A study on implementing a multithreaded version of the SIRENE detector simulation software for high energy neutrinos

Petros Giannakopoulos (b), Michail Gkoumas (b), Ioannis Diplas (b) ,Georgios Voularinos (b),
Konstantia Balasi (a), Katerina Tzamarioudaki (a), Christos Filippidis (a,b),
Yiannis Cotronis (b), Christos Markou (a)

a:NCSR Demokritos, b: University of Athens

SIRENE

- Is a program that simulates the detector response to muons and showers.
- It is based on the formalism of the probability density function (PDF) of the arrival time of light.
- It uses the muon energy loss cross sections by P. Kooijman.

Goal: Development of a multithreaded version of the SIRENE detector simulation software for high energy neutrinos

- Allow utilization of multiple CPU cores and GPUs → Can lead to a potentially significant decrease in the required execution time compared to the sequential code.

- Making use of parallel frameworks:
  - MPI, OpenMP → production of multithreaded CPU code
  - CUDA → leveraging the processing power implicating the GPU

Processing steps:

1. Read Event from MonteCarloEventWriter output file
2. Remove existing hits
3. Propagate muon(s)
   - simulate energy loss and EM-showers
   - generate hits (direct and single scattered light)
4. Process shower particles from primary vertex
   - generate hits (direct and single scattered light)
5. Merge hits (to speed-up TriggerEfficiency)
   - DT_{max} typically 15 ps
6. Write Event to MonteCarloEventWriter compatible output file

---

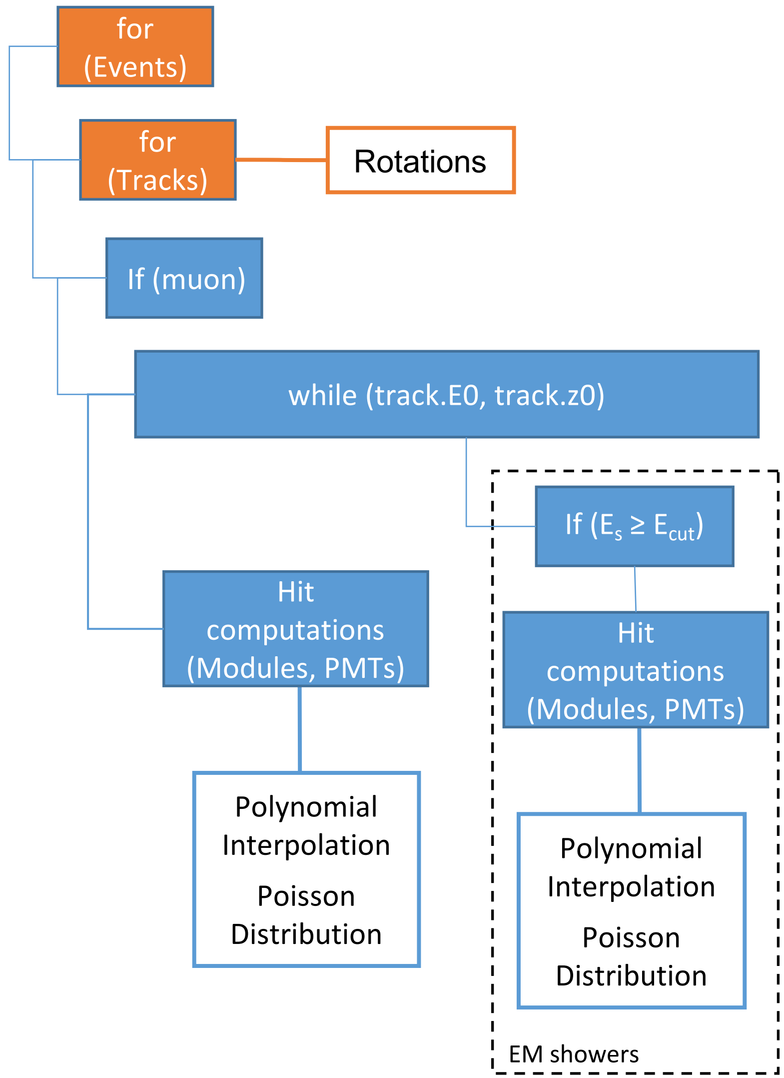
¶ Elapsed time steps 1–6 ≤ 15 min. per file with about 36,000 event

SIRENE: Overview of original implementation (1/2)

SIRENE implements a number of nested loops :

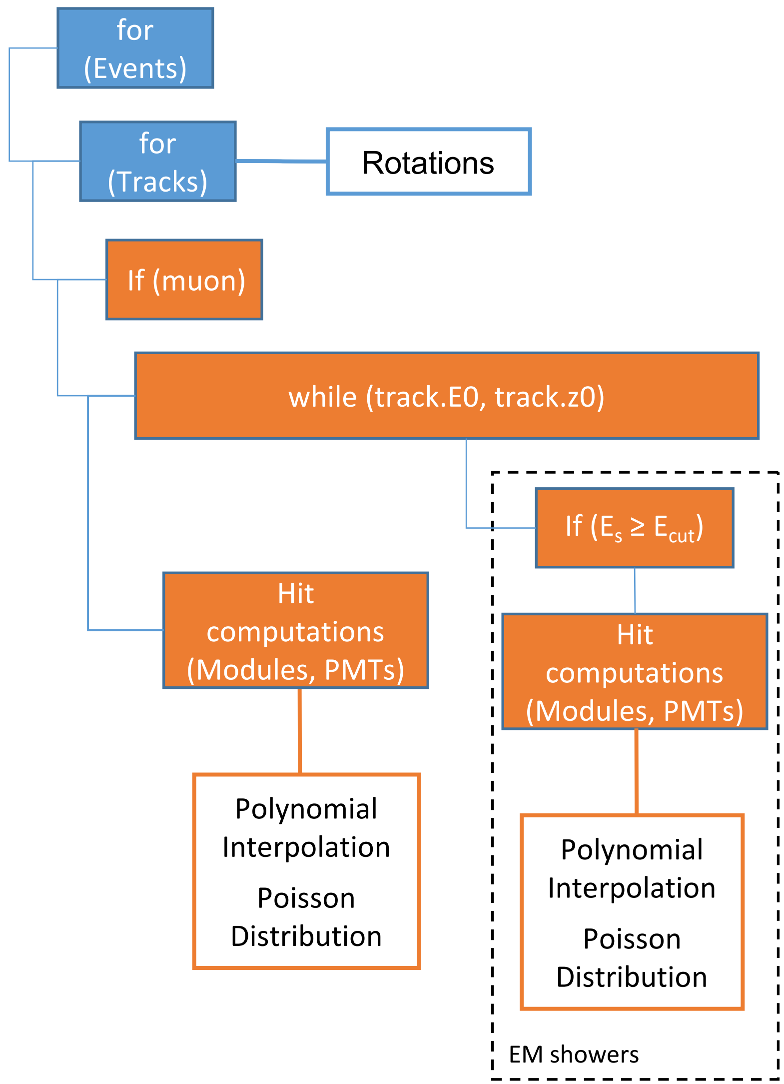
- Events
  - Consist of a # of particles described as tracks.
  - Events are independent of each other.

  - Tracks
    - Describes path of each particle:
      - •Propagation and hit computations.
      - •Rotation of each track's coordinate system.



Overview of original implementation (2/2)

- Tracks
  - Only muon tracks are considered.
  - Energy loss and propagation calculated in steps on each iteration.
  - Hit computations for muons/EM showers (direct/scattered):
    - Polynomial Interpolation for calculating photon emissions from the Cumulative Density Function (CDF)
    - Poisson Distribution for calculating the expectation values of the number of photo-electrons on a Module and PMTs



SIRENE: Overview of parallel implementation (1/7)

- The most time consuming original code segments are:

  I. Coordinate system transformations for Tracks and PMTs

  II. Calculations for muon propagation and hit probabilities on PMTs for each Track (main loop)
     - i. Muon propagation in steps, with energy loss and z-axis position calculation on each step
     - ii. Integration of CDF through polynomial interpolation method
     - iii. Poisson distribution

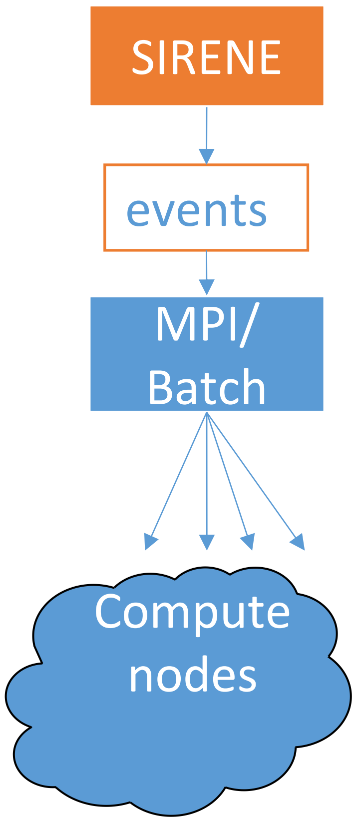
  III. PDFs to CDFs conversion

Overview of parallel implementation (2/7)

❑ MPI/Batch for Events

- million scale
- are independent of each other
- do not exchange data.
- Event computation is an embarrassingly parallel problem.

- Solution: MPI/Batch

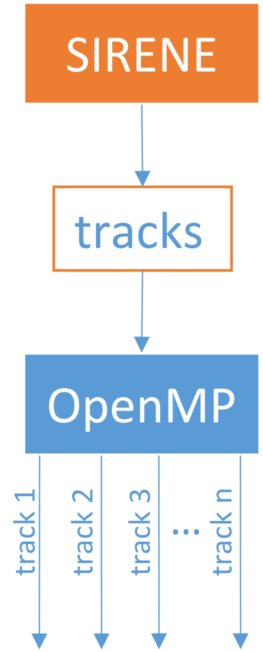


Overview of parallel implementation (3/7)

❑ OpenMP for Tracks

OpenMP could be used for the parallel processing of tracks:

- Tens of tracks compose one event.
- Output is saved in a shared object (Event)
  - One 'machine' should be involved, otherwise a different implementation would require a great deal of overhead.



Solution :

- enlist one thread per track
- Each thread is capable of creating sub-threads as needed, according to the inner loops and the available system resources.

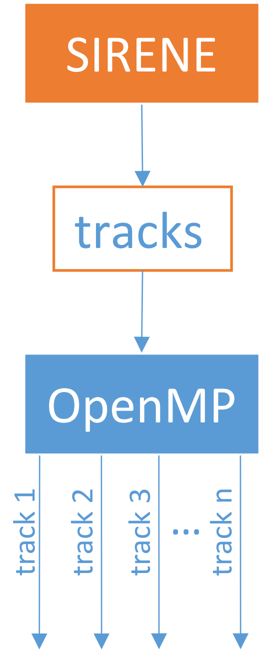
Overview of parallel implementation (4/7)

❑ OpenMP for Tracks

- INPUT:
  - 1000 Events (small)
    - ✓ 1 track per event
  - ~4000 modules (Large detector)

➢ Preliminary results:

- Tracks parallel processing:
  - Early stage implementation shows a promising degree of speedup
  - Work In Progress: Need to test with a much larger input and validation of results with those produced from sequential code



❑ OpenMP applied to PDF → CDF conversion

➢ Results: ~3.8x speedup with 4 cores/threads

Overview of parallel implementation (5/7)

❑ CUDA for suitable functions

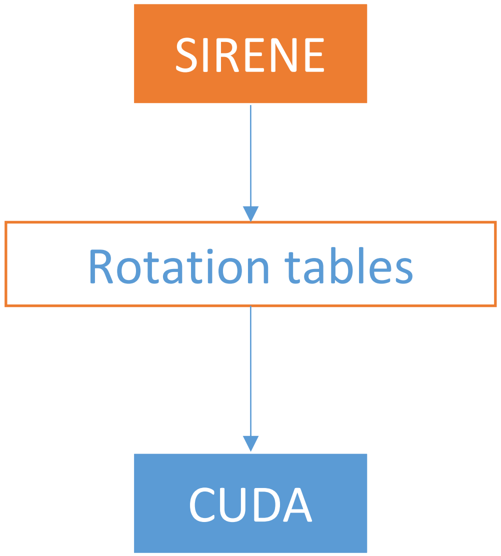
1. The rotation of the coordinate system

Implementation method: straight parallelization applied to the existing sequential algorithm:

I. Calculate cos(θ), sin(θ), cos(ϕ), sin(ϕ)
   - θ, ϕ: track direction

II. Build rotation matrix R

III. Rotate track coordinate system

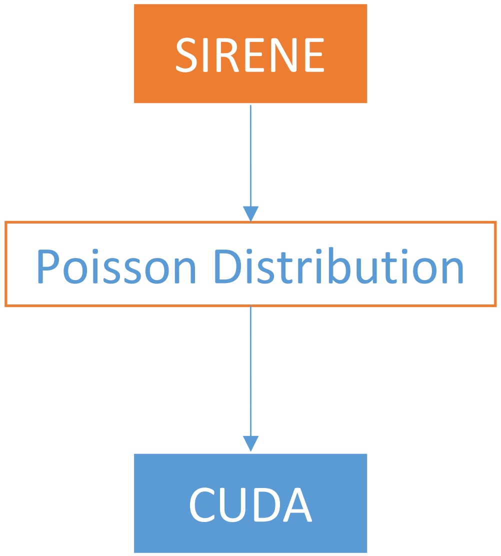


➢ Results: 3x-4x speedup.

Overview of parallel implementation (6/7)

❑ CUDA for suitable functions

2. The Poisson distribution random number generator



Implementation method: look into alternate parallel versions of the random number generation (RNG) algorithm:

- Use of the NVIDIA CUDA Random Number Generation library (cuRAND):
  - Provides Merseinne Twister RNG algorithm. Also used in TRandom.
  - Provides Poisson distribution RNG

➢ Work In Progress

Overview of parallel implementation (7/7)

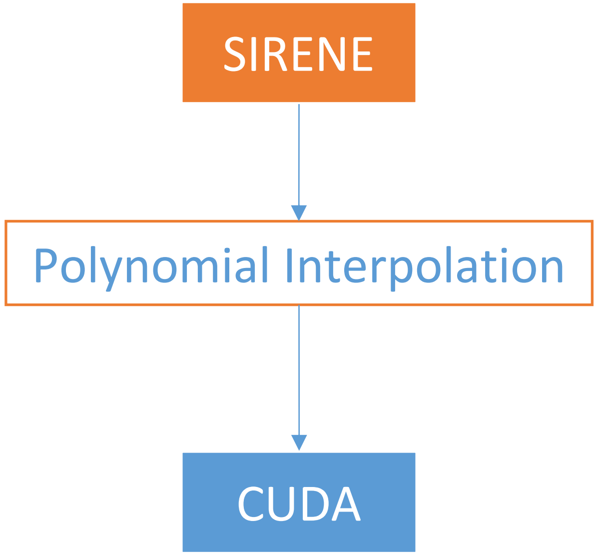
❑ CUDA for suitable functions

3. The Polynomial Interpolation

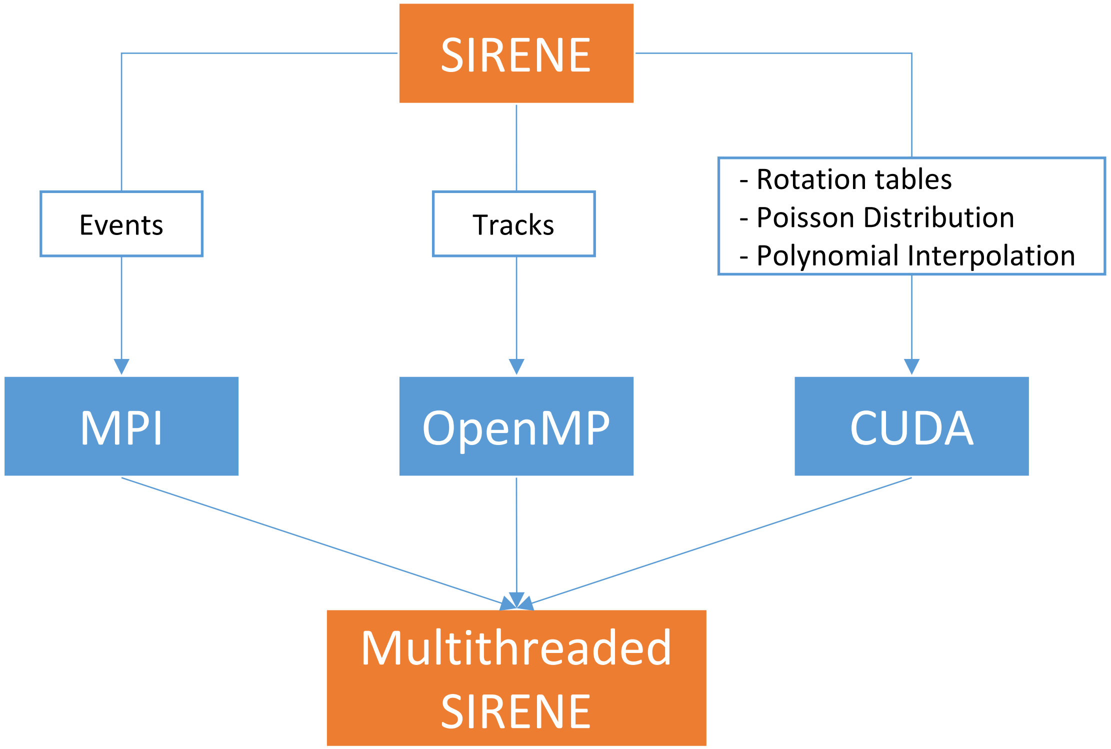
Implementation method: look into alternate parallel versions of interpolation algorithms:

- Use of fast hardware implementations such as texture-based 1D Linear interpolation and Nearest Neighbor interpolation.
- More accurate Cubic B-Spline interpolation for CUDA

➢ Work In Progress



Summary


SIRENE
Events
Tracks
- Rotation tables
- Poisson Distribution
- Polynomial Interpolation
MPI
OpenMP
CUDA
Multithreaded SIRENE

backup slides

Time :

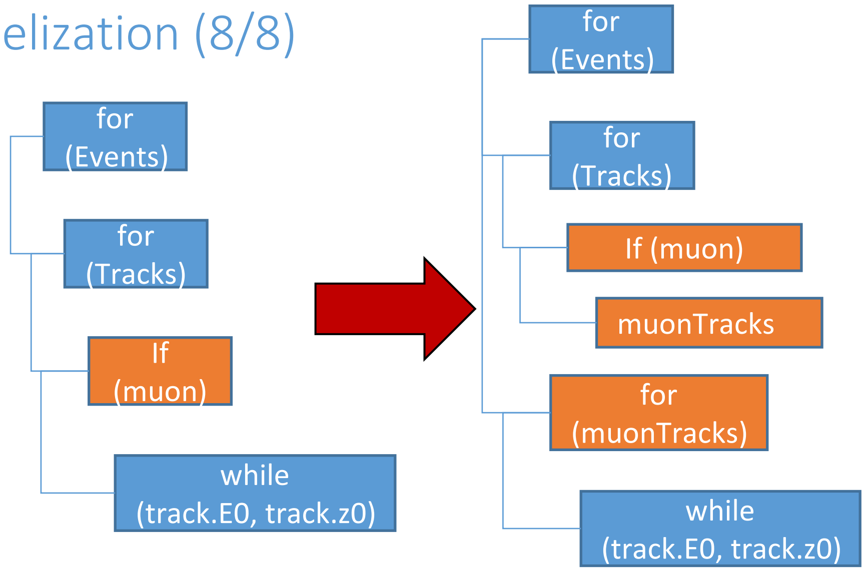
- Build CDF tables:
  - < 5min (small geometry)
  - > 1hr (big geometry)

- Event processing:
  - Rotations of coordinate systems of Tracks and PMTs: 1/3rd of total execution time
  - Propagation and hit computations

Preparation of sequential code for parallelization (8/8)

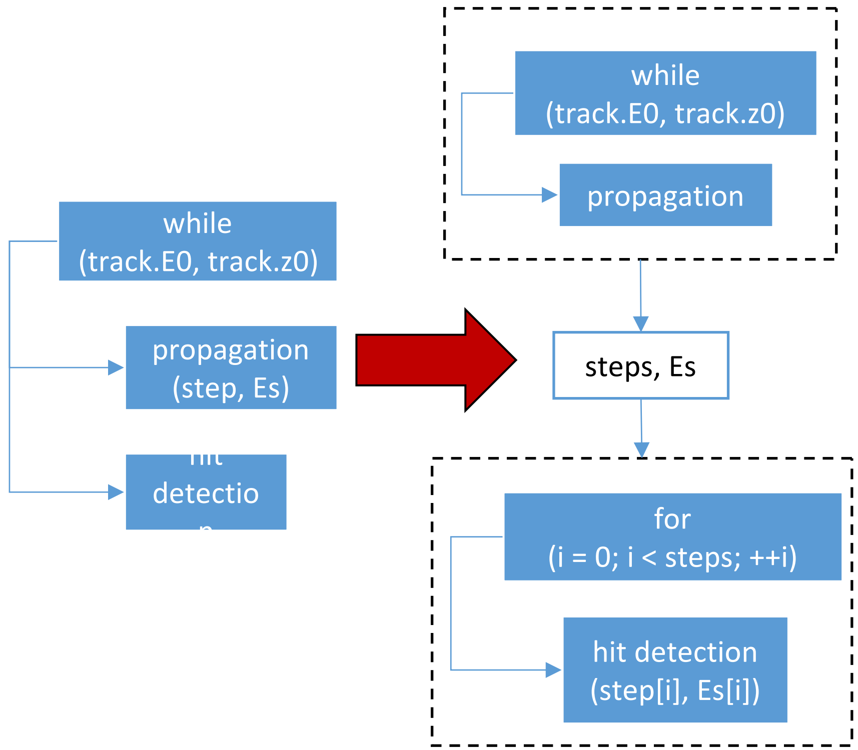
- Conditional operations are generally a manageable problem in SIRENE sequential code.

  Solution: create new lists of items that contain the use of the conditional operation to the initial list



- The 'while' loop where the particle propagation and hit detection take place
  - Most critical part of the sequential code
  - Number of iterations is not initially known

Solution: transformation into a 'for' loop by writing in a vector all the steps calculated in the while loop.



- Preparation of sequential code for parallelization

Certain code segments relying on conditional operations must be altered in order to allow parallel execution or improve its performance:

1. Move 'if' operations outside of loops targeted for parallelization. E.g. the check if a track is a muon can be moved outside the Tracks loop .
   - Less conditional operations → improved performance of parallel code

2. Transform 'while' loops with an unknown number of iterations into 'for' loops with known number of iterations. E.g. The 'while' where the particle propagation and hit detection computations take place can be transformed into a 'for' loop with a known number of steps.
   - While loops cannot be efficiently parallelized. For loops can be efficiently parallelized with OpenMP