Exploiting new CPU architectures for the SuperB analysis framework

CHEP12 Event Processing session

Marco Corvo
on behalf of the SuperB Computing Group

CNRS and INFN

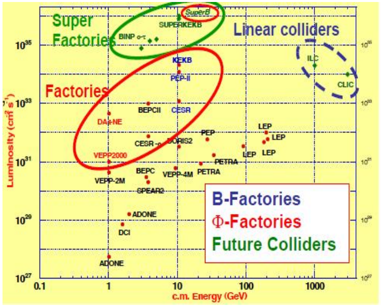
May 22 2012

Outline

The experiment

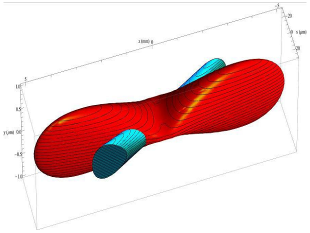
- SuperB is a next-generation high-luminosity e^+e^- collider facility designed to operate primarily at the $\Upsilon(4S)$

- Its physics goal is to search for evidence of physics beyond SM in precision studies of rare and forbidden decays
- Will be built at Cabibbo Laboratory, Tor Vergata, Italy (near Frascati INFN lab)

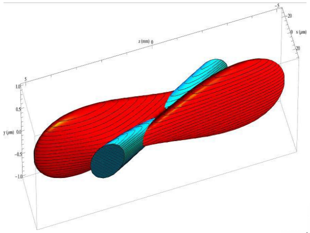


Collider parameters

- Design luminosity is $10^{36} cm^{-2} s^{-1}$
  - Integrated $15ab^{-1}$ per year at design luminosity
- Target overall integrated luminosity is $75ab^{-1}$ in 5 years operations
- $4.18\ GeV\ (e^{-})\ x\ 6.7\ GeV\ (e^{+})$
- Benefit from a new design of the beam crossing area, the so called **crab waist**



Without Crab Waist



With Crab Waist

The computing challenge

- 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**

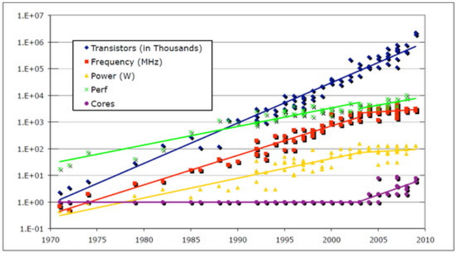
The computing challenge II

The computational power we need has two major issues:

- It has a cost in terms of electric power supply
- CPU clock frequencies cannot rise forever, thus limiting the peek of processing power

The technology trend in CPU architectures provides the solution

- In the last years we've seen that the frequency trend just flattened
- But number of cores integrated on single or multiple dies increased



Trends in CPU architecture

Match HEP requirements

Commercial hardware turned to parallelism: smaller computational unit working altogether and sharing resources

- HEP must adapt to this trend exploiting:
  - Multi and many cores CPU
  - (GP)GPU
  - Hybrid accelerators (Intel® MIC)
- Every machinery has its own peculiarity regarding the development tools and the specific computational target
  - GPU computing has more applications in algorithm parallelism
  - Multi and many core computing is more suitable for intra algorithms and event parallelism

Parallelism

Parallelism is the possibility to exploit every processing unit for a specific calculation, thus distributing the computational payload while sharing some resources (memory, system buses, IO . . .)

Physics problems in HEP are a natural source of parallelism

- Analysis algorithms work on millions of events $\rightarrow$ execute the same algorithm on many events concurrently
  - Memory bottleneck
- Some algorithms can be split up $\rightarrow$ execute chunks of computation on the same event concurrently
- Dataflow can be factorized to run different processing steps concurrently $\rightarrow$ this is our primary goal with the SuperB framework

Current scenario

The current analysis frameworks share some features which prevent parallelism

- Written long time ago thus suffering from severe lack of modern programming paradigms
- Worst of all **intrinsically serial**

First step toward the parallelization of SuperB Framework is the analysis of current code, mostly based on BaBar legacy code, specifically one of the executables of Fast Simulation.
The analysis of a particular dataflow has a main goal:

- Factorization of the workflow

Measuring hidden parallelism

The starting point is a specific Fast Simulation executable whose data flow includes 127 modules

- The analysis of the workflow has been performed
- For each module the analysis extracts:
  - The list of **required input** or **data products** needed by the module to run
  - The list of **provided output** generated by the module
  - The event processing time
- Basically the trick is to look inside the *Event* and dive into physics data products to understand who provides or requires what
- These lists are used to build an adjacency matrix, a means to represent the connections among vertices of a graph

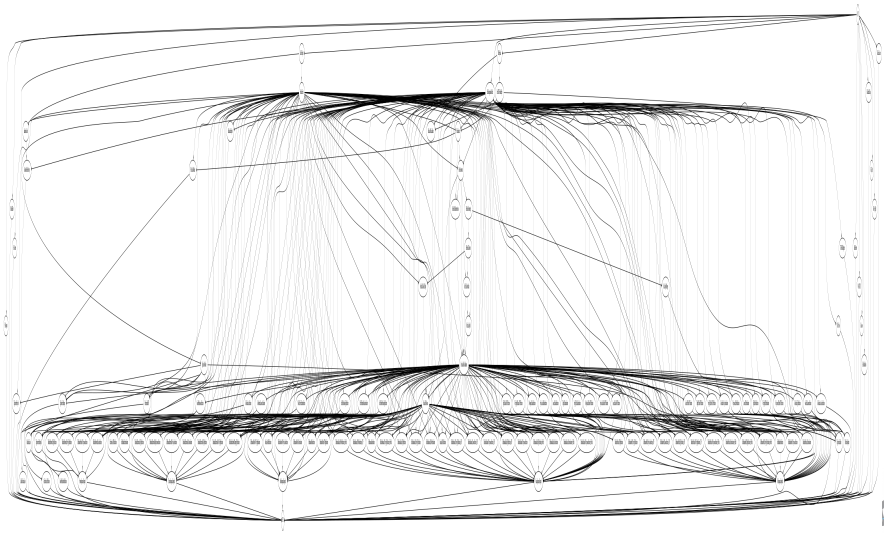
Results

This analysis shows that the current code of Fast Simulation could benefit from modules parallelization.

- In particular, for this specific case, we found that the **tree** has ten levels, each with twelve nodes on average
- This clearly suggests that, on average, twelve modules could be scheduled in parallel

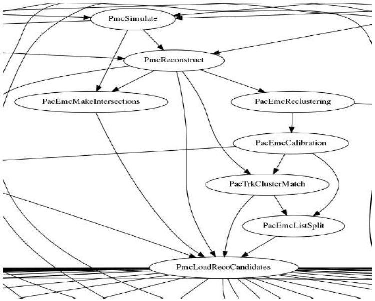
Num of modules	127
Graph Depth	10
Min Rank	1
Max Rank	54
Avg Rank	12

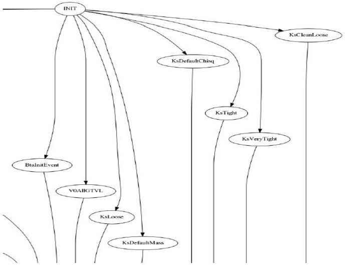
Complete graph of dependencies



Zoom in

These are snapshots of the complexity we have to deal with. A big effort has been done to extract the dependencies of the modules, as the only source of information are the data products that modules write into the event, the data structure where physics results are stored





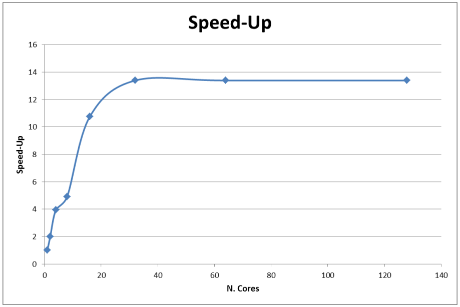
Scheduling simulation

To show the potential speedup, a parallel scheduling algorithm has been developed

- It's based on the same idea of an operating system scheduler (readiness of tasks)
- The algorithm loops over the list of modules and selects for execution those in a *ready* state
  - The condition is satisfied when all the **required inputs** are available to a particular module
- We measured the global time needed for the whole chain to complete, assuming for every module an average duration measured from real use cases.

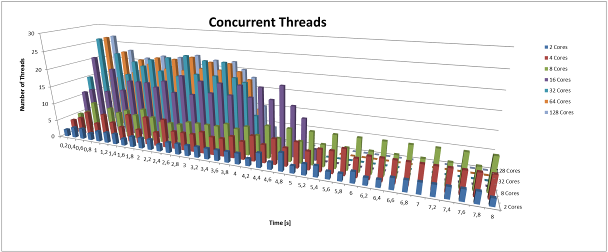
Results

- The simulation has been done limiting the maximum number of threads which can run concurrently
- For each simulation we practically limited the number of modules in a given queue to be 2^n
- The aim is to show that the speedup increases linearly at least up to the average number of modules which are able to run concurrently (12 in our previous measurement)
- The figure shows that the speedup in fact is scaling almost linearly with the number of threads



Speedup

Results II



Time distribution of threads

- Time distribution of threads shows that, with our assumptions, their number remains almost constant during job execution
- This means that we should be able to optimize the usage of resources, keeping loaded all available cores

Available tools

A number of tools to express parallelism is available (list not exhaustive)
Multi/many cores CPU:

- Grand Central Dispatch/libdispatch
- OpenMP tasks
- Intel® Threading Building Block

GPU:

- OpenCL
- CUDA

Introducing parallelism

We demonstrated that SuperB code can be run in a parallel fashion, that is scheduling independent modules concurrently.
To implement our prototype we choose to use:

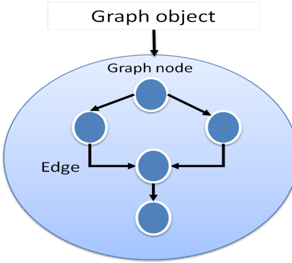
1. The SuperB framework
2. Intel® Threading Building Blocks (Tbb from now on)

- Tbb seems promising to implement easily the **require-provide** mechanism
- SuperB framework should be the easiest place where to try implementing parallelism
  - Modules are written for this framework, so effort is required only to introduce our **require-provide** mechanism
  - Major effort required to modify modules according to Tbb schema
    - Modules must be implemented as *functors*

Intel® Tbb

- Tbb is a library offering a rich approach to express parallelism in a C++ program
- It represents a high-level, task-based parallelism that abstracts platform details and threading mechanisms

- For our purposes we are investigating a particular object of Tbb, that is `flow::graph`
- There are three components interacting in this environment
  1. The `graph` object
  2. The *nodes*
  3. The *edges*
- Nodes invoke user function objects and manage messages passing to/from other nodes



Graph example

- Edges represents the connection, or arch, between two nodes

Putting all together

- Every module which is managed by SuperB framework declares two lists
  - A **require** list with the names of products needed by the module to run
  - A **provide** list with the names of products generated by the module
- These lists are then mapped onto module names which are represented by Tbb nodes
- The framework builds consequently the graph object of dependencies
- Tbb scheduler takes the responsibility to run the modules according to the graph of dependencies

Short and mid term plans

- Extract the complete dependency graph for all SuperB FastSim modules
- Modify modules accordingly to express them as Tbb nodes
- Measure performances
  - A side effect would be also the improvement and redesign of current code to optimize it
- Develop the prototype of framework designed with the **require-provide** schema, based on our experience with the SuperB one and other existing frameworks (Art is currently under evaluation)

Conclusions

- Reveal parallel potential inside existing code is strategic
  - It optimizes (hopefully) resources usage
  - Helps to better understand algorithms for future development
- Current efforts are focused on adapting SuperB framework and modules to Tbb infrastructure
- In the long term we will abandon the current SuperB framework for a new one which is natively parallel and whose prototype will be designed based on our experiences