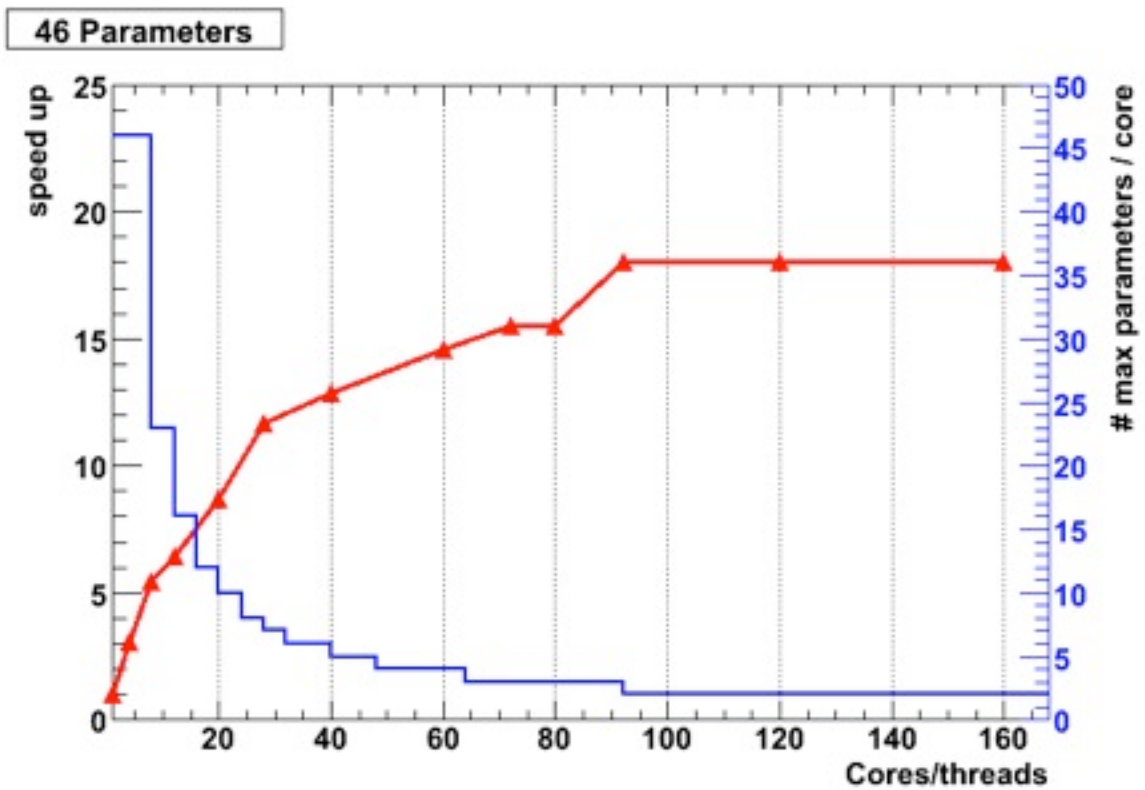
Minuit parallelization

- Minuit parallelization is independent of user code
  - requires thread safety user code to evaluate the likelihood function when using OpenMP
- Examples:

unbinned fit with 20 parameters using openMP



complex BaBar fitting by A. Lazzaro and parallelized using MPI



- Log-likelihood parallelization (splitting the sum) is more efficient but requires the user to change its code

RooFit/RooStats Parallelization

- RooFit supports parallelization in evaluating log-likelihood function
  - multi-process parallelization
  - use fork to parallelize likelihood on multi-processes
    - `pdf->fitTo( data, NumCPU(8) );`
  - Support also for PROOF and PROOFLite
    - for multiple likelihood fits (e.g. for toy studies, goodness of fits, etc.)
- RooStats: parallelization of toys (generation and fitting) using PROOF
  - loop on toys to obtain test statistic distributions
  - results from each toy (ROOT object) are automatically merged and returned to the user
  - PROOFLite is very convenient to use on user desktops
  - tested also on large clusters (ATLAS)
  - memory can start to be a problem with very large models and many cores
- Trivial parallelization performed at job level
  - run several jobs on Grid or on cluster each with a small number of toys
  - RooStats provides the tools for merging results but user still needs to do it
  - most common usage of RooStats for complex analysis

Likelihood Parallelization

- Study Likelihood parallelization in detail (A. Lazzaro, Openlab)

- Recalling the NLL definition

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

① Each $\mathcal{P}$ (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

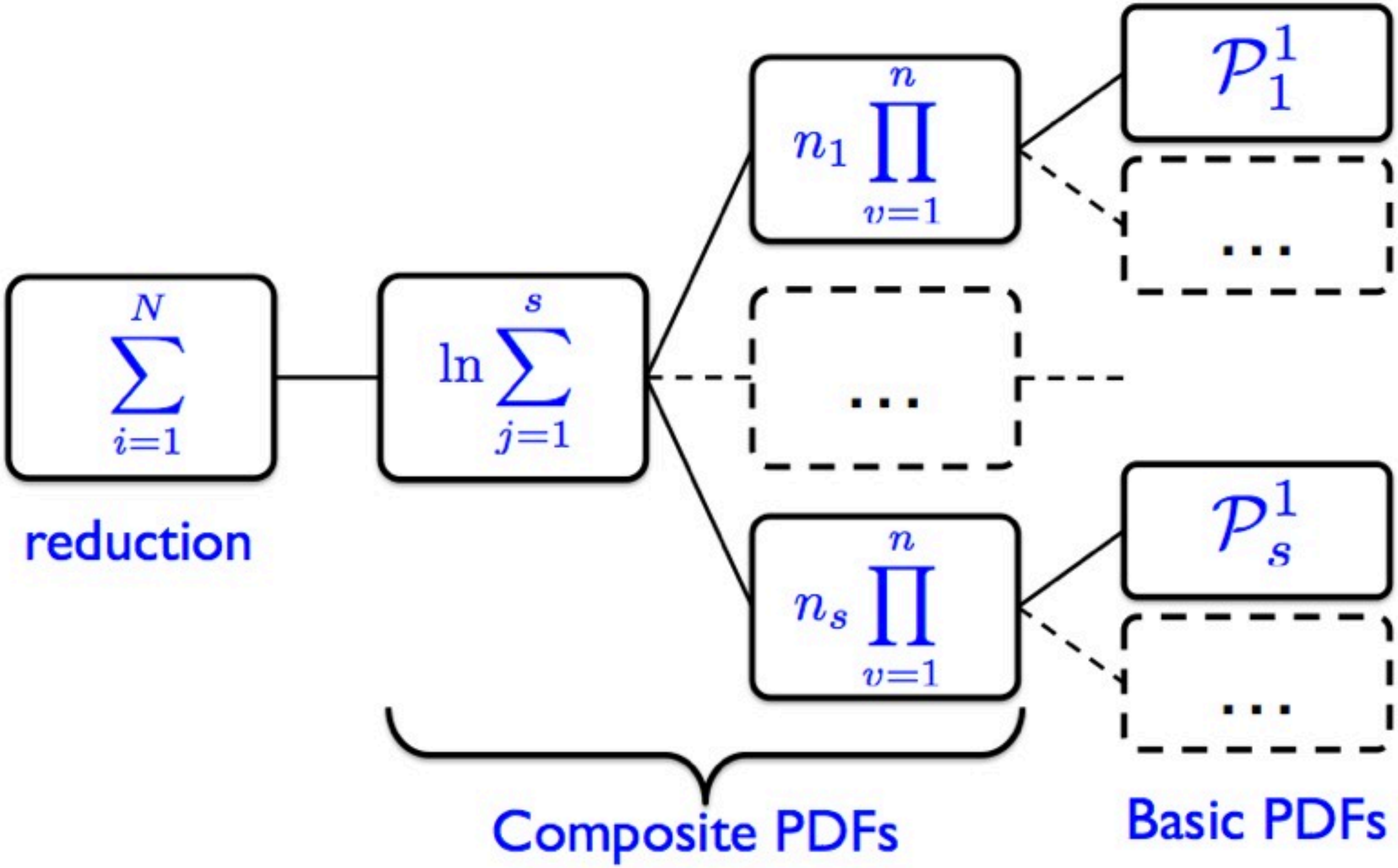
② Product over all observables (composite PDF)

③ Sum over all species (composite PDF)

④ Reduction of all values

Log-Likelihood Evaluation

- We can visualize the NLL evaluation as a tree

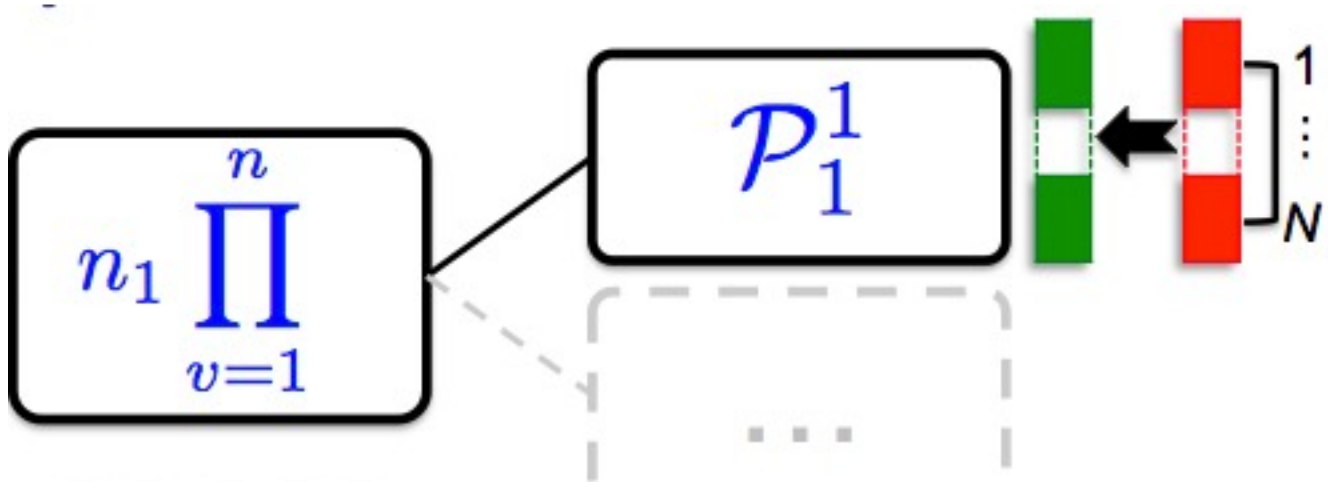


Possible various level of parallelization

Vectorization of PDF's

- Organize data (observables) as vectors
- Evaluate PDF not on a single observables but on vector of observables

$$P_i = P(x_i|\theta) \Longrightarrow \overrightarrow{P}(\overrightarrow{x}|\theta)$$



  - data vector is read-only during evaluations
  - evaluate vector of pdf results traversing the tree

- Allows SIMD vectorization during the pdf evaluation

Openlab Prototype

- Studied parallelization at various levels using multi-threads
  - CPU with OpenMP
  - GPU with CUDA or OPENCL
  - hybrid setup to optimize CPU/GPU load with OpenCL
- Levels:
  - parallelize loop on the single PDF evaluation of the observables
  - parallelize outer loop for summing the final result
- Try to have minimal change in RooFit code
  - see various Openlab presentations and reports

Openlab Prototype Findings

- Inner loop parallelization:
  - small memory footprint and better for race conditions
  - suffer from OpenMP overhead in having multiple parallel regions
  - require manage a large number of arrays with the evaluation results
  - cache problems when evaluating composite PDF's
    - much better scalability when using processors with larger cache
  - GPU -> CPU communication problems for summing final results

- Outer loop parallelization:
  - better scalability
  - suffer from race conditions
  - more difficult to implement, it requires more changes in original code
    - developed prototype has many changes and is difficult to port in RooFit production code

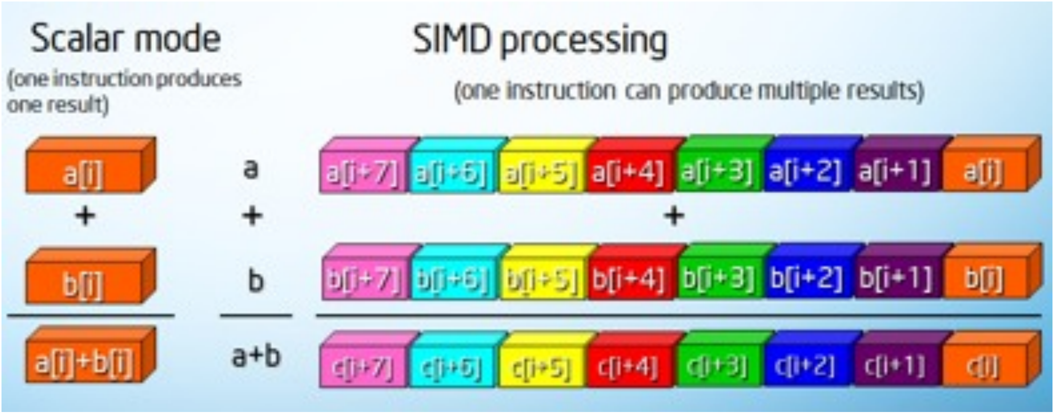
Conclusions on Likelihood Parallelizations

- Lesson learnt:
  - PDF vector evaluation is promising and should be further investigated
  - importance of optimizing and redesigning code to have good scalability for many threads
    - this will result also in a faster scalar version of the code
  - need to work in close collaboration with the RooFit author for an optimal solution

- Important note:
  - Numerical precision problems when evaluating final log-likelihood sum from the thread-results
    - need appropriate algorithm (e.g. Kahan summation) which minimizes evaluation errors

Vectorization

- Another parallelization dimension
  - the vector processing using SIMD (*Single Instruction Multiple Data*)
- Perform numerical operations in parallel
  - size of registers depending on architectures
    - SSE : 128 bits : 2 double's or 4 float's
    - AVX: 256bit : 4 double's or 8 float's)



- Compilers can try to perform auto-vectorization of loops
  - require data organize in vectors and iteration independence
  - branches (if statement) can break vectorization
  - new compilers (e.g. gcc 4.6) are much better
- Can use special instructions for processors (intrinsic)
  - SSE or AVX instructions
- Libraries exist to hide this complexity to user
  - e.g. Vc library

Vc Library

- C++ library developed by *M. Kretz (and V. Linderstruth)* to ease vectorization
- Used in processing tracks in ALICE L3 trigger (Kalman filter) with very good results
  - See http://code.compeng.uni-frankfurt.de/projects/vc/
- portable across compilers and architectures
  - application written in Vc can be compiled for SSE, AVX and scalar case
- Vc provides new vector types:
  - `Vc::float_v` or `Vc::double_v`
  - `float_v::Size` will depend on architecture (e.g 8 on AVX)
  - basic operations (+,-,/,*) for these types are supported
  - also basic Math and transcendental functions (sin,cos, log,etc..)
    - exponential is not yet implemented
- User can vectorize code without need to use and know intrinsic instructions

Vc Code Example

Listing 5.1 Scalar Code

```
void func(float *v, int N, float s, float b, float t)
{
  for(int i = 0; i < N; ++i) {
    v[i] = v[i] > t ? 0.5f * s * (v[i] + t) + b
                    : s * v[i] + b;
  }
}
```

Listing 5.2 Vc Code

```
void func(float *v, int N, float s, float b, float t)
{
  const float_v c = 0.5f * s * t + b;
  for(int i = 0; i < N; i += float_v::Size) {
    float_v v1(&v[i]);
    float_v v2 = s * v1 + b;
    v2(v1 > t) = 0.5f * s * v1 + c;
    v2.store(&v[i]);
  }
}
```

from M. Kretz diploma thesis

Vc Code Example (2)

- Same code using intrinsic for SSE

Listing 5.3 SSE Intrinsics Code

```
void func(float *v, int N, float s, float b, float t)
{
  const __m128 c = _mm_set1_ps(0.5f * s * t + b)
  __m128 v1, v2, mask;
  for(int i = 0; i < N; i += 4) {
    v1   = _mm_load_ps(&v[i]);
    v2   = _mm_mul_ps(_mm_set1_ps(s), v1);
    v2   = _mm_add_ps(v2, _mm_set1_ps(b));
    mask = _mm_cmpnle_ps(v1, _mm_set1_ps(t));
    v1   = _mm_mul_ps(_mm_set1_ps(0.5f * s), v1);
    v1   = _mm_add_ps(v1, c);
    v2   = _mm_blendv_ps(v2, v1, mask);
    _mm_store_ps(&v[i], v2);
  }
}
```

Vc Evaluation

- An initial evaluation performed 2 years ago
- Try to use Vc as a template argument in the ROOT matrix and vector libraries
  - `SMatrix<double_v, N>` , `SVector<double_v, N>`
  - also tried in Physics and Geometrid vectors (e.g. `LorentzVector`)
  - when looping on set of vector or matrices, loop size reduced by the size of the Vc type (`NITER = NITER / double_v::Size`)
    - example: Kalman filter equations for updating error matrix
- Some tests show promising results, but in some cases no significative improvement found
  - explained as some compiler limitation at that time (gcc 4.4 was used)
- Would be interesting to try with new compiler versions
- Should to continue evaluation and maybe provide as a possible library to use either inside ROOT or externally ?
  - Matrix and Vector libraries should be able to profit from it
- Can be useful for reconstruction or simulation applications

Vectorization Activities in CMS

- Activities by *D. Piparo and V. Innocente (CMS)*
- New fast implementation of transcendental functions using Cephes (an old C library)
- Make code in a way that can be auto-vectorized by compiler
  - no need to use intrinsic
  - provide API to pass arrays of values instead a single one
    - `double exp(double x)` ⇒ `void exp_vect( const double *, double *, int)`
- Promising results obtained
  - use latest compiler version 4.7
  - good speed-up and also the numerical results are at the expected precision
- Functions are being included in CMS SW framework
- Consider to include these functions at some point in ROOT for common usage ?
  - require latest compiler versions for vectorization and use also C++OX
- CMS is also trying to parallelize (using OpenCL) some heavy used SMatrix operations (e.g. similarity $A^T B A$)
  - see presentation of T. Hauth at one of the last meetings

Random Numbers

- Parallelization of pseudo-random numbers generators
  - most used generator are very fast (RanLux is maybe the exception)
  - time in generating random numbers is often not critical in majority of our applications
    - one does much more time consuming things with a random number
- Using the random numbers in parallel application is more problematic
  - many good generators have a very large state
    - e.g. Mersenne and Twister (TRandom3) has state of 624 words (32 bits)
    - This makes them problematic to run on GPU
      - see work from F. Carminati and others presented at ACAT 2011
  - problem in seeding many independent sequences and in bookkeeping them
    - need generator with very long periods, which normally can be obtained only with large states
      - need care in seeding the generators to have really independent states
    - or dedicated parallel generators which allow to jump in the sequence
      - need to know in advance max length of each stream

New Parallel Random Numbers

- New PRNG based on counters without a state (J. Salmon et al.)
  - based on a counter **n** and key **k**
    - $k : x_n = f_k(n)$
  - instead of an iterative sequence
    - $x_i \rightarrow x_{i+1} = f(x_i)$
  - no state (can be easily used in parallel applications)
  - generators derived from algorithms used in cryptography
  - awarded best paper at the SC11 conference
- These new generators pass the most stringent tests
  - *BigCrush* of TestU01 from L'Ecuyer
- but are empirical generators (lack of mathematical analysis)
  - very complex algorithm
- interesting to watch this new development

Summary

- Parallelization in tools for data analysis and concentrated on likelihood evaluation (fitting)
  - most time consuming tasks and immediate benefit for end-users
  - other analysis tools (e.g. multi-variate tools) would benefit as well from same code optimization
  - very useful findings from prototype developed by Openlab
  - opportunity to work on optimize and parallelize algorithms at the same time
- Whenever possible, a parallelized version of an algorithm should be provided
  - Example of Minuit. Parallel version can be used without changing user code
- Need to improve also thread safety of existing code
- Started investigation of parallelization in vector and matrix operations (reconstruction or simulation applications)
  - vectorization looks promising
- Random number generators for parallel applications
- Other parallelization opportunities exist but less relevant in HEP
  - e.g. parallelization of large linear algebra systems

References

- OpenLab parallelization studies:
  - Various reports, latest ones:
    - S. Jarp *et al.*, *Parallel Likelihood Function Evaluation on Heterogeneous Many-core Systems*, proceeding of International Conference on Parallel Computing, Ghent, Belgium, 2011. EPRINT: CERN-IT-2011-012
    - S. Jarp *et al.*, *Parallel Likelihood fits with OpenMP and CUDA*, Journal of Physics: Conference Series EPRINT: CERN-IT-2011-009
- Vc
  - http://code.compeng.uni-frankfurt.de/projects/vc/
  - *M. Kretz, V. Lindenstruth, Vc, a C++ Library for explicit vectorization*
  - *M. Kretz, Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading, Diplomarbeit (2009)*
- Pseudo-Random Number Generators based on counters
  - *http://www.thesalmons.org/john/random123/papers/random123sc11.pdf*