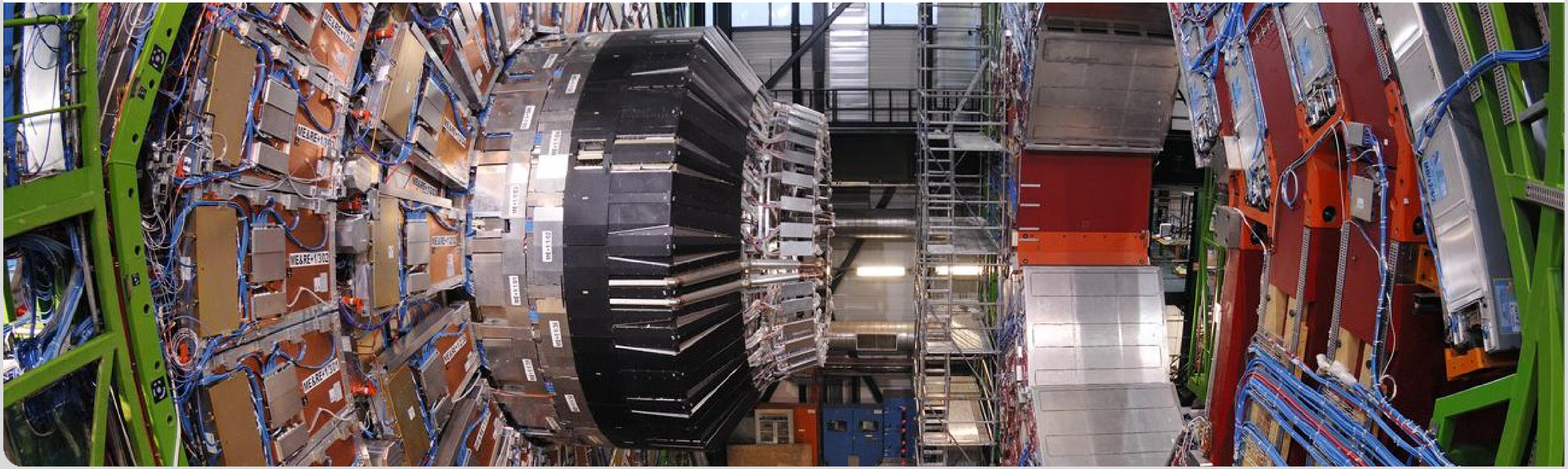
First Algorithm Parallelism in CMS Software Framework (CMSSW): Track Seeding

Thomas Hauth, Danilo Piparo, Vincenzo Innocente



Framework and Algorithm Parallelism

Beyond Event Level Parallelism

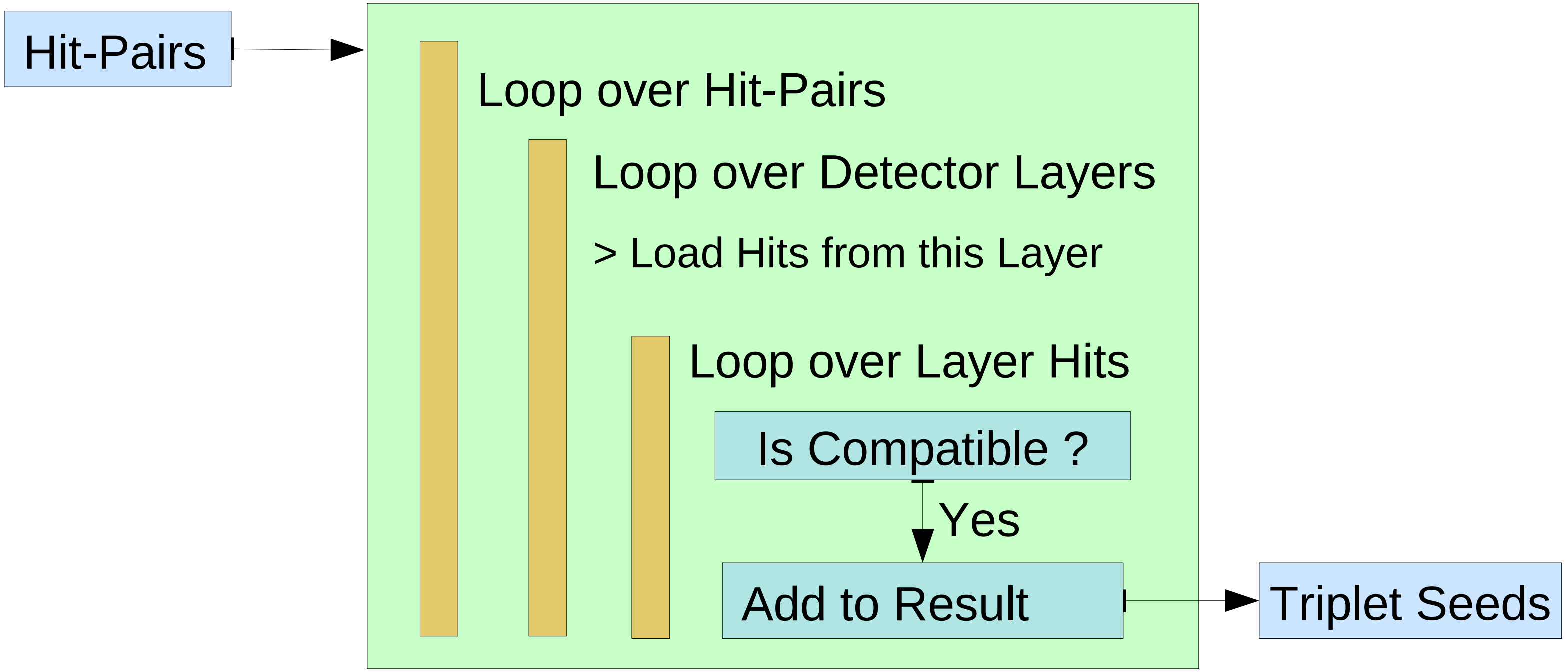
- Framework Parallelism
  - After some modifications (declaring dependencies etc.), parallel execution of already existing serial modules is possible
  - Hides most of the multi-threading complexity from the module developer
  - Scales very well at the price of loading and writing multiple events at the same time. See the presentation by Chris Jones*
- Algorithm Parallelism
  - Changes mostly contained in one module
  - Very lightweight scaling (in terms of memory)
  - Transparent to subsequent Modules
  - Most profitable to apply on long-running Modules which can only operate sequentially (like CMS Iterative Tracking)

A great potential lies in combining these two levels of parallelism: scale with the amount of input data and the number of available computing cores.

* Forum on Concurrent Programming Models and Frameworks, 14.03.2012
http://indico.cern.ch/conferenceDisplay.py?confId=181721

Triplet Seeding in CMS

- Energy deposits of charged particles in the CMS tracker are reconstructed as hits
- Before starting the track reconstruction, seeds from three topologically compatible hits in the tracker are searched: hit-triplets
- Starting with two hits which have been already found to be compatible (hit-pair) possible hits of subsequent tracker layers are evaluated
- This seeding procedure amounts to about 10% of the overall runtime of the CMS Reconstruction



Chosen Parallelization Technology: Intel TBB

- Intel Threading Building Blocks (TBB) 4.0 update 3 Open Source (GPL license)
- Compiled with GCC 4.6.2 (default compiler of CMS Software)
- Very nice integration with C++ (in contrast to OpenMP or OpenCL):
  - Templated thread-safe containers and other data types
  - Encapsulate parallel code segments in C++11 lambda expressions
- The package provides:
  - Loop parallelism constructs
  - Concurrent containers
  - Locking constructs
  - Atomic operations
  - Memory allocation
  - Task-Based programming model

  The green functionalities have been used in our implementation

- TBB was picked for this work due to its complete function set and easy deployment on ScientificLinux, but various other technologies are evaluated by CMS (libdispatch, OpenCL, ...)

http://threadingbuildingblocks.org/

Extension of the CMS Software Framework

- Only very punctual and small changes were necessary to accommodate this parallelization in the framework
- Services within CMSSW provide common functionalities to all Modules
- A TBB Service was created
  - Preserves a Thread Pool over event boundaries
  - Modules can query how many threads they can use for their parallel processing and partition their work chunks accordingly
  - The number of threads can be set in the python CMSSW configuration

```
process.TBB = cms.Service( "TBB" ,
                           threadCount = cms.untracked.uint32(6) )
```

  - If the TBB Service is not loaded via the configuration file, the serial version will be run
- The CMSSW reference counting has been made thread-safe with atomic operations by using a TBB data type:

```
mutable unsigned int referenceCount_;
```

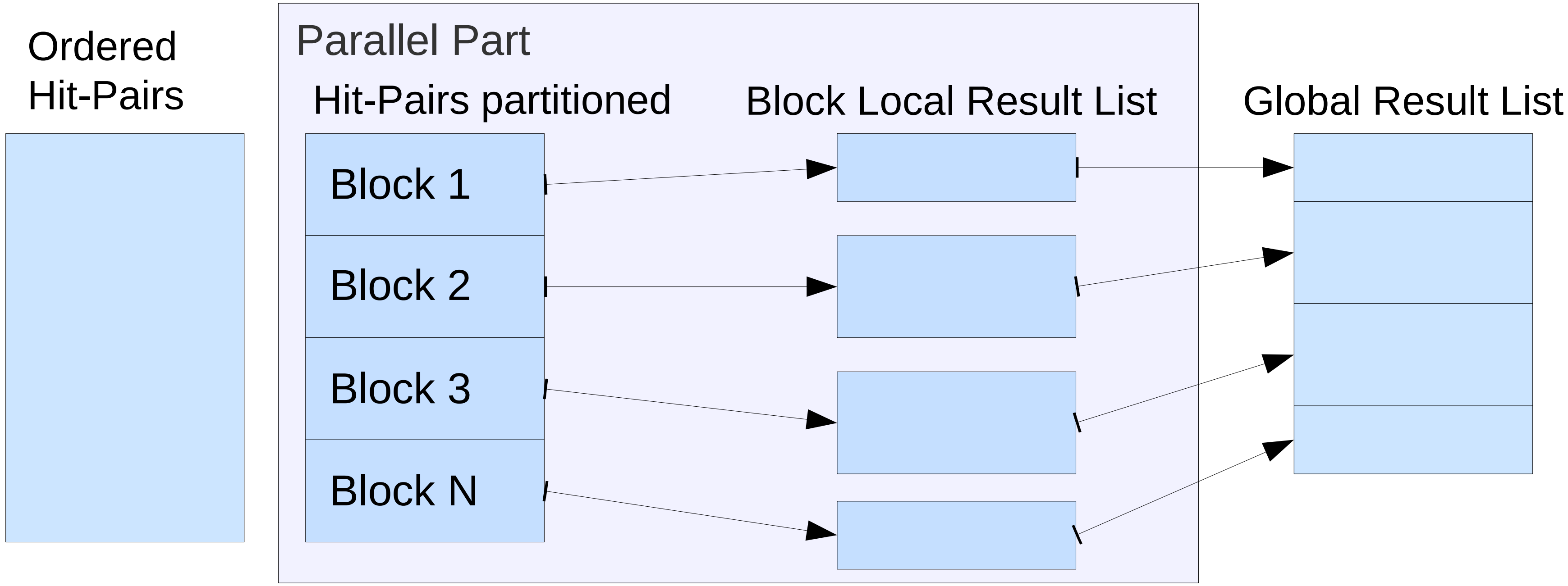
was changed to

```
mutable tbb::atomic<unsigned int> referenceCount_;
```

to guarantee atomicity when threads change the reference count

Triplet Seeding in Parallel

- Preserving the ordering of the output collection is essential for subsequent algorithms and validation purposes
- Filling an unsorted output collection with multiple threads at the same time can result in non-reproducible results
- We used a scheme to partition the input collection of hit-pairs in equally sized blocks
- A private result list is associated with every block and is merged in the correct order into the global result list at the end of the algorithm execution. No explicit sorting needed.
- The distribution of the blocks to the available threads is handled by TBB



How to ensure thread-safe code ?

- High quality of CMSSW code base helps, **`const`**-correctness enforced everywhere
- **`const`** is your friend:
  - **`const`** objects and methods can be accessed safely
  - But not always: C++ **`mutable`** keyword
  - Non-const variables can be assigned to a **`const`** reference to ensure safe access within the mutli-threaded code section:

```
AClass aobject(size);
AClass const& aobject_threadsafe = aobject;
```

- Use of TBB concurrent containers whenever multi-threaded write access to collections is necessary
- **`tbb::atomic`** data type was used to ensure thread safe reference counting
- Ultima-Ratio: Explicit Locking
- Software Tools for big applications:
  - Helgrind (part of valgrind) was tested on a simple example outside of CMSSW, but produced **many** false positives
  - Suggestions or hints are very welcome
- Use the serial implementation and run a lot of multi-threaded validation, check for crashes and compare the outputs
- Sourcecode of the seeding class: http://hauth.web.cern.ch/hauth/code/PixelTripletLargeTipGenerator.cc
  Methods: `hitTriplets_single` [regular implemenation], `hitTriplets_parallel` [TBB version]

Validation

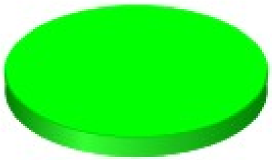
- We compared the multi-threaded version (10 threads) and the official (serial) release of CMSSW
- Considering 100 samples coming from the 2011 HighPU dataset
  - Comparing bin2bin all 43k Data Quality Monitoring (DQM) histograms did not reveal any difference
  - Tracks are 1:1 identical (momenta,chi2...)
- No crashes or segmentation faults have been observed in all test runs
- Large scale tests are of course needed but there is no reason to expect a difference

Part of a complete validation procedure using DQM histograms:

Tracking

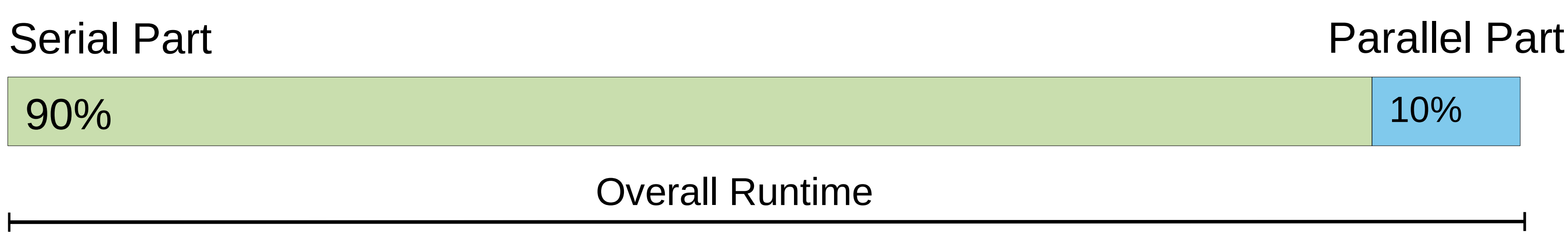
142 COMPARISONS:

- SUCCESS: 100.0% (142)

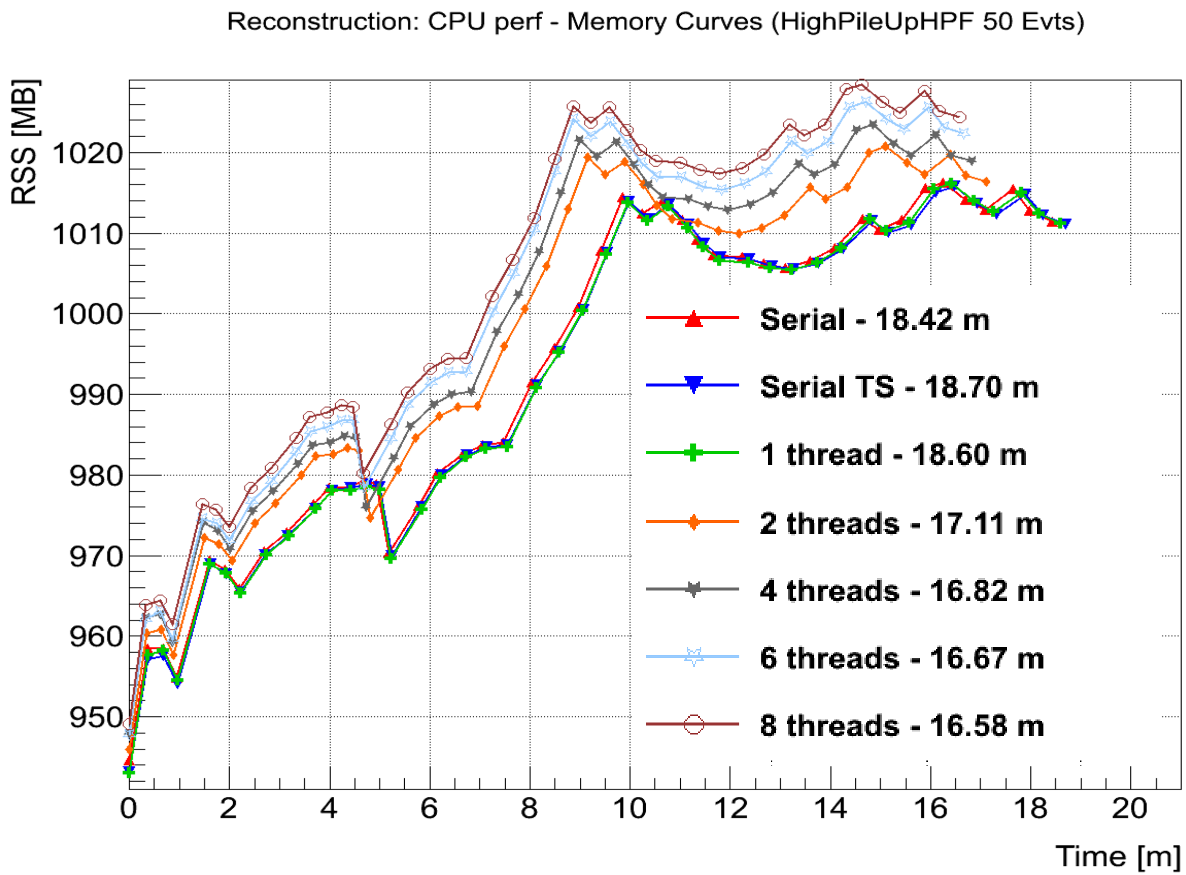


Performance Measurements

- The full CMS reconstruction chain (but: no output to disk) was run with different numbers of threads
- Input: 50 events of the highest pile-up sample recorded with the CMS detector in 2011
- On average, one event contains ~40 collisions
- Test Setup:
  - Intel(R) Core(TM) i7 CPU X 980 @ 3.33GHz with 6 physical cores
  - 6 GB RAM
  - Scientific Linux 5.8
  - CMSSW 5.2 official release (with modifications for the multi-threading code)
  - The measurements labeled *Serial* refer to an unchanged version of CMSSW (no TBB Service, no atomic operations)
- The triplet seeding takes about 10% of the runtime in the serial version
- Therefore, the maximum speed-up when running multi-threaded is 10% over the serial runtime



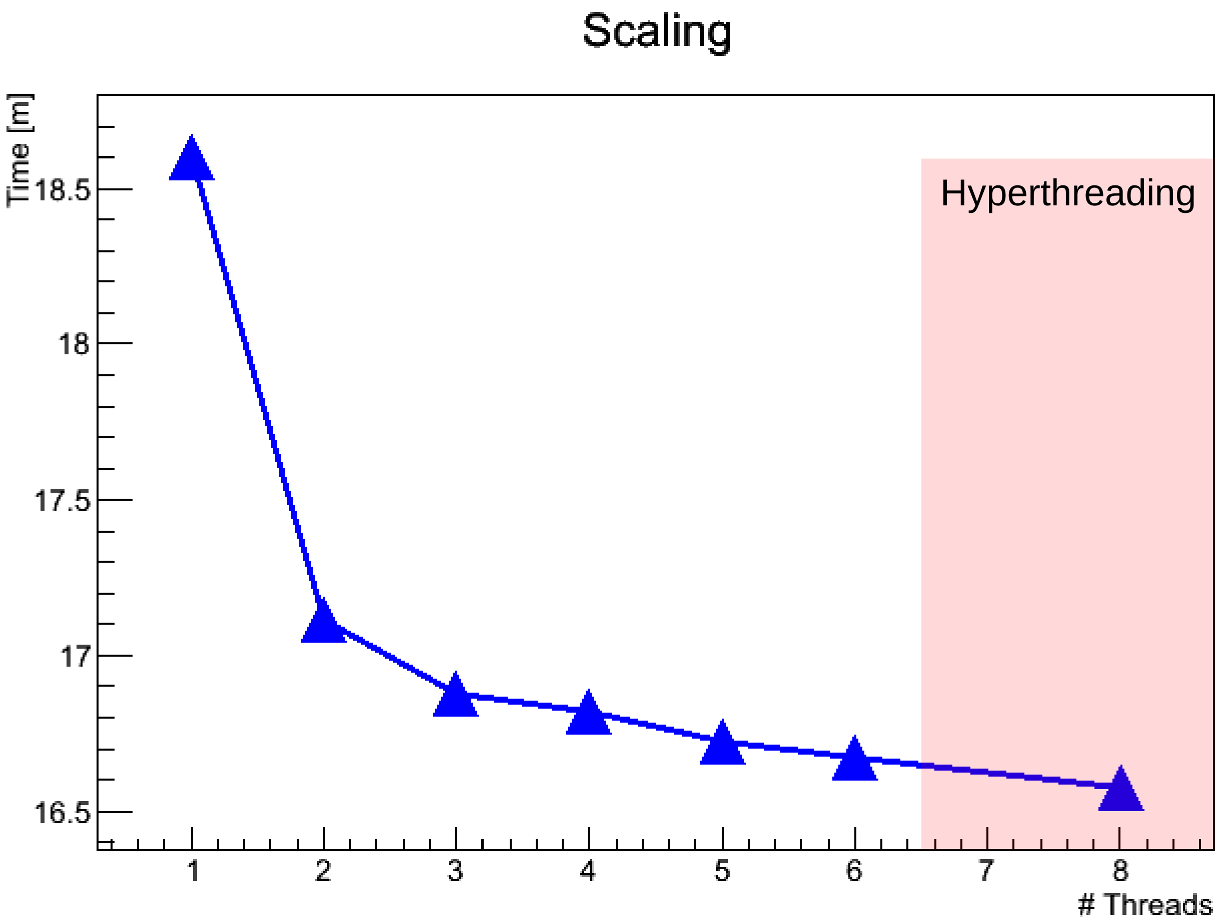
CMS Reconstruction Runtime and Memory



- Using thread-safe atomic reference counting for all data-structures adds about 1% to the overall runtime (Serial vs. Serial Thread-Safe)
- This effect can be reduced by using thread-safe reference counting only for data structures which are used in multi-threaded code
- Each thread adds less than 2 MB to the overall memory consumption

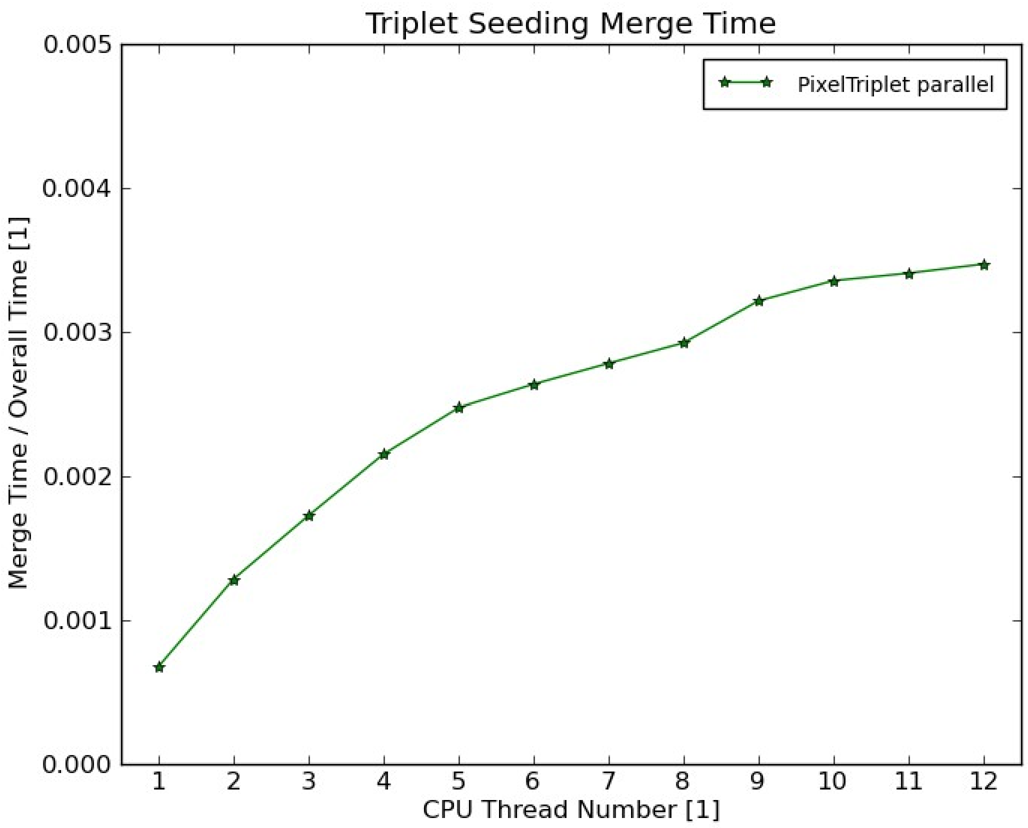
Scaling behavior of the Implementation

- Higher-than-expected scaling from 1 to 2 cores, probably due to the positive effects of using the L1/L2 caches of two cores simultaneously



Final Merge Overhead

- The thread-private work blocks are merged after the triplet seeding algorithm is complete
- Compared to the overall runtime of the algorithm, the merge step only takes about .1 to .3 percent of the triplet seeding time
- This depends on the number of threads: for more threads more blocks are partitioned



Hyperthreading: Food for Thought

- Intel Hyperthreading is disabled in CMS Tier-0 because of memory boundaries
- With a multi-threaded application we can use more (Hyperthreaded) Cores with very little memory overhead (less than 2 MB per Thread)

Test Scenario:
Slightly different Machine > need more RAM :)
Intel Core i7-3930K CPU at 3.20GHz
6 Physical Cores (12 Hyperthreaded)
16 GB RAM
Scientific Linux 6.2
50 High-Pileup Data Events

Runtime of **6 Single-Threaded** CMSSW Applications: **14.40 min +/- 0.10 min**

Runtime of **6 Two-Threaded** CMSSW Applications: **13.79 min +/- 0.08 min**

Using the Hyperthreading of the machine results in a decrease in runtime of 4.3 %
This number is very close the theoretical decrease of 5% with two threads. The cache benefit is not visible here, as the Hyperthreading can only use the cache of the 6 physical cores.

A good way to utilize the already purchased resources ?

Conclusions

- A multi-threaded track seeding using TBB was implemented within the CMS Software Framework
- Much more than a prototype: Tested and validated in a production environment with actual CMS proton-proton data
- By separating the input in blocks, the multi-threaded implementation produces exactly the same output as the serial implementation, independently of the number of threads
- The implementation scales as expected with number of available cores
- The memory consumption of additional thread is very moderate:

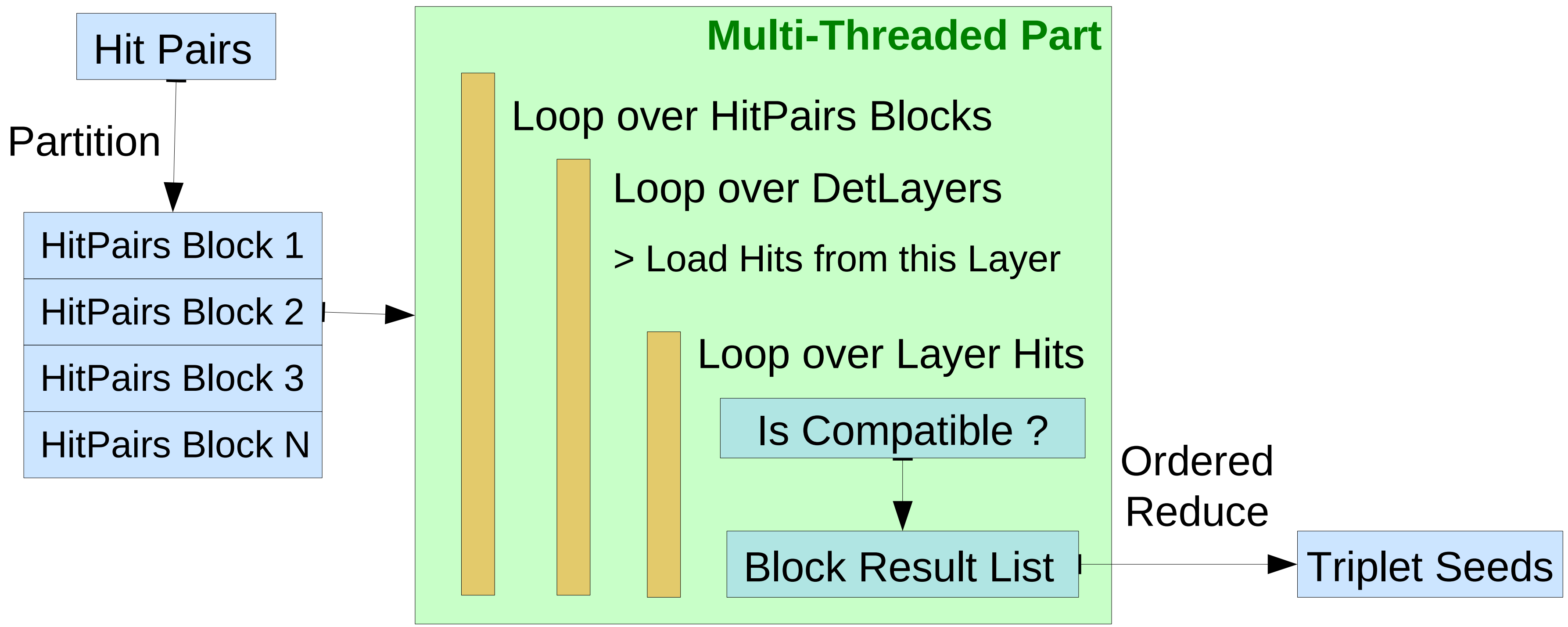
  ~2MB/ thread
- Algorithm Parallelism is a feasible way to speed-up long-running and serial module chains

Thanks to Benedikt Hegner, Chris Jones and Lassi Tuura for the fruitful discussions

BACKUP

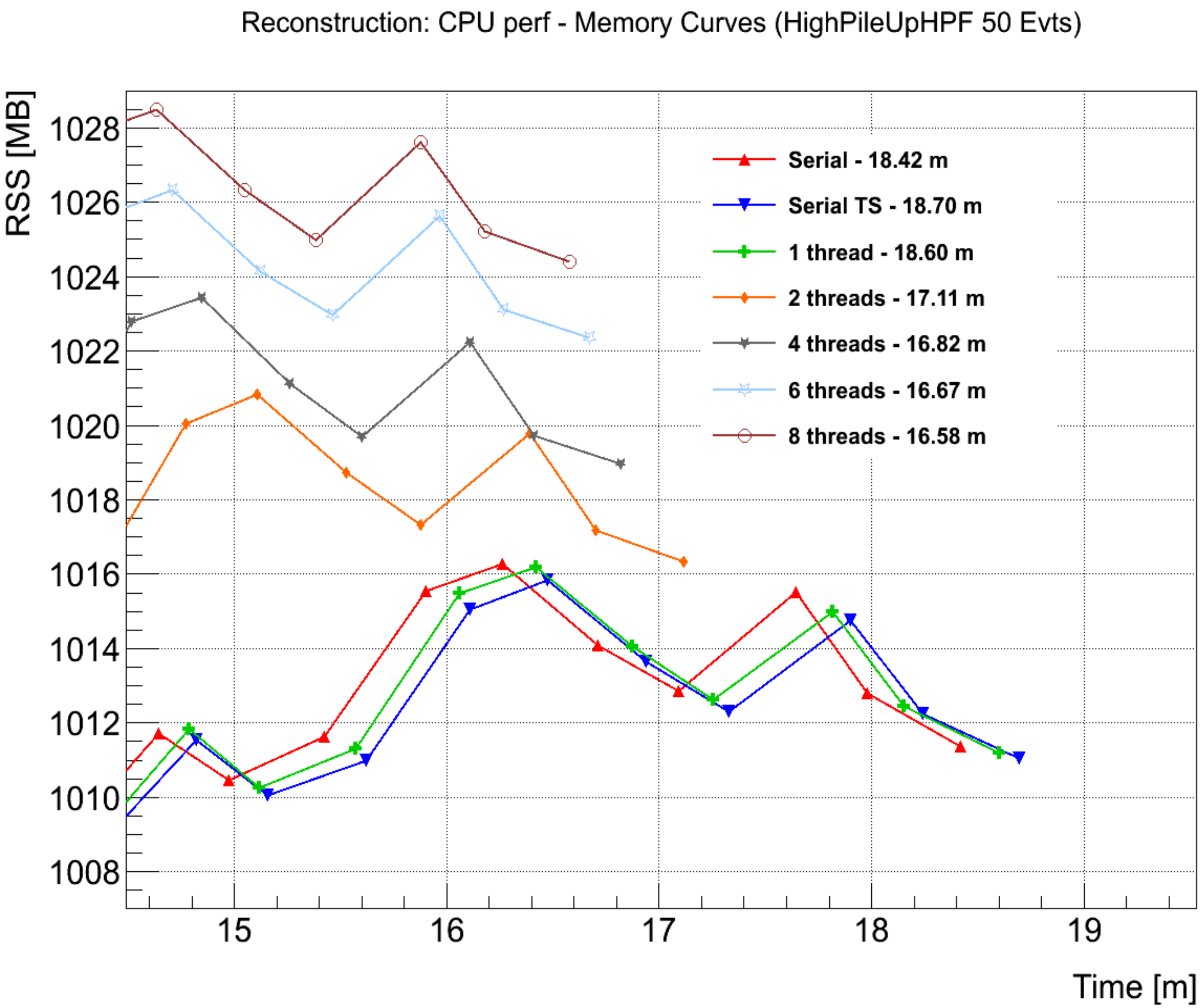
Triplet Seeding in CMS: Parallel Execution

- Before running the multi-threaded code part, the Hit-Pair list is partitioned into N equally sized work chunks
- The available threads process the HitPair Blocks via TBB's `parallel_for` method and stores the resulting TripletSeeds in a Block-local Result List
- TripletSeeds resulting from a HitPairs Block are merged into the output collection respective to their order in the input collection
- This guarantees the order of the output is not depending of the amount of threads



Full CMS Reconstruction Runtime and Memory

Zoom on the End Region



Source Code Excerpt – Private Result Lists

```
tbb::parallel_for(
        tbb::blocked_range<size_t>(0, pairs.size(),
        // ensure we do as little blocks as possible, cause we have this local array overhead
        std::max( pairs.size() / tbbService->GetThreadCount() / 4, (unsigned int )1) ),
            [&] (const tbb::blocked_range<size_t>& pairs_block)
            {
            std::vector< OrderedHitTriplet > * thread_local_result =
                    new std::vector< OrderedHitTriplet >();
            size_t loc = (size_t) pairs_block.begin();

            // create local result list
            {
                    ThreadLocalResultsMutexType::scoped_lock lock;
                    lock.acquire( threadLocalResultsMutex );
                    thread_local_results.insert( loc, thread_local_result);
            }
            ....
```

Full Source: http://hauth.web.cern.ch/hauth/code/PixelTripletLargeTipGenerator.cc

Source Code Excerpt – Final Merge

```
result.reserve( result_seeds );

// fill the result list in the order, the HitPairs were
for ( LocalResultMap::const_iterator map_it = thread_local_results.begin();
          map_it != thread_local_results.end();
          ++ map_it )
{
          for (std::vector< OrderedHitTriplet >::const_iterator it = map_it->second->begin();
                    it != map_it->second->end();
                    it ++ )
                    {
                              result.push_back( *it );
                    }
}
```

Full Source: http://hauth.web.cern.ch/hauth/code/PixelTripletLargeTipGenerator.cc