A Parallel Framework for the SuperB Super Flavor Factory

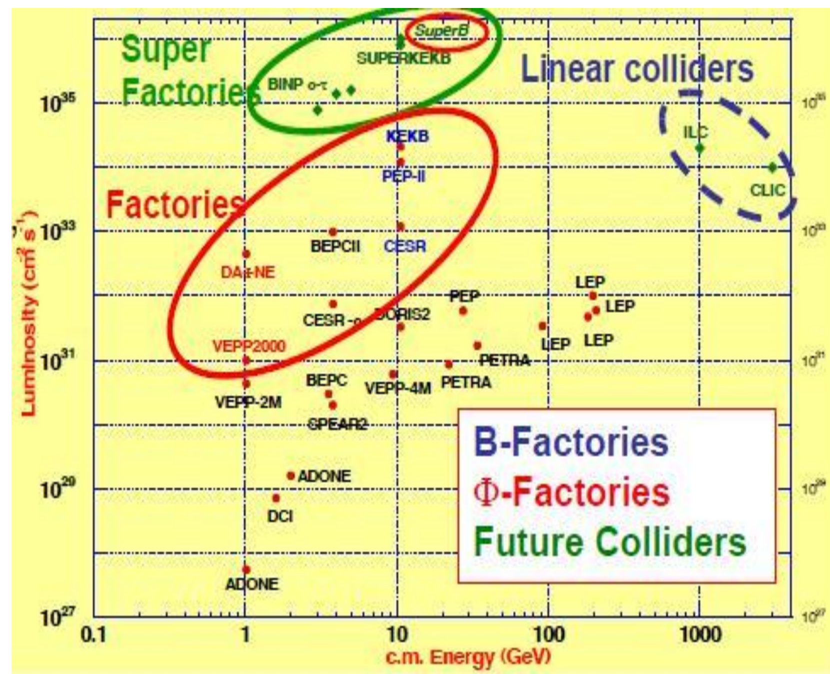
Stefano Longo

on behalf of the SuperB Computing Group

Annual Concurrency Forum Meeting – 04 Feb 2013

SuperB – Main facts

- SuperB is a next-generation high-luminosity $e^+ e^-$ collider facility designed to operate primarily at the $\Upsilon(4S)$
- SuperB carries on the science work of BaBar
- Goal: evidence of physics beyond SM (precision studies/ rare decays)
- Location: Cabibbo Laboratory, Tor Vergata, Rome (IT)
- Design luminosity : $10^{36} cm^{-2} s^{-1}$ (15 ab^{-1} per year)
- Integrated luminosity: 75 ab^{-1} (5 years of science run)
- 4.18 GeV (e^-) x 6.7 GeV (e^+)
- Use crab waist technique



SuperB – Computing

- SuperB is expected to produce as much data as the LHC experiments
  - O(600PB) during its lifetime
- It is clear that the computing challenge is strategic
  - And can benefit from experience gained by LHC experiments

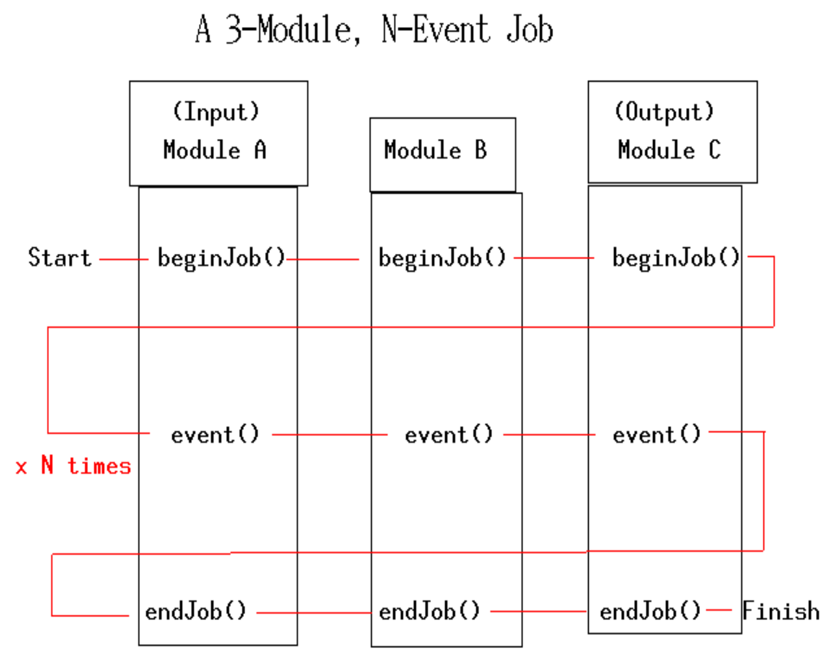
CPU (kHEPSpec)	2016	2017	2018	2019	2020	2021	2022
Physics analysis of Data	54	205	421	638	854	1.070	1.286
Physics analysis of MC	59	222	457	691	925	1.159	1.393
Beam data reconstruction	66	186	265	265	265	265	265
Montecarlo generation and processing	210	588	840	840	840	840	840
Skimming of data	31	86	122	122	122	122	122
Skimming of MC	30	84	120	120	120	120	120
Reprocessing of beam data (previous years)	0	66	252	517	782	1.048	1.313
Regeneration of MC (Previous years)	0	210	798	1.638	2.478	3.318	4.158
Reskimming of reprocessed data	0	46	174	358	542	725	909
Reskimming of reprocessed MC	0	45	171	351	531	711	891
CPU Total	**449**	**1.738**	**3.621**	**5.540**	**7.459**	**9.378**	**11.297**

Need for a framework able to exploit efficiently the computing power of modern many-core systems!

Simulation Application

SuperB detector simulation (FastSim) was used as a testbed to produce a «proof of principles» application, using the BaBar Framework (1995)

- Modular application with hundreds of analysis modules available to the user
- Dynamic simulation setup done via configuration files (Tcl)
- The configuration sets the modules execution sequence and parameters
- Event structure employed to encapsulate every information regarding a simulated event



Analysis

From the analysis point of view we had:

- Studied the dependencies – based on a producer/consumer schema – of each module.
- Designed an algorithm that schedules module execution based on module dependencies.
- Developed a simulator to study speedup and CPU usage efficiency of our solution.

Using module-level parallelism, we have determined that the execution speed-up gained is just **1.43x**

Code Analysis

Analyzing Fastsim code we have found that:

- CPU consumption is really unbalanced between modules
- There is a huge usage of Fortran code, mainly during event generation/simulation (EvtGen, pythia, photos, etc.)

Name	CPU Usage
PmcReconstruct	61.6%
PmcSimulate	20.2%
BtaLoadMcCandidate	4.1%
PacTrkClusterMatch	3.5%
GfiEvtGen	1%

- A single container (Event) is employed to carry all the information inside the analysis pipeline
- Event container used in a non proper way (e.g. for communication between objects, even if no event exists)
- Diffuse usage of static methods employed both to communicate among objects and as a form of «optimization»

Parallel environment

Several parallel/thread libraries were investigated to search for the best match with our model (OpenMP, Cilk+, etc.)

We have decided to employ Intel Threading Building Blocks (TBB), for its feature. In particular:

- Flow graph: allows to use 3 levels of parallelism (between events, inside event and inside algorithms, at the same time)
- Concurrent containers: provides several thread safe containers to repleace stdlib ones
- Concurrent memory allocation: support concurrent heap allocators, to be used instead of standard new/malloc/etc.
- Task synchronization: provides several signaling mechanism between tasks (both wrapping O.S. calls or TBB specific)

Scheduling Model [1/2]

Legacy code was modified in such a way that each module

- Declares what data (products) have to be present inside the Event to start the execution
- Declares what products it adds to the Event
- Has a lock to prevent concurrent execution

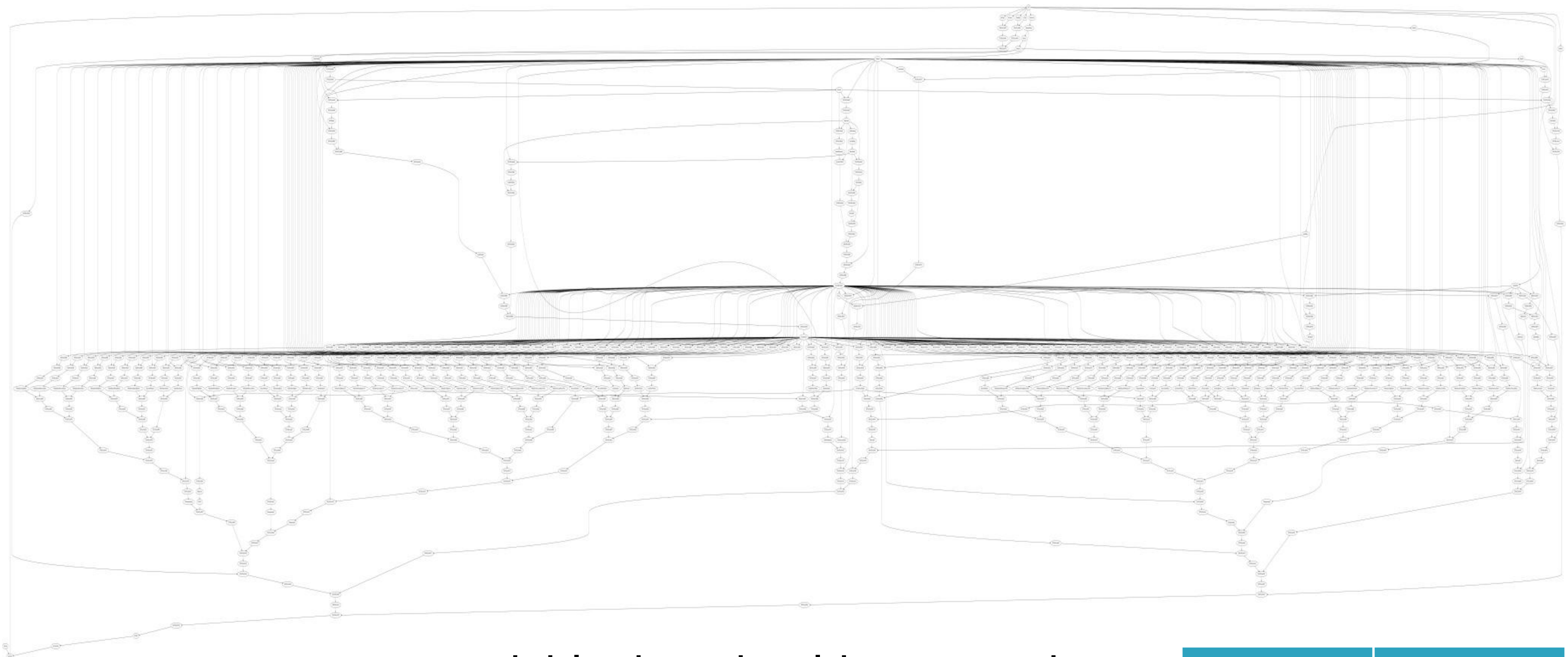
From those information we can produce a dependencies graph, a tree where each node represent an analysis module and each arc a product.

A path from the root to a node is the list of products needed to start the execution of that node.

This schema allows scheduling based on data dependencies

Scheduling Model [2/2]

This is an example of FastSim dependencies graph



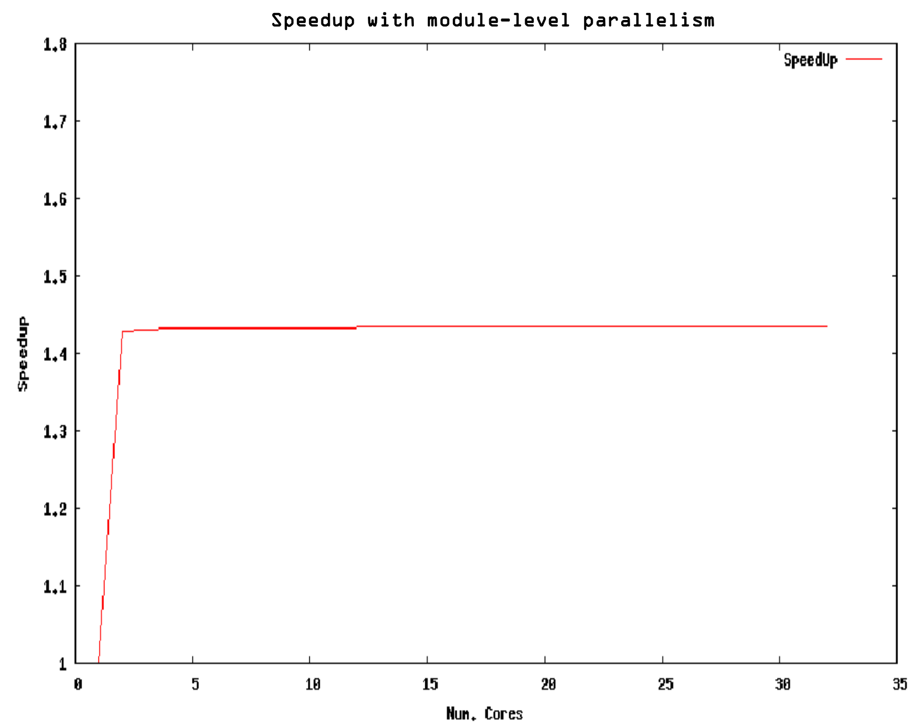
Unreadable, but should give an idea of the problem complexity

Nodes	743
Arcs	1048

Prototype Measures [1/3]

First set of measurements were carried out using module level parallelism only (same setup as legacy code analysis)

- Events are processed one at a time
- Different modules can be executed concurrenlty on the same event (pipeline-like)



This configuration had confirmed the analysis results

- Speedup upper limit **~1.4x**

Prototype Measures [2/3]

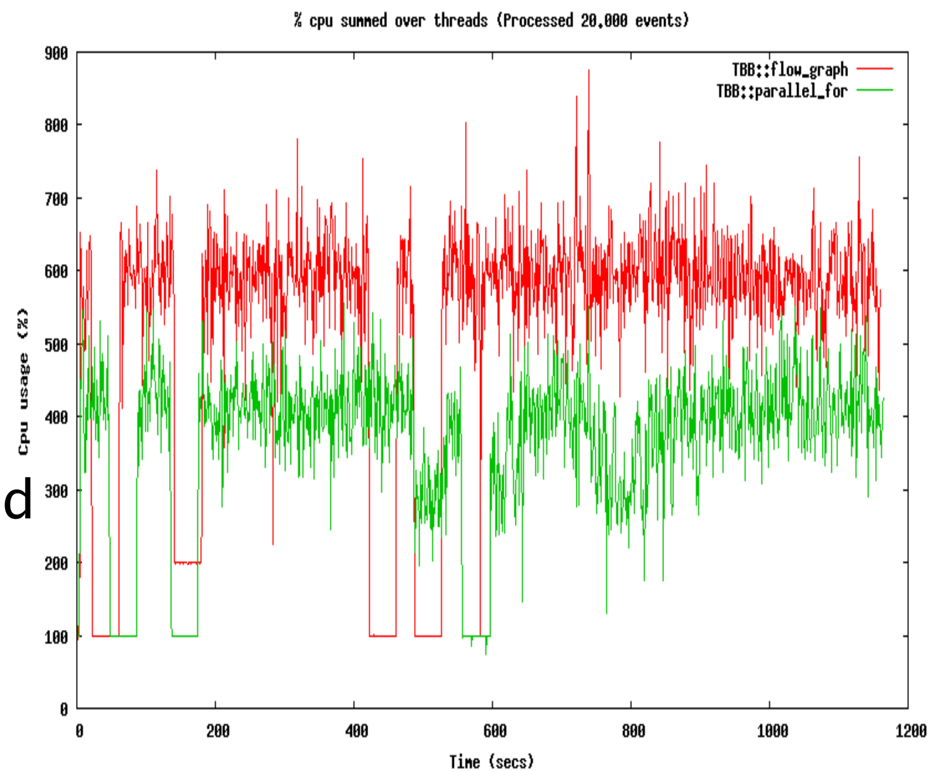
Second set of measurements were performed introducing also parallel event processing (more events processed concurrently)

Benchmark setup

- System: 2 way, 24 cores
- CPU: AMD Opteron 6238
- RAM: 3 GB per core

Parallelization schema:

- *parallel_for* : several analysis sequences executed concurrently, modules executed serially inside sequences
- *flow_graph* : dependencies graph implementation

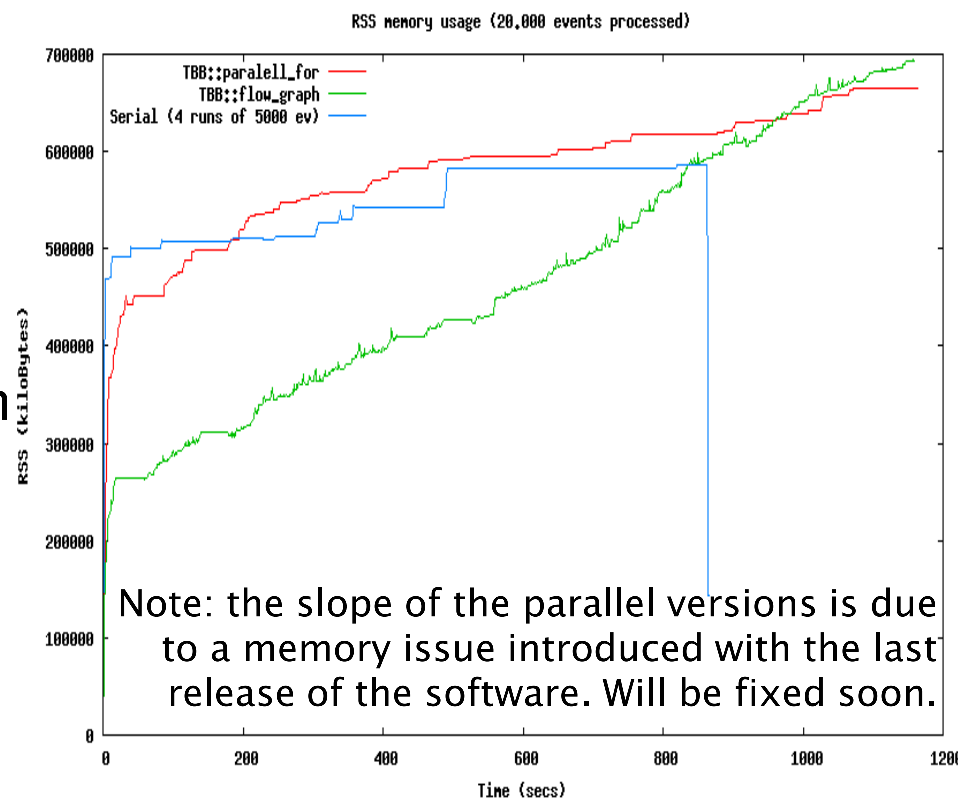


Prototype Measures [3/3]

A last set of measurements were devoted to the application memory footprint

Benchmark setup
- System: 1 way, 4 cores (HT)
- CPU: Intel Xeon E5630
- RAM: 3 GB per core

Comparison
- 4 concurrent *serial* execution Fastsim (5000 events each)
- 1 *parallel_for* Fastsim processing 20000 events
- 1 *flow_graph* Fastsim processing 20000 events



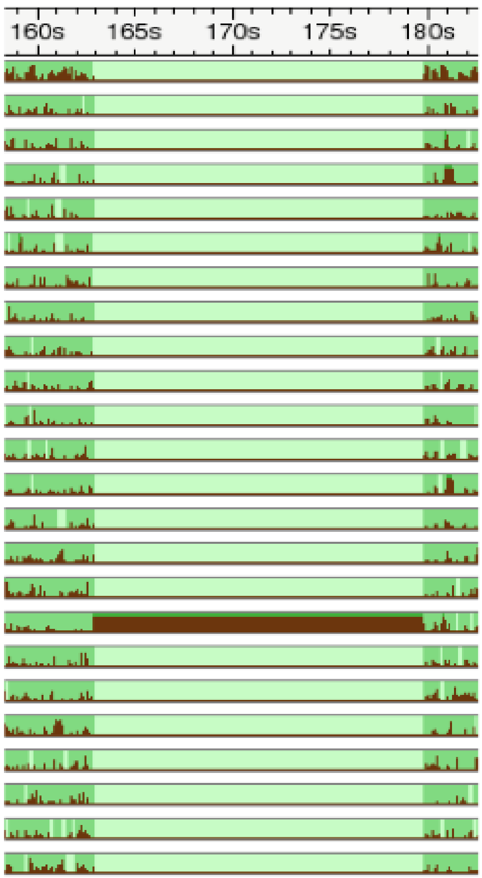
Note: the slope of the parallel versions is due to a memory issue introduced with the last release of the software. Will be fixed soon.

Algorithm Parallelism [1/4]

- A further step was the introduction of the parallelism at algorithm level
- We choose one event generator module – EvtGen – as the test case
- Inside EvtGen, the target algorithm choosen was the computation of the hadronic mass spectra
- Module parallelization was done using a “parallel for” paradigm
- Main goal was to check the usabilty of all the three parallelism levels at the same time

Algorithm Parallelism [2/4]

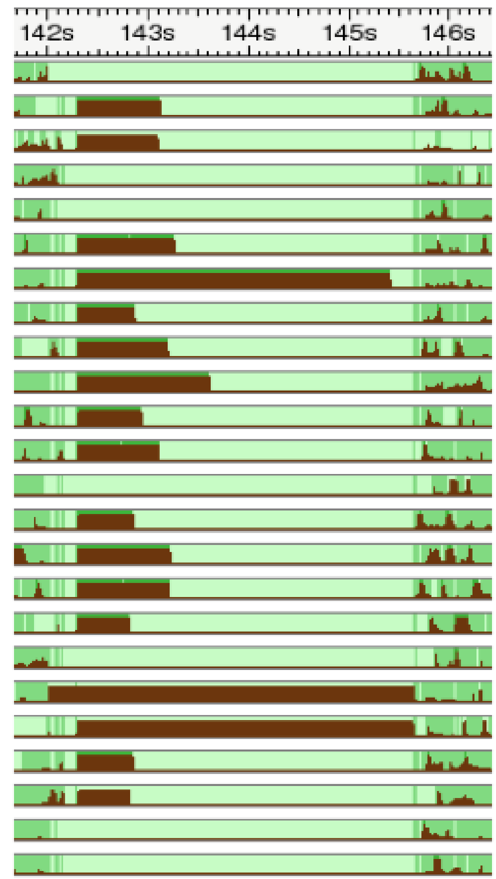
Threads usage: comparison of serial VS parallel execution



Benchmark setup
System: 2 way, 24 cores
CPU: AMD Opteron6238
RAM: 3GB per core

Serial version
≈ 16.5 sec

Parallel version
≈ 3.5 sec



Algorithm Parallelism [3/4]

Putting all together: is FastSim using all the available parallelism levels at the same time?

```
(gdb) thread apply all where full
[. . .]
Thread 14 (Thread 0x7fffe0a49700 (LWP 5326)):
   #7  0x0000000000ec8e2b in ModuleNode::operator()
       eventID = 1132
[. . .]
Thread 7 (Thread 0x7fffe224f700 (LWP 5320)):
   #8  0x0000000000ec8e2b in ModuleNode::operator()
       eventID = 1126
[. . .]
Thread 4 (Thread 0x7fffdbfff700 (LWP 5316)):
   #23 0x0000000000ec8e2b in ModuleNode::operator()
       eventID = 1130
[. . .]
Thread 1 (Thread 0x7ffff7e53720 (LWP 5312)):
   #7  0x0000000000ec8e2b in ModuleNode::operator()
       eventID = 1133
[. . .]
```

Algorithm Parallelism [4/4]

```
(gdb) thread apply all backtrace
[. . .]
Thread 23 (Thread 0x7fffd97f5700 (LWP 5336)):
   #5  0x0000000000d52ddb in PmcSimulate::event
[. . .]
Thread 16 (Thread 0x7fffdb3fc700 (LWP 5329)):
   #5  0x000000000232bb87 in RacRandomControl::event
[. . .]
Thread 4 (Thread 0x7fffdbfff700 (LWP 5316)):
   #21 0x00000000002311a48 in GfiGenerator::event
[. . .]
```

```
(gdb) info threads
[. . .]
* 6 Thread 0x7fffe1a4d700 (LWP 5321)  LoopClass::operator()
  5 Thread 0x7fffe2650700 (LWP 5318)  LoopClass::operator()
  4 Thread 0x7fffdbfff700 (LWP 5316)  LoopClass::operator()
  3 Thread 0x7fffe2a51700 (LWP 5317)  LoopClass::operator()
  2 Thread 0x7fffe2e52700 (LWP 5315)  LoopClass::operator()
[. . .]
```

Conclusions [1/2]

From the prototype we have defined a computing model where:

- An analysis is defined as a set of modules
- Each module has to be independent from others
- A module must define the products it needs to run
- A module must define what it produces during its execution

Measurements done on the prototype demostrates that

- The model can be used to reduce the memory footprint (as an alternative to run N separate analysis, with N=number of cores)
- The scheduling schema may be employed to efficiently use systems with large number of cores
- Event, module and algorithm parallelisms can be emploied simultaneously

Last but not least, measurements on the prototype were taken using a production setup -> The prototype works!

Conclusions [2/2]

Some general software development guidelines were defined based on the framework analysis and prototype:

- Fortran code has to be removed
- Widespread usage of static objects has to be avoided
- Each module has to be more OOP-compliant, in particular for what concern incapsulation
- Auxiliary data structures (Event container, etc.) have to be developed to allow concurrent access to data
- For some analisis algorithms a code rewriting can provide a massive parallelism level

[Old] Future plan:

- Ready to formalize specifications for analysis modules.
- Ready to start the development of a production framework

Thanks
For your attention!