Optimizing existing large codebase

Sébastien Ponce
sebastien.ponce@cern.ch
CERN

Thematic CERN School of Computing 2021
Outline

1. Measuring Performance
   - What is performance?
   - Tools available
   - Finding bottlenecks

2. Code modernization

3. Improving Memory Handling
   - Context
   - Containers and memory
   - Container reservation
   - Detecting offending code

4. The nightmare of thread safety
   - Context and constraints
   - Identifying problems
   - Solving problems
   - Thread contention

5. Low level optimizations
   - Scope and target
   - How to measure?
   - Improving

6. Conclusion
Goal of this course

- make the theory explained so far more concrete
- and adapt it to the special case of
  - dealing with large projects
  - dealing with legacy code
- I’ll only talk of C++ projects
Specificity of the exercise

- Dealing with large code base (Mloc)
  - most of them unknown to you
  - and (usually) not supported by anyone

- Dealing with old code
  - using old fashion coding style (e.g. FORTRAN like)
  - modified n times, grew organically

- Target latest hardware
  - many cores
  - hyperthreading / superscalar
  - vectorization
Overall strategy

First measure!
- understand where time is spent
- understand the main limitations

Then attack these limitations
- modernize the code
- optimizing memory handling
- optimizing parallelism
- optimizing low level code
Defining our performance

Key question is: what is performance

- simply going faster?
  - not at all costs (money, physics results)
- making better use of the hardware
  - most of the time hardware is cheaper than people!
- you need to define your “Key Performance Indicators”
  - e.g. nb Evt/s/$ with constant man power for a trigger
- and get a clear idea of your different costs
  - flops/$ of your machines
    - including network, cabling, cooling, buildings, ...
  - human costs
  - cost of transition
  - ...

S. Ponce - CERN
Measuring our software

Many parameters can be measured

- overall timing
- memory usage and cache efficiency
- CPU efficiency (Cycles per instructions, vectorization level)
- level of parallelism, usage of the different cores
- I/O limitations if any

For each of them, you need

- both overall data and detailed split per code unit
- per item, per core and full machine measurement
How to measure

The counters approach
- use CPU counters to find out what happened during actual execution
- do not slow down execution, so only do sampling

The software instrumentation
- run your code in a “virtual” environment
- measure everything precisely
- at the cost of speed
Counters approach in practice

- give precise timing of a realistic execution on your CPU
  - using real cache prediction, actual vectorization, ...
  - using real CPU behavior (e.g. downclocking when overheating...)
- allows to measure CPI (Cycles Per Instruction) and low level behavior in general (caching, pipelining)
- but data is only statistical
  - so you need sufficient statistics
  - also not always reproducible, so hard to compare
    - e.g. first test on cold processor, second on warm one
- Main tools available: perf and variants, Intel VTune
Software instrumentation in practice

- give precise measurements of where you spend instructions
  - including many details
  - reproducible, so your can compare stuff
- but not always realistic
  - no real timing, only instructions count
  - memory caching is only simulated, often far from real case
  - no clue on low level efficiency (CPI in particular)
  - and gives no clue on hardware / OS behavior
- Main tool available: valgrind family
Finding bottlenecks

Understand where we can improve

- analyze each part of the software
- in order to find out where most time is spent
- and understand whether it can be improved

Most usual bottlenecks

From biggest to lowest impact (usually)

- IO
- Memory
- Parallelization
- Low level behavior: vectorization, cache behavior, high CPI
Make use of latest C++ features

- C++ has evolved dramatically between 2010 and now
- four new versions: C++11, C++14, C++17, C++20
- a LOT of new features targeting performance
  - move semantic
  - threading library
  - variadic templates
  - vectorization coming!
- converting existing code may already bring speed
- see previous courses for technical details
- see my extended C++ course if you’re not at ease with the language
While reviewing the code for converting to latest C++:
  
  - drop unused code
  - drop unnecessary code
    - e.g. do I really need to sort by hits here?
  - drop too generic APIs if they are finally not needed
  - replace virtual inheritance with templating when possible
  - consider dropping use of unmaintained libraries

- It is very often surprising how much you gain there
Due to Moore’s law in the 80s and 90s, there is a gap between CPU and memory performances.

Consequences:
- Access to memory is now extremely slow (relatively).
- Level of caches have been introduced to mitigate.
- Good usage of caches has become a key parameter.
Typical cache structure

- **L1 data**
- **L1 instruction**
- **L2 Cache**
- **L3 Cache**
- **DRAM**

Typical data, on a Haswell architecture:

<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>
Practical consequence in C++

Guidelines
- we want as few heap memory allocations as possible
  - stack usage is much better!
- we want continuous memory blocks, specially for containers
  - that means containers of objects, no pointers involved
  - e.g. `vector<Obj*>` or `array<vector<Obj>>` are banned!

2 main rules
- use container of objects, not of pointers
  - use (const) references everywhere
  - avoid any unnecessary copy of data
    - including implicit ones
- use container reservation
Container of objects in memory

Simple vector case

```cpp
std::vector<int> v;
```

```
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 ...
```

Vector of objects

```cpp
struct A { float x, y, z; };
std::vector<A> v;
```

```
x0 y0 z0 x1 y1 z1 x2 y2 z2 x2 ...
```

```latex
A_0 \quad A_1 \quad A_2
```
Container of pointers in memory

Naïve view

```cpp
struct A { float x, y, z; };
std::vector<A*> v;
```

![Naive view diagram]

Realistic view

![Realistic view diagram]
Optimizing existing large codebase

Container of objects in cache

<table>
<thead>
<tr>
<th>Memory view for vector&lt;A&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each line corresponds to a cache line (64 bytes, 16 floats)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00C0</td>
<td>· · · · · · · · · · · · · ·</td>
</tr>
<tr>
<td>0x0080</td>
<td>x0 y0 z0 x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4 x5</td>
</tr>
<tr>
<td>0x0040</td>
<td>y5 z5 x6 y6 z6 x7 y7 z7 x8 y8 z8 x9 y9 z9 · ·</td>
</tr>
<tr>
<td>0x0000</td>
<td>· · · · · · · · · · · · · ·</td>
</tr>
</tbody>
</table>

All data are nicely collocated in cache
### Container of pointers in cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0240</td>
<td>x8, y8, z8</td>
</tr>
<tr>
<td>0x0200</td>
<td>x9, y9, z9</td>
</tr>
<tr>
<td>0x01C0</td>
<td>x7, y7, z7</td>
</tr>
<tr>
<td>0x0180</td>
<td>x5, y5, z5</td>
</tr>
<tr>
<td>0x0140</td>
<td>x6, y6, z6</td>
</tr>
<tr>
<td>0x0100</td>
<td>x3, y3, z3</td>
</tr>
<tr>
<td>0x00C0</td>
<td>x4, y4, z4</td>
</tr>
<tr>
<td>0x0080</td>
<td>x2, y2, z2</td>
</tr>
<tr>
<td>0x0040</td>
<td>x0, y0, z0</td>
</tr>
<tr>
<td>0x0040</td>
<td>p0, p1, p2, p3, p4, p5, p6, p7, p8, p9</td>
</tr>
</tbody>
</table>

Cache nightmare: data is completely sparse

Note from Andrzej: this is already optimistic
Container with no reservation

```cpp
struct A { float x, y, z; };
std::vector<A> v;
```

**Construction**

Default constructor creates and empty vector, with no storage

<table>
<thead>
<tr>
<th>start</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish</td>
<td>0x0</td>
</tr>
<tr>
<td>end_of_storage</td>
<td>0x0</td>
</tr>
</tbody>
</table>

**First push**

allocates storage for the first element only!

- 0x1234
- 0x1240
- 0x1240
Container with no reservation

Second push

1. allocate new piece of memory for 2 items
2. copy existing content
3. write new content
Container with no reservation

Second push

1. allocate new piece of memory for 2 items
2. copy existing content
3. write new content
4. update pointers
5. Deallocate original piece of memory
Container with no reservation

Third push

1. allocate new piece of memory for 4 items
   - double size at each iteration
2. copy existing content
3. write new content
Container with no reservation

Third push

1. Allocate new piece of memory for 4 items
   - Double size at each iteration
2. Copy existing content
3. Write new content
4. Update pointers
5. Deallocate original piece of memory
Optimizing existing large codebase

Container with proper reservation

Construction and reservation

- you can avoid all that thanks to reserve

```cpp
std::vector<int> v;
v.reserve(1000);
```

- ensures single allocation, no copies, no reallocation

Without second line

- first item would have be copied 10 times!
- 1023 items would have been copied in total
- 11 pieces of memory allocated, 10 released
Avoiding copy when filling container

What should be avoided

```cpp
1 std::vector<A> v;
2 v.reserve(10);
3 A tmp{args};
4 v.push_back(tmp);
```

What actually happens:
- allocate space in the vector (line 2)
- allocate space for the temporary A object (line 3)
- call A constructor (line 3)
- call copy constructor for A (line 4)
- deallocate temporary A (end of scope)
- using `std::move(tmp)` on last line not necessarily better
Proper solution for vectors

**In place construction**

1. `std::vector<A> v;`
2. `v.reserve(10);`
3. `v.emplace_back(args);`

What actually happens:

- allocate space in the vector
- call constructor for A
  - using args as the constructor arguments
  - using the space allocated in the vector

For the record, this is using variadic templates, new in C++11
Proper solution for maps

In place construction + piecewise construct + forward_as_tuple

1. `std::map<int,A> m;`
2. `m.emplace(piecewise_construct,
   make_tuple(5),
   forward_as_tuple(args));`

- emplace_back now creates an `std::pair`
- piecewise_construct constructs the 2 items of the pair in place
- forward_as_tuple prevents a copy of args
  - by using a tuple of references
Detecting memory offending code

Look at your measurements!

- How much time do you spend in `malloc/new/free/delete/...`? More than a few %? Room for improvement!
- What is your last level cache miss rate? 1 % or more? Room for improvement!
- Drill down to the method where the numbers are bad
- Find the responsible container / data structure

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator new</code></td>
<td>16.7%</td>
</tr>
<tr>
<td><code>_int_free</code></td>
<td>8.3%</td>
</tr>
<tr>
<td><code>PrPixelTracking::bestHit</code></td>
<td>5.7%</td>
</tr>
<tr>
<td><code>PrForwardTool::collectAllXHits</code></td>
<td>5.7%</td>
</tr>
<tr>
<td><code>PVSeed3DTool::getSeeds</code></td>
<td>2.7%</td>
</tr>
<tr>
<td><code>PrStoreFTHit::storeHits</code></td>
<td>4.5%</td>
</tr>
</tbody>
</table>
Evolution of CPUs in the past decades

- clock speed is now fixed
- flops come now from parallel processing
- and especially from many cores

source: https://github.com/karlrupp/microprocessor-trend-data
Adapting existing software: multiprocessing

Cheap and easy approach

- works when you can split your data
- just process the pieces in parallel
- launching your original application n times
  - where n is roughly the number of cores
- and glue the results together

Main problem

- the memory usage (amount and cache efficiency)
Adapting existing software: multithreading

Less easy but rewarding approach

- remember threads share their heap
- your memory is mainly “read only” during processing
  - all code and libraries
  - detector geometry, conditions
- threads share this common memory
  - 40 threads instead of 40 jobs means memory usage divided by 40!

Main problem

- race conditions on non read-only memory parts
To Be Precise: Data Race

Standard language rules, §1.10/4 and /21:

• Two expression evaluations conflict if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

• The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behaviour.
Simple Example

Concurrency can compromise correctness

- Two threads: A and B, a variable X (44)
- A adds 10 to a variable X
- B subtracts 12 to a variable X

2 Threads only
No crash
Bogus results!

<table>
<thead>
<tr>
<th>A then B</th>
<th>Desired</th>
<th>B then A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread A</strong></td>
<td><strong>Thread B</strong></td>
<td><strong>X Val.</strong></td>
</tr>
<tr>
<td>Read X (44)</td>
<td>Read X (44)</td>
<td>44</td>
</tr>
<tr>
<td>Add 10</td>
<td>Subtract 12</td>
<td>44</td>
</tr>
<tr>
<td>Write X (54)</td>
<td>Subtract 12</td>
<td>54</td>
</tr>
<tr>
<td>Write X (32)</td>
<td>Add 10</td>
<td>32</td>
</tr>
<tr>
<td>Write X (42)</td>
<td>Write X (42)</td>
<td>42</td>
</tr>
</tbody>
</table>

RACE

Danilo Piparo – CERN, EP-SFT
What is not Thread Safe?

Everything, unless explicitly stated!

In four words: **Shared State Among Threads**

Examples:

- Static non const variables
- **STL containers**
  - Some operations are thread safe, but useful to assume none is!
  - Very well documented (e.g. [http://www.cplusplus.com/reference](http://www.cplusplus.com/reference))
- Many random number generators (the stateful ones)
- Calls like: `strtok`, `strerror`, `asctime`, `gmtime`, `ctime` ...
- Some math libraries (statics used as cache for speed in serial execution...)
- Const casts, singletons with state: indication of unsafe policies

*It sounds depressing.* But there are several ways to protect thread unsafe resources!
Converting to multithreading

Identify all shared state
and secure them
Identifying problematic shared states

A very hard topic!

Level 0: use gdb and debug crashes
- but race conditions do not happen at every run
- and crashes is the lucky case, wrong results is the other

Level 0.5: helgrind
- detects race conditions that did not happen
- but only in code that ran (coverage issue)
- and is very slow
Optimizing existing large codebase

Identifying problematic shared states

Actual (partial) solution: use the language

- C++11 constness is different from original one
- It now means “visibly const and race condition free”
- That is the “visible” state is not modified
- And internally thread safety is guaranteed if anything changes

Bottom line: proper use of constness can save us!

- Especially constness of member functions
- It ensures they are reentrant
- It will tell us at compile time if not :-)
Practically identifying shared state

- introduce constness everywhere
- look at compiler errors
- and look for globals, const does not apply here
- and do not forget special cases where constness is not respected...
  - mutables
  - const_cast, C casts
- better be fluent with grep...
Typical legacy code

```cpp
/// hack to allow for tools with non-const interfaces
template <typename IFace>
IFace* fixup(const ToolHandle<IFace>& iface) {
    return &const_cast<IFace&>(*iface);
}

void CaloHypoNtp::operator()(...) const {
    ...
    if( m_checker )
        fixup(m_2MC)->from(hypo)->descriptor();
    ...
}
```
Securing a shared state

**Non optimal way**: locks and mutexes

- Serializes access to given piece of code
- Fine under low pressure
- But costly and leads to contention under high pressure
  - See later

**Better way**

- Analyze the data race, extract the state
- Can it be replicated/moved to higher level?
Optimizing existing large codebase

State replication option

When to use it

- when the state can be replicated
- so not used as a synchronization point
- for heavily used, small state

Example: random number generator

- state is small, does not need to be shared
- so you can have one generator per thread
- or modify the random generator
  - and use thread local storage for the state
  - although may be more costful
Moving state to higher level

When to use it

- when the state can be replicated
- so not used as a synchronization point
- fine with small and large states
- typically for caches, when the state is data reused between calls

Principle

- simply make the state visible to the caller
- and initialize it one level higher on the stack
- so that you have local states per thread

Think of the state as hidden data passed magically to your calls, just make it explicit
Optimizing existing large codebase

State moving option example

Original class

```cpp
class CachedAccessor {
    mutable CachedObject m_myCache;
    Item get(const KeyObject& key) const {
        ...
        m_myCache.update(...);
        ...
    }
};
```

Usage

```cpp
CachedAccessor accessor{dataSource};
for(const auto& key: keyList) {
    auto item = accessor.get(key);
    ...
}
```
State extraction to higher level

Thread safe version of the class

class CachedAccessor {
    Item get (const KeyObject& key, CachedObject& cache) const {
        cache.update(... key ...);
        ...
    }
};

thread safe usage

CachedAccessor accessor{dataSource};
CachedObject cache;
for(const auto& key: keyList) {
    auto item = accessor.get(key, cache);
    ...
}
What do we call contention?

- a situation where all hardware resources cannot be used efficiently because of one bottleneck in the chain called point of contention
  - can be anything
    - hardware: network, I/O, ...
    - lines of code, typically with a lock
    - architectural: one producer not fast enough blocking many consumers
  - actually can be anything shared between threads
Optimizing existing large codebase

How to detect contention?

- benchmark the code
- look at scalability
- look at thread usage

Example output of 40 threads on 20 physical cores’ system

**Effective CPU Utilization 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 utilization value.
Where is the contention?

Much more difficult question!
- you need to find the hotspot
- without any clue of its type
- look for
  - high CPU in a single thread
  - lengths of your queues (too full, too empty?)
  - global state of your machine (I/O, network)
Example of memory based contention

**Figure:** Table of the main CPU consumers in the application

<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>

What was happening:
- Too many small memory allocations across threads
- New has an internal locking for thread safety
- And cannot cope!

**Another good reason to care about memory allocation!**
Scope and target

- improve usage of the hardware for a given code
- that includes in particular
  - better caching of memory accesses
  - better usage of pipelining and superscalar features
  - better vectorization
- in principle, the compiler does all that for you
- but you have better high level knowledge of the code!
Measuring low level efficiency

CPI

- Cycles Per Instruction
- should be smaller when pipelining and superscalar features are used
- but typical values highly depend on the type of code

Cache misses

- should be really low (<1%)
- remember Andrzej’s slide: “5% is catastrophic”

Vectorization

- look at specific counters in vtune/valgrind (extension)
- check the assembly code directly looking for SIMD instruction codes
How to improve

By acting mainly on 2 constraints that the compiler has to obey

**your data dependencies**
- you need close-by operations to be independent
- compiler code reordering is limited
- and pipelining/superscalar need that

**your data structure**
- especially true for caching and vectorization
- your data structures should group together what goes together
- Called SoA (Structure of Arrays) approach

In both cases: work more in parallel!
Conclusion

Key messages of the day

- typical large old code can benefit a lot of optimizations
  - typically speedup of $> 2$!
- main lines of optimizations is reworking the data structures
  - for optimizing memory allocation
  - for allowing more parallelism/vectorization
- handling multicores typically goes through threading
  - huge effort to be made to ensure thread safety
  - constness can greatly help