Efficiently exploit multicore architectures

The LHCb experience

Sébastien Ponce
sebastien.ponce@cern.ch
Outline

Context - LHCb and computing

Multi-threading and scheduling

Memory management

Results and lessons
Context - LHCb and computing

Multi-threading and scheduling

Memory management

Results and lessons
LHCb Run 3 landscape

- Upgrade of the detector itself to take more luminosity (x5)
  - still 30MHz collisions
  - more pile-up (now 5.5, was 1.1)
LHCb Run 3 landscape

- Upgrade of the detector itself to take more luminosity (x5)
  - still 30MHz collisions
  - more pile-up (now 5.5, was 1.1)
- New trigger system
  - no hardware, fully software
  - input rate x30!

Run 2

<table>
<thead>
<tr>
<th>Experiment</th>
<th>30MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware trigger</td>
<td>1MHz</td>
</tr>
<tr>
<td>Software Trigger</td>
<td>2 levels (HLT1, HLT2)</td>
</tr>
</tbody>
</table>

Run 3

<table>
<thead>
<tr>
<th>Experiment</th>
<th>30MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Trigger</td>
<td>2 levels (HLT1, HLT2)</td>
</tr>
</tbody>
</table>
**Computer science landscape**

- Hardware evolution continues
  - Moore’s law still holding
  - in numbers of transistors

- Hardware always more complex
  - more parallelization
  - pipelines, fuse multiple add, hyperthreads, vectors, ...

- Many-core area has started
  - easily 40, up to 100 logical CPU cores

Data source: https://github.com/karlrupp/microprocessor-trend-data, modified to only show transistors
More computer science landscape

CPU versus Memory improvements in last decades

- Memory is now extremely slow (relatively)
- Level of caches have been introduced to mitigate
- Good usage of caches has become a must
How to adapt?

- Multi-core architecture asks for multi-threading
  - and careful scheduling
- Memory management is of utter importance
  - while it had been neglected in the past
  - and thus in our code bases
- Low level optimizations can make a difference
  - and in particular vectorization
  - this will be the topic of Arthur’s talk
Multi-threading and scheduling

Context - LHCb and computing

Multi-threading and scheduling

Memory management

Results and lessons
Why not multi-job?

• Because it exhausts easily the memory
  • think of an application needing 10GB of memory
  • launch it 256 times on a KNL machine...
  • mitigation exists, but no more sufficient

• Because it harms the memory caches
  • jobs are competing for memory
  • while threads are cooperating, as they share most of it
  • resulting in performances gains (20% for LHCb)
Why not multi-job?
Implications of multi-threading

All code needs to be reentrant
Implications of multi-threading

All code needs to be reentrant

non-reentrant code

```c
void MyClass::handleXYZ {
    ...
    m_xyzCounter++;
}
```
Implications of multi-threading

All code needs to be reentrant

non-reentrant code

```cpp
void MyClass::handleXYZ {
    ...
    m_xyzCounter++;
}
```

- Hard to identify non reentrant code!
- Need to review all the code
- Implies major changes in coding habits
A practical approach in LHCb

Use the framework of the experiment

- Users write algorithms
  - their entry point is the `operator()` method
  - which now has to be reentrant
- Which interact with a white board
  - items in the whiteboard are now immutable
  - so you can no more modify them once created

Use latest C++ features

- `const` means “bit-wise constant or thread-safe”
- Hence `const` methods of classes are reentrant
- Thread unsafe code leads to compile errors
Make sure all cores are busy

Constraints

- Each thread needs to run independant tasks
  - avoid contention and false sharing
- Still some time dependencies

Consequences

- A directed acyclic graph of tasks
- “scheduling” needed
**LHCb’s HLT1 example**

**Tasks**

- Only use event level parallelism
- No intra event multi-threading
- One event is only 1ms of CPU

**Scheduling**

- Static scheduling
- Graph solved at initialization time
  - and converted to linear sequence
Memory management

Context - LHCb and computing

Multi-threading and scheduling

Memory management

Results and lessons
Remember Memory is really slow

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data</td>
<td>64 kB</td>
<td>4 cycles</td>
</tr>
<tr>
<td>L1 inst.</td>
<td>256 kB</td>
<td>10 cycles</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>10 MB</td>
<td>40 cycles</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>64 GB</td>
<td>400 cycles</td>
</tr>
<tr>
<td>DRAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Typical data, on an Haswell architecture
Remember Memory is really slow

<table>
<thead>
<tr>
<th>Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 kB</td>
<td>4 cycles</td>
</tr>
<tr>
<td>256 kB</td>
<td>10 cycles</td>
</tr>
<tr>
<td>10 MB</td>
<td>40 cycles</td>
</tr>
<tr>
<td>64 GB</td>
<td>400 cycles</td>
</tr>
</tbody>
</table>

Typical data, on an Haswell architecture

Cost of an access to RAM

- 400 cycles, that is of the order of 10 Kflop!
Memory management strategy

- Limit seeks and jumps to the minimum
  - to load all in one single access
  - i.e. collocate what goes together
- Limit memory allocations to the minimum
  - the number of them, not the size
  - so group many allocations into one
Example of bad code (1)

```cpp
std::vector<Track*> myTracks;
for (...) {
    myTracks.push(new Track(...));
}
```

- Each new track is an allocation
- Tracks are completely scattered in memory
Example of bad code (1)

```cpp
std::vector<Track*> myTracks;
for (...) {
    myTracks.push(new Track(...));
}
```

- Each new track is an allocation
- Tracks are completely scattered in memory

Rule 1: no container of pointers!

at least when they own their content
Example of bad code (2)

```cpp
std::vector<Track> myTracks;
for (...) {
    myTracks.push(Track(...));
}
```

- Vector will get reallocated many times
- And existing items copied over
Example of bad code (2)

```cpp
std::vector<Track> myTracks;
for (...) {
    myTracks.push(Track(...));
}
```

- Vector will get reallocated many times
- And existing items copied over

Rule 2: reserve space in your containers!
Example of bad code (3)

```cpp
std::vector<Track> myTracks;
myTracks.reserve(100);
for (...) {
    myTracks.push(Track(...));
}
```

- Tracks get copied
- They should be created directly in place
Example of bad code (3)

```cpp
std::vector<Track> myTracks;
myTracks.reserve(100);
for (...) {
    myTracks.push(Track(...));
}
```

- Tracks get copied
- They should be created directly in place

Rule 3: use emplace!
Do you think this is optimal?

```cpp
std::vector<Track> myTracks;
myTracks.reserve(100);
for (...) {
    myTracks.emplace(...);
}
```

Of course not!

- Use `std::array` or `boost::small_vector`
- And wait for Arthur's talk for more!
Do you think this is optimal?

```cpp
std::vector<Track> myTracks;
myTracks.reserve(100);
for (...) {
    myTracks.emplace(...);
}
```

Of course not!

- Use `std::array` or `boost::small_vector`
- And wait for Arthur’s talk for more!
Memory management and threading

• Heap allocations are serialized
• Too many new/malloc/… will lead to contention
• Another good reason to reduce their usage
Memory management and threading

- Heap allocations are serialized
- Too many new/malloc/… will lead to contention
- Another good reason to reduce their usage

Example of a bad case on 40 virtual cores:

![CPU Usage Histogram]

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.
Detecting memory offending code

• Measure time spent in malloc/new/free/delete/... ?
  • more than a few % ? Room for improvement!
• What is your last level cache miss rate ?
  • above 1% ? Room for improvement!
Detecting memory offending code

- Measure time spent in `malloc/new/free/delete/...`? 
  - more than a few %? Room for improvement!
- What is your last level cache miss rate? 
  - above 1%? Room for improvement!

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator new</td>
<td>16.7%</td>
</tr>
<tr>
<td>_int_free</td>
<td>8.3%</td>
</tr>
<tr>
<td>PrPixelTracking::bestHit</td>
<td>5.7%</td>
</tr>
<tr>
<td>PrForwardTool::collectAllXHits</td>
<td>5.7%</td>
</tr>
<tr>
<td>PVSeed3DTool::getSeeds</td>
<td>2.7%</td>
</tr>
<tr>
<td>PrStoreFTHit::storeHits</td>
<td>4.5%</td>
</tr>
</tbody>
</table>
Results and lessons

Context - LHCb and computing

Multi-threading and scheduling

Memory management

Results and lessons
3 years of LHCb HLT1 performances

Multithreading, from 500 evts/s to 3500 evts/s

• Make the HLT1 code thread safe and scalable

Vectorization, 2x to 3x speedup per algo

• Vectorize key algorithms

Change event model, from 24K evts/s to 33K evts/s

• Adopt SoA and plain old data – see Arthur’s talk

Numbers measured on a “reference” machine, corresponding to 1% of the HLT1 farm capacity
Lessons

We can gain factors !

- Modern CPUs can be efficiently used
- And they are pretty good and fast actually

... not for free ...

- Deep changes in the code and data structures
- A change of paradigm, similar to the GPU

but it’s rewarding

- New code is shorter, faster and more readable !
Advises, learnt the hard way...

- Start by cleaning up your code
  - will save you unnecessary work
  - will already gain up to 2x in speed!
- Deal with memory before you go threaded
  - or the contention will be immediate
- Go to a simple event model
  - do not overdo object orientation
  - think structure of arrays from the beginning
- Only then vectorize
  - only if worth it, do not expect miracles
  - check expected gain with Amdahl’s law in mind
Final remark

- Computing has become very complicated
- Huge need for disseminating the knowledge
- This is the key point to success for run 3 and 4!