Updates of R&D activities on SuperB analysis framework

Marco Corvo

CNRS and INFN

September 20th 2012

Outline

1. Brief review of past activities
2. A (temporary) change of strategy
3. Where we are now
4. Open issues
5. Conclusions

A concurrent framework

(The first slides are grabbed from our presentation given at NYU for CHEP 12)
The current SuperB 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 was 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 (to be revised)

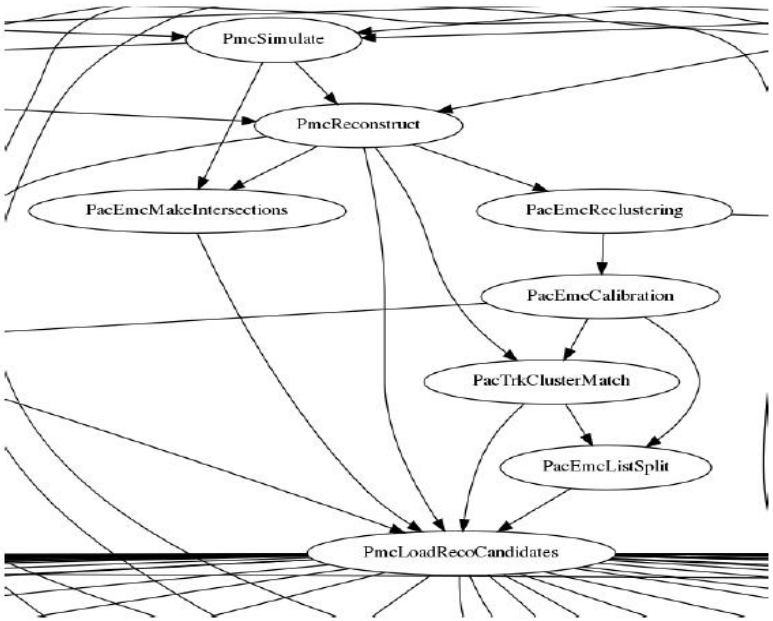
This analysis shows that the current code of Fast Simulation could benefit from modules parallelization.

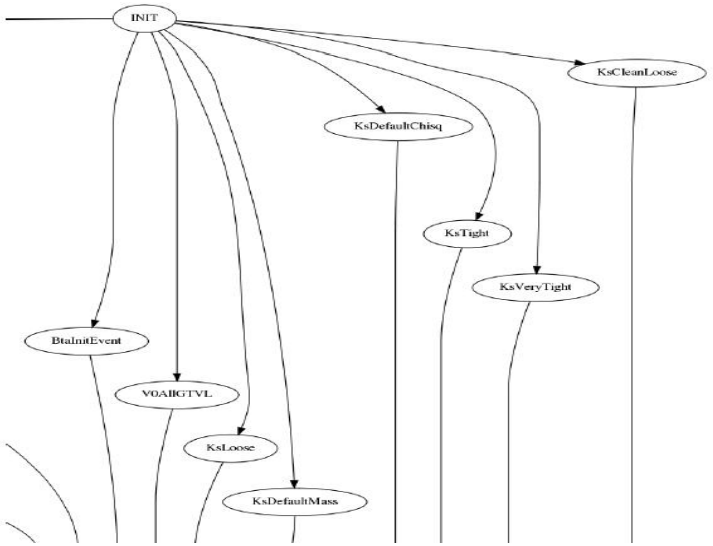
- In particular, for this specific case, we found that the **tree** has ten levels, each with twelve nodes on average
- This clearly suggests that, on average, twelve modules could be scheduled in parallel

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

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





Timing error

One of the results that made us confident to be able to improve dramatically the performances of our framework was the amount of time spent by this particular analysis into the "parallelizable" set of modules

- If the analysis were to spend 90% of the wall time into "parallelizable" modules, then exploiting concurrency would result in gaining a speed-up of (roughly) $S = \frac{T_{wall}}{\bar{t}_{moduleX}}$
- The timer we had (BaBar legacy code) had a bug
- This led us to change our approach to the problem, that is to try to increase the number of events analyzed (move from module level to event level parallelism)

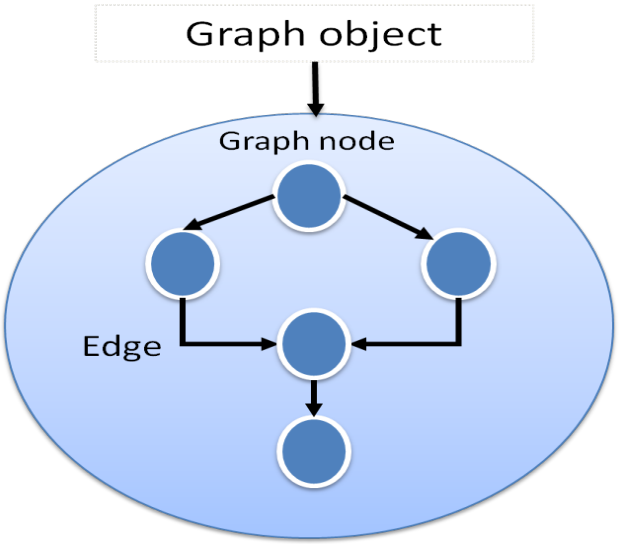
Moving to event parallelism

- 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 class `parallel_for` applies a so called "functor", the physics *module*, to a range of objects, the *events*

Integrate with module parallelism

- The `parallel_for` approch allows only palallelism at the event level (we inject more that one event in the analysis chain)
- Our goal is to exploit also module level parallelism (allow modules to run concurrently on the same event)
- This is the approach that we want to pursue despite the initial bad performances
  - Because it is a valid and general model that can improve even more the performances
- This was done using Tbb `flow::graph` objects where every module in the analysis chain is mapped onto a `graph` node

Module parallelism



Graph example

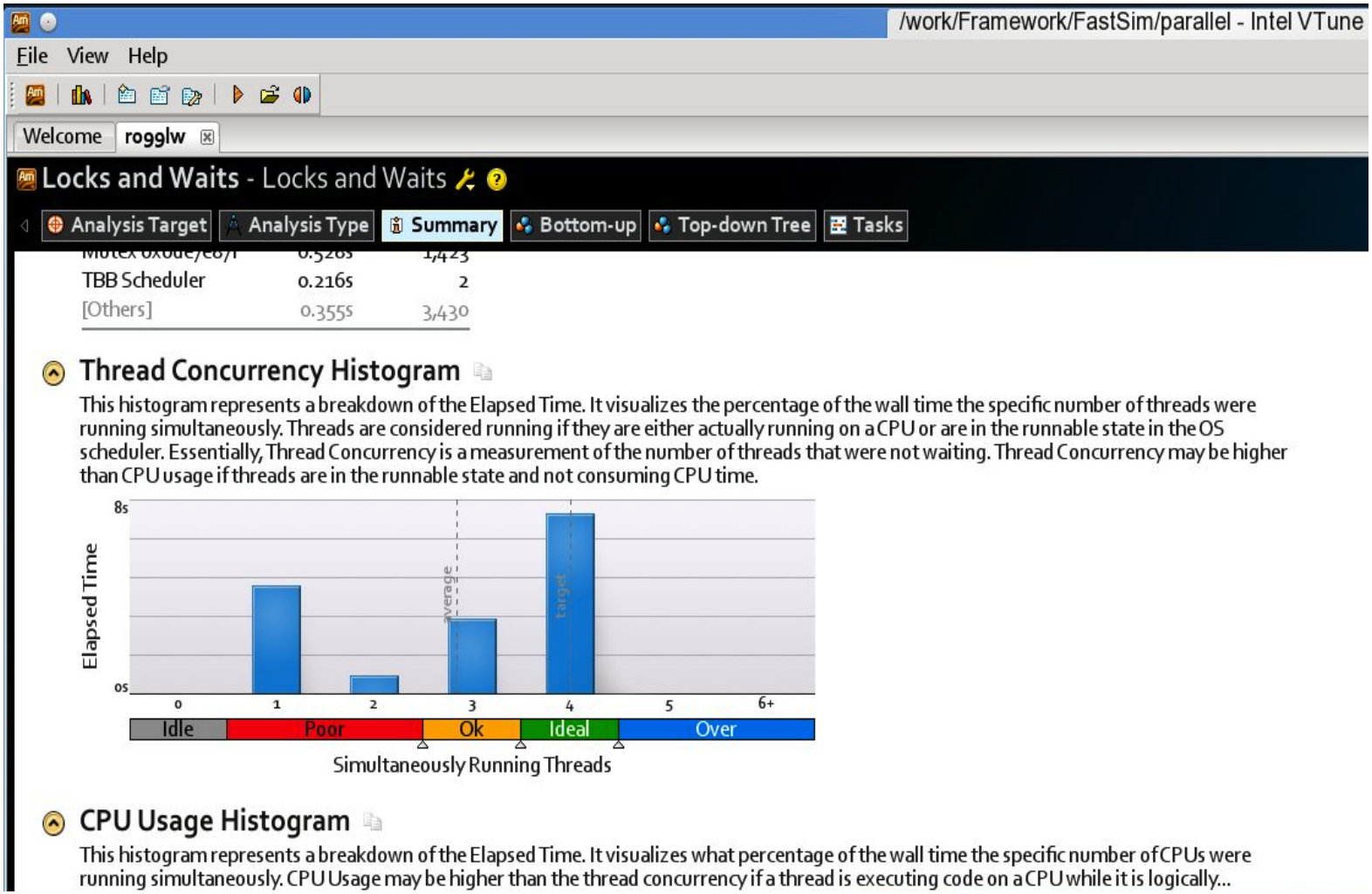
- To map the modules of the analysis chain to `graph` nodes, we analysed the event which is the container where all physics products are read and written
- The result is a list of requirements and products for each module that we use to build a graph of dependencies
- Major effort required to modify modules according to Tbb schema as modules must be implemented as functors

Features of this work

In this work we introduced some features

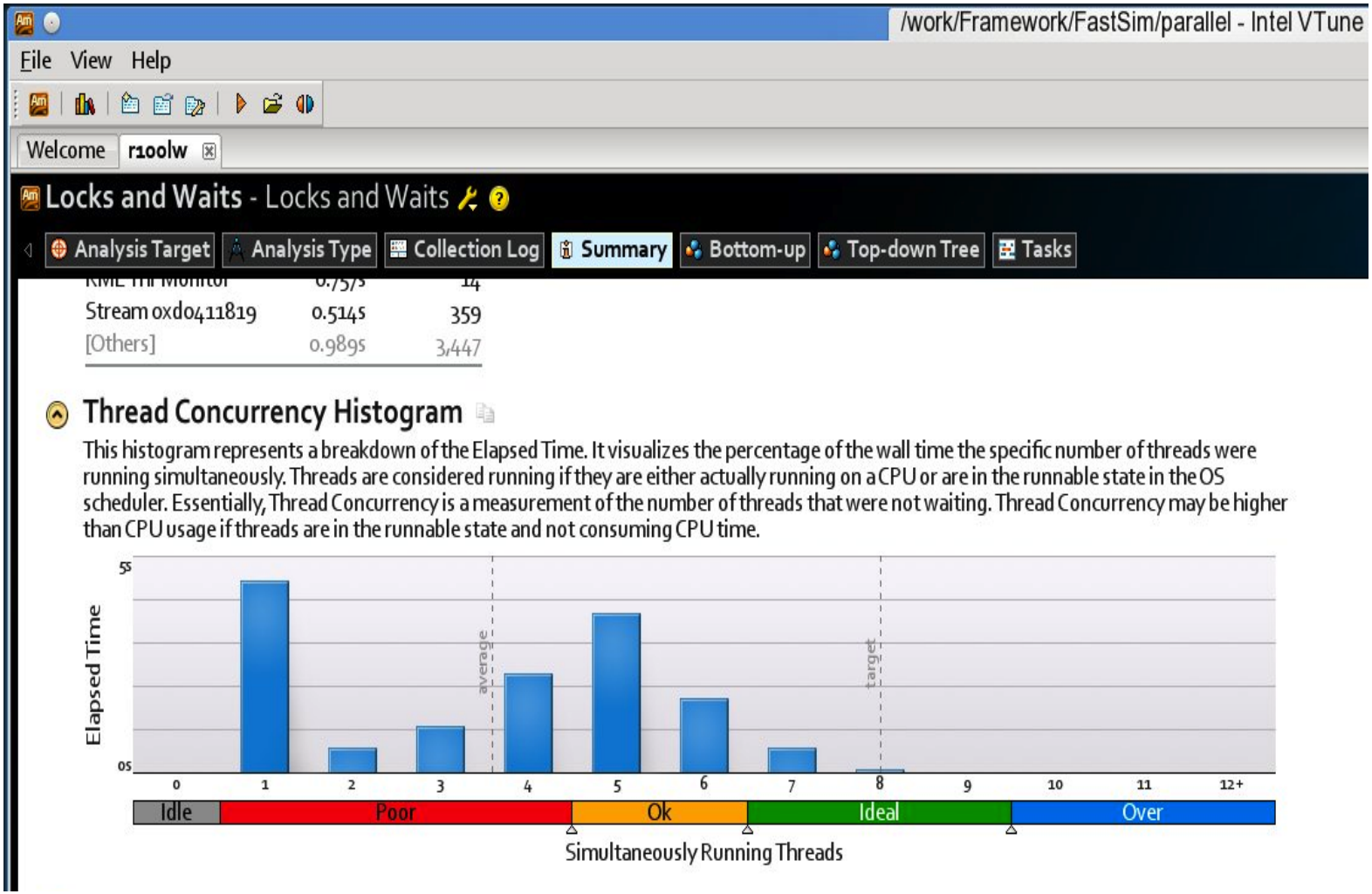
- Modules have been wrapped by a functor which calls the `event` method on the event
- Every module has been instrumented to extract two lists: one contains the requirements for the module to operate, the second contains the products of the module itself
- A method has been added to the class `Framework` to build a Tbb `flow::graph` object
- The event has been made "thread safe" substituting the list of physics products with a Tbb `concurrent_hash_map`
- A mechanism to limit the "injection" of events into the graph has been implemented

Results I



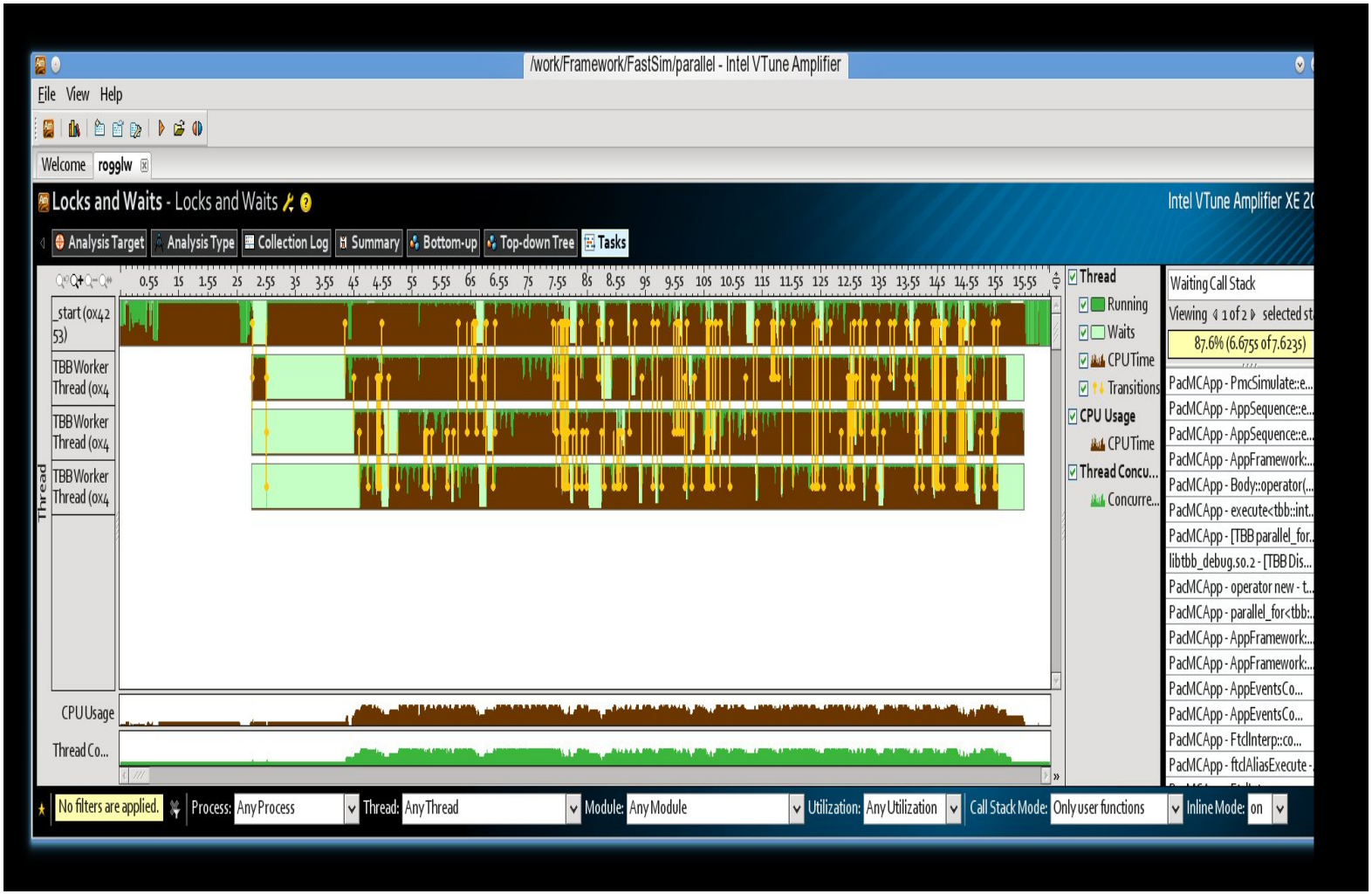
Results with parallel_for (HT off)

Results II

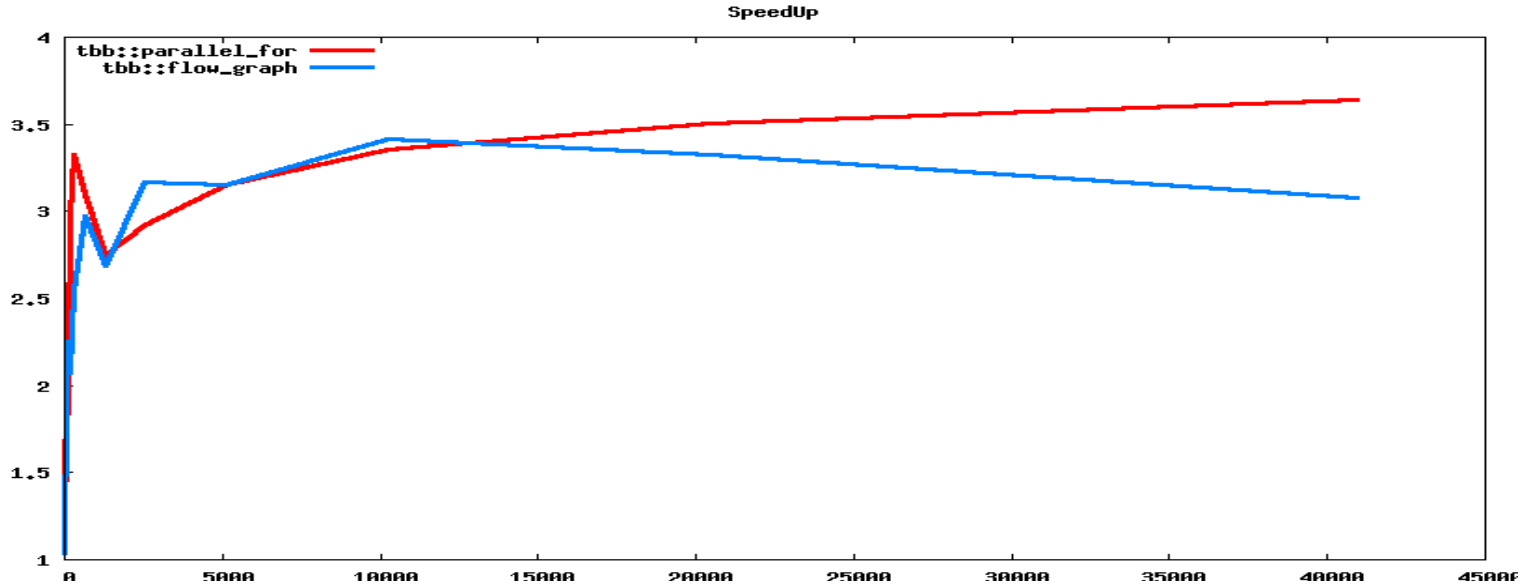


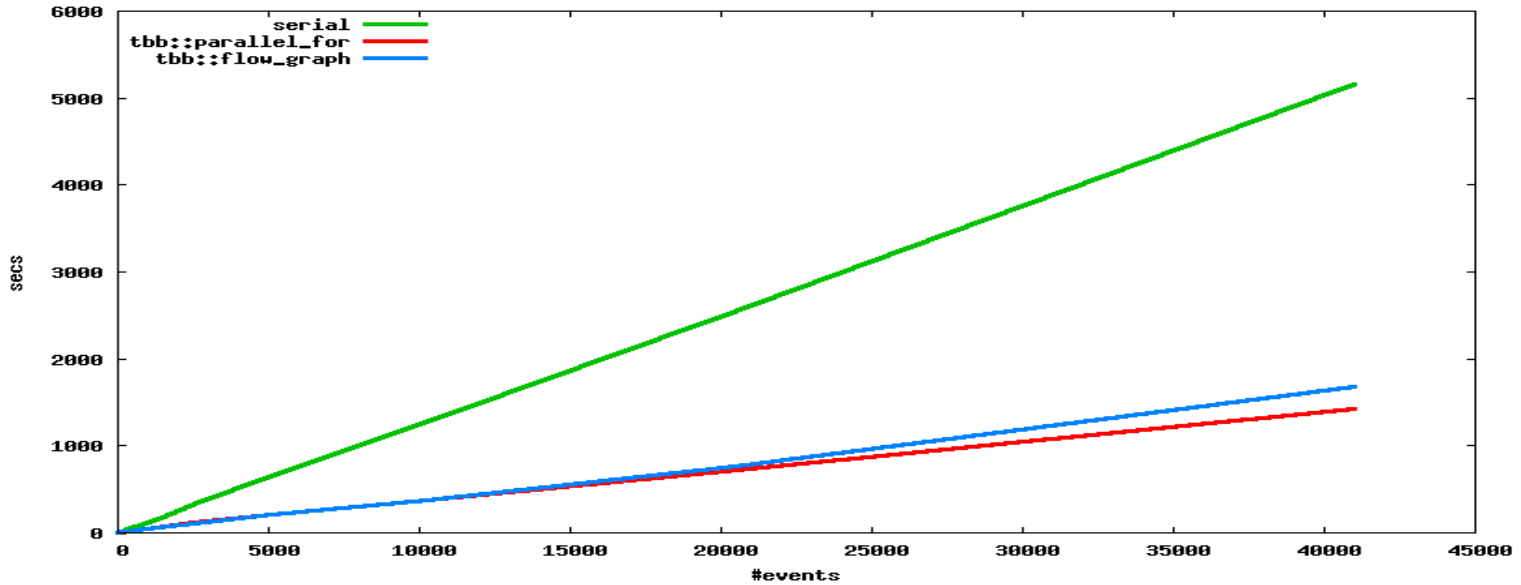
Results with parallel_for (HT on)

Results III



Concurrency breakdown (HT off)


SpeedUp
tbb::parallel_for
tbb::flow_graph
4
3.5
3
2.5
2
1.5
1
0
5000
10000
15000
20000
25000
30000
35000
40000
45000

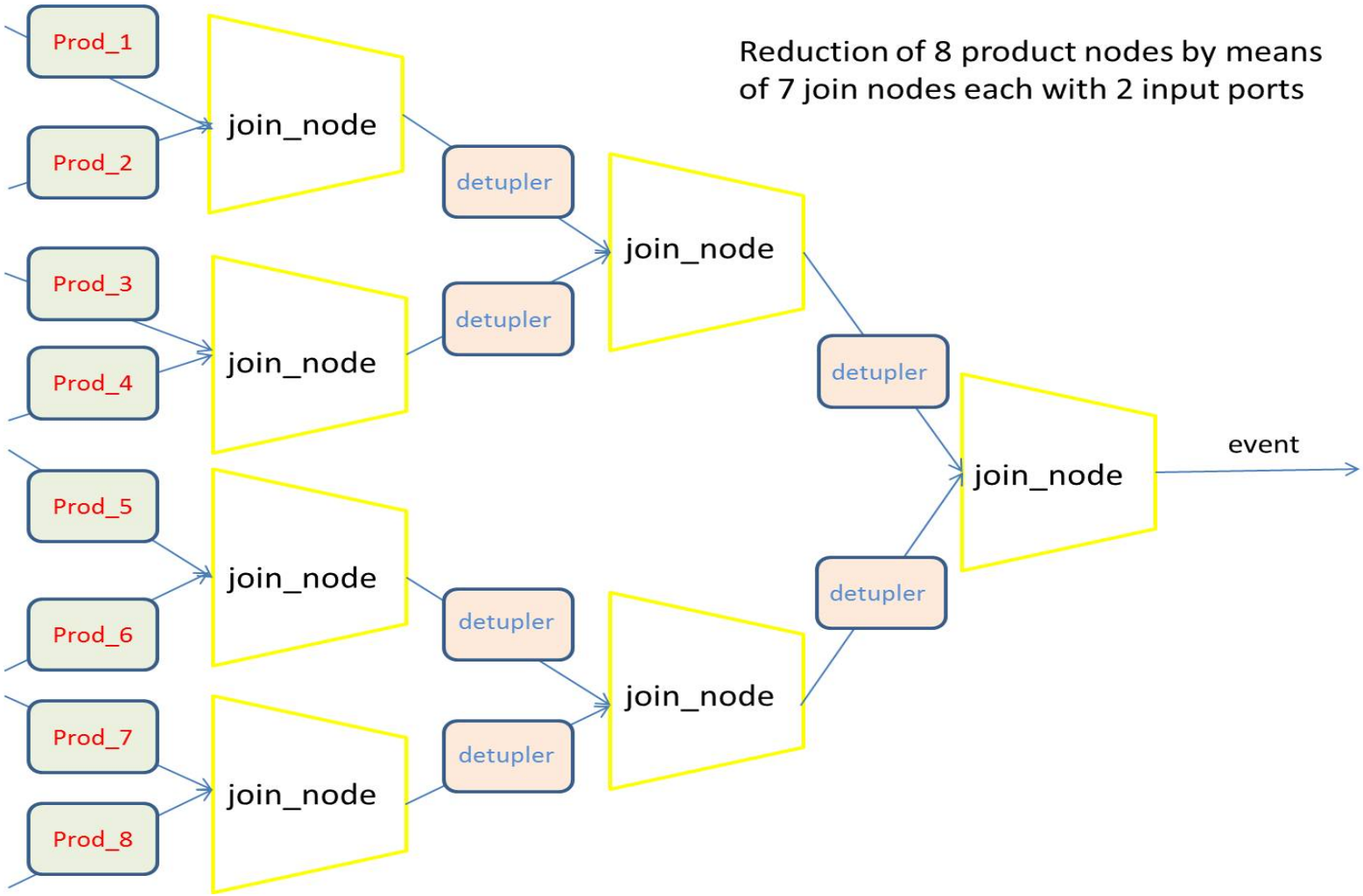

Wall Clock Time
serial
tbb::parallel_for
tbb::flow_graph
secs
6000
5000
4000
3000
2000
1000
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
#events

Issues

There's a intrinsic limit in Tbb `flow::graph` as regards the management of 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



Graph reduction

A Tbb limit?

- We encountered another problem on our way towards parallelism: a deadlock which prevents us from injecting more than 8 events into the graph
- This problem rises as we lock a particular module
- Locks are currently needed on certain modules to guarantee that only a given event can use it
  - This happens because modules are instantiated once at graph creation and we must avoid that two events access the same "common blocks" messing up the variables
  - Luckily this is true only for a limited number of modules, while for others we modified things to have thread safeness (e.g. data locality)
- The limit is equal to the number of threads that Tbb spawns to perform calculations
  - I. e. we knock against this deadlock when we inject 9 events
- Problem under investigation...
- During our test we also saw what is likely to be a memory leak

Short and mid term plans

- We need to come to a full understanding of all dependencies among modules
- This is a good starting point, but there are a lot of analysis patterns to consider, not to mention a full reconstruction framework
- This exercise was helpful to understand also what modules can do and what modules must NOT do to improve optimization
- There's room for improvements also in the generation part, trying to limit (or at least factorize) the use of Common Blocks in Fortran code

Conclusions

- Our efforts so far have been rewarded as we shown the speedup gained by event and (partial) module level parallelism
- The (disclosed) 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