Use of Threading Building Blocks in a parallel framework prototype for SuperB

Marco Corvo

CNRS and INFN

October 10th 2012

Outline

A proof of concept

1. This activity nust be regarded as a 'proof of concept'.
2. Results are a starting point for the definition of an architecture and a computing model and are not intended to be used in a production system

A concurrent framework

The first step towards the parallelization of SuperB Framework is the analysis of current code, mostly based on BaBar legacy code. In particular we focused on one of the executables of Fast Simulation.
The analysis of a particular dataflow has the main goal of factorizing of the workflow to exploit:

- Parallelism at event level (more events being processed concurrently)
- Parallelism at module level (more modules running concurrently on the same event)

Measuring hidden parallelism

The specific Fast Simulation executable data flow includes 127 modules. 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 processing time

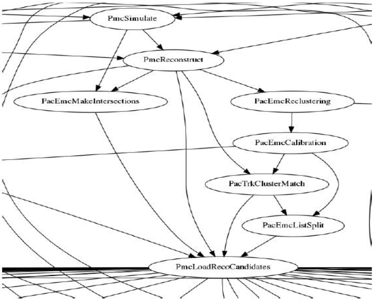
Basically the trick is to look inside the *Event* and dive into physics data products to understand who provides or requires what.
These lists are used to build a graph of dependencies

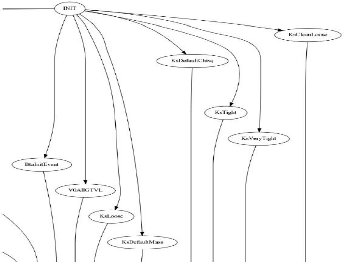
Results

- This analysis showed that the current code of Fast Simulation could benefit from modules parallelization, that is there are modules which can run concurrently
- It turned out that modules which can run in parallel take only a few percent of the overall processing time ($\approx 10\%$)
- On the other hand most of the processing time is spent inside modules which cannot run in parallel

Zoom in

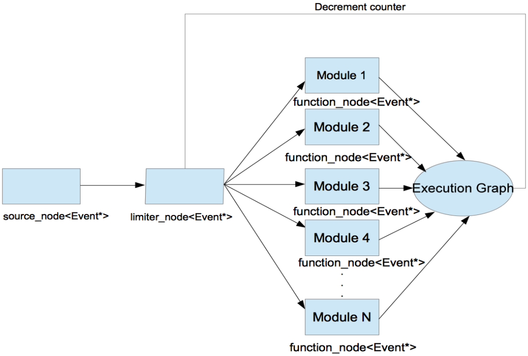
These are snapshots of the complexity we have to deal with. A big effort has been done to extract the dependencies of the modules, as the only source of information are the data products that modules write into the event, the data structure where physics results are stored





Implementing module level parallelism

Module level parallelism is implemented using Tbb `flow::graph` objects where every module in the analysis chain is mapped onto a `flow::graph` node



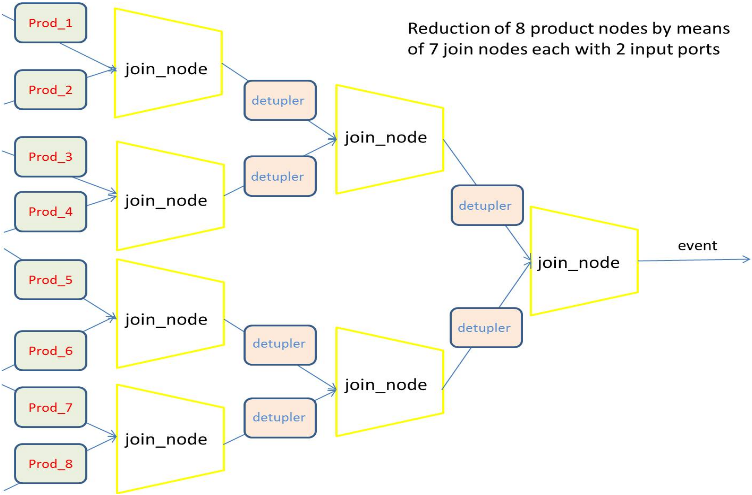
- The message passed among modules is a pointer to an *event*
- Each module is a `function_node` whose `operator()` executes the module algorithms and returns the same pointer to *event*
- The event has been made "thread safe" substituting the list of products with a `concurrent_hash_map`

Issues

Currently there's a limit in Tbb `flow::graph` as regards joining nodes

- The schema works so that a given module runs when all its "required" products are available
- If module A needs N products to run, we need to notify A when they are all available
- This is possible using a particular `flow::graph` node called `join_node`
  - This node has a mechanism which forwards a message to its successors only when all of its input ports have been filled
  - The issue with this node is that the number of input ports must be declared in advance and not dinamically
- The solution is to recursively reduce the graph combining join nodes in couples

Graph reduction



Reduction of 8 product nodes by means of 7 join nodes each with 2 input ports

Graph reduction

Moving to event parallelism

We know that our current implementation suffers from the usage of locks around modules, to protect against Fortran Common Blocks, from C++ objects used as Common Blocks and static objects.
We also expect that module level parallelism suffers from a limitation due to the bad distribution of the computing time among modules.
For these reasons we explored parallelism also at event level

- The first attempt was to exploit Intel® Tbb `parallel_for`
  - This is the simplest approach as with minor changes in the main loop we are able to inject more than one event into the analysis chain
- The algorithm `parallel_for` applies a function, the physics *module*, in parallel to a range of objects, the *events*

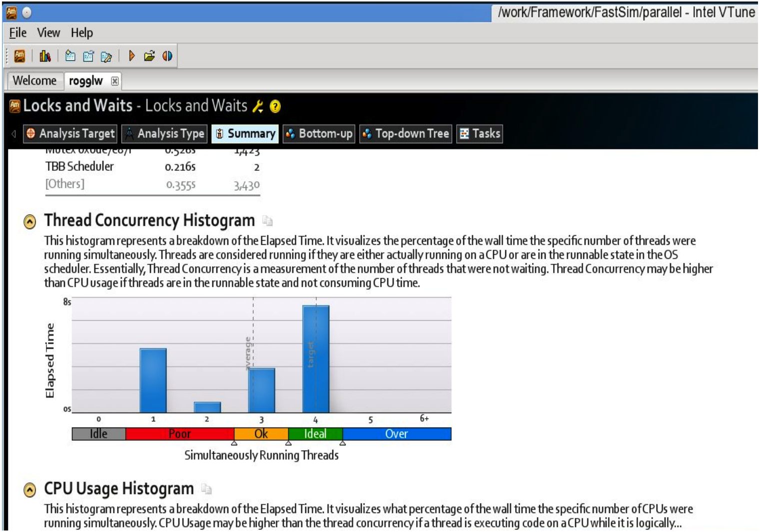
Event level parallelism should be adopted anyway also to allow a unique Tbb instance to manage efficiently all the available resources through its scheduling mechanism

Integration with module parallelism

To integrate module level parallelism we substituted the `parallel_for` function with two `flow::graph` nodes

- Our initial setup was to have a `queue_node` responsible to inject the events into the graph towards a `limiter_node` in order to control the number of events running concurrently
- The `queue_node` creates all the events to be processed by the workflow
    - This causes the initial memory footprint of the executable to be significantly higher than the ordinary 'serial' one ($\approx$ 800 Mb vs $\approx$ 450 Mb with 10k events)
- The solution is to put a `source_node` before the `limiter_node` as the former creates a new event to be injected only when the latter has room for another one

Results I (Single Intel® Xeon E5630 (quad core, 2Gb/core))



Mutex 0x0de7e87f	0.528s	1,423
TBB Scheduler	0.216s	2
[Others]	0.355s	3,430

Thread Concurrency Histogram

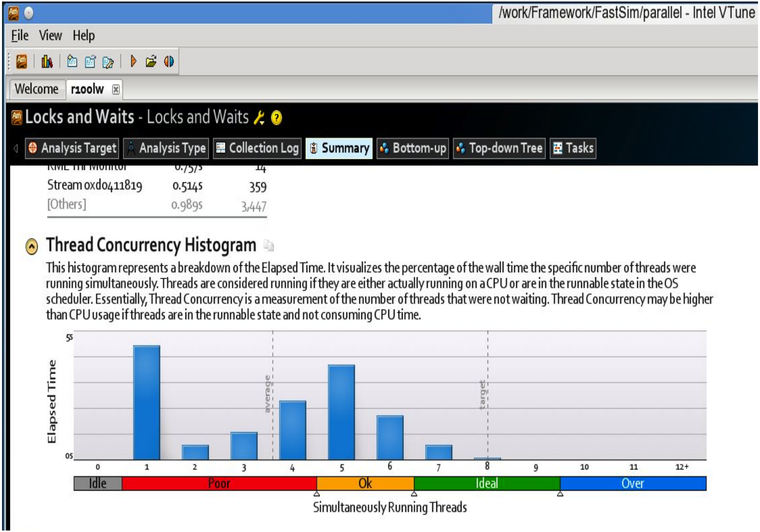
This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically...

Results with parallel_for (HT off)

Results II (Single Intel$^{®}$ Xeon E5630 (quad core, 2Gb/core))



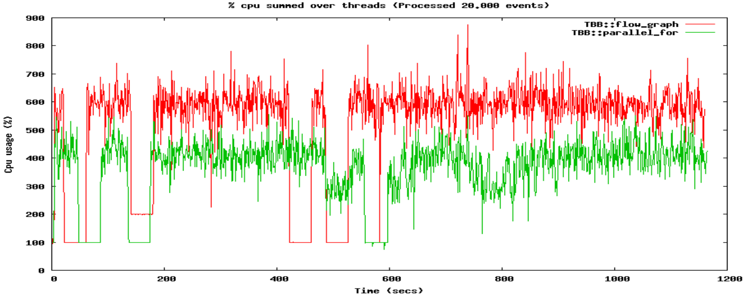
Stream 0xd0411819	0.514s	359
[Others]	0.989s	3,447

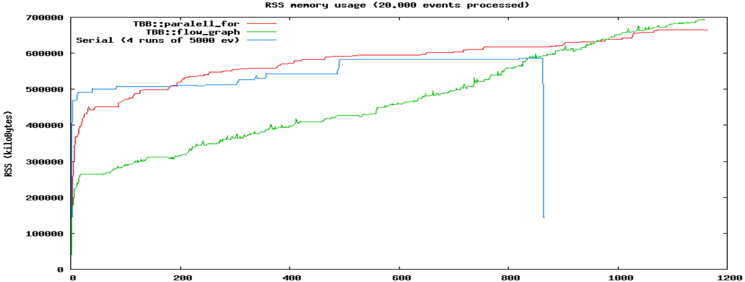
Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

Results with parallel_for (HT on)

Results III (AMD$^{®}$ Opteron 6238 (2x12 cores, 3Gb/core))


% cpu summed over threads (Processed 20.000 events)


RSS memory usage (20.000 events processed)

Conclusions

- Our implementation so far is not able to push up the CPU occupancy due to eccessive locking and serialization points in the code
- Our efforts have been nonetheless rewarded as we have shown the speedup gained by event and module level parallelism
- The parallel potential inside existing code is strategic
    - It optimizes resources usage and increases computing speed
    - Helps to better understand algorithms for future development
- Issues are still under investigation, but we are confident to be able to solve them
- In the long term we will abandon the current SuperB framework for a new one which is natively parallel and whose architecture will be designed based on our experiences