optimization of large code base

Sébastien Ponce

October 7, 2025

1 Foreword

In this exercise, we will play with (a small subset of) the LHCb first level trigger code.

The code is reading real LHCb raw data and executing the first phase of the reconstruction, that is the tracking inside the Velo (Vertex Locator) subdetector. The code is (almost) the original code that was there before a huge effort was started in LHCb to optimize and speed up the software in view of the run 3 upgrade.

There are a lot of improvements that can be achieved in this code, and the goal of the exercise is to help you to find some of them, improve the code and measure the gains. In a second step, you will try to run the code in a multi-threaded mode. You will first need to make some fixes on the non-reentrant parts of the code, before you can try to optimize the throughput.

The main tool that we will use is valgrind, the open source suite of tools dedicated at debugging and profiling. We will in particular use callgrind but also the less known helgrind. Finally we'll have a quick view of mutrace, a useful little tool to debug thread contentions.

For this exercise we will mainly be using remote phsical machines at CERN. This is especially important for the threading part of the exercise where having a good number of cores helps. The exercise is using the linux command line and the many tools provided in a standard linux distribution. For the ones you do not know, please open the man page and read at least the description. Also feel free to ask any questions.

We will also use your local machine for graphical tools. This will require to install some software. It is recommended to work under linux for that, but other operating systems should also be usable if you cannot. Just choose linux if you have a choice.

2 Goals of the exercise

- measure behavior of a "large" application
- detect and solve inefficiencies
  - unadapted data structures
  - non rentrant code
  - thread contention
- learn how to use several useful opensource tools
  - callgrind
  - helgrind
  - kcachegrind

- htop
- mutrace

3 Setup

Note that the instructions of this document are linux oriented. In case you're using a different operating system on your local machine, you'll need to adapt some parts. However you should be able to work on any platform.

If not under linux, you may actually skip the sshfs configuration part. You will then have to deal with file transfers by hand and log to lxplus manually before attempting to log to the CERN servers.

3.1 setup of remote machine

Check which username and machine were assigned to you. These machines have with 2 CPUs and 10 physical/20 logical cores per CPU. In order to simplify measurements, each of you has been assignated a given CPU within his/her machine. This also avoid interferences with the student working on the other CPU.

Let's now log to provided machines at CERN and set them up.

- setup your ssh to be able to reach csc machines directly proxying through lxplus. Practically add the following lines to file ~/.ssh/config. Create it if needed, change lxplus to lxplus-old if needed:

  ```
  Host csc-2025*
      ProxyJump lxplus.cern.ch
  ```

- open a terminal and login with ssh. Note that you'll go through lxplus automatically

  ```
  ssh <userName>@csc-2025-<nn>
  ```

- make sure you're using only your CPU. Replace 0-15 by 16-31 if you're using CPU 1

  ```
  numactl -C 0-15 $SHELL
  ```

  Note that you should use this command in any new shell that you would open on the machine

- create a workspace in /tmp and clone the course's repository there

  ```
  cd /tmp
  mkdir -p <username>; cd <username>
  git clone https://gitlab.cern.ch/sponce/tcsccourse.git
  ```

- setup the environment to use a recent gcc compiler, here 15.1.0 and a recent valgrind, here 3.25.0

  ```
  source /cvmfs/sft.cern.ch/lcg/views/LCG_108/x86_64-el9-gcc15-opt/setup.sh
  ```

- go to exercise4 and prepare configure using cmake

  ```
  cd tcsccourse/exercises/exercise4
  mkdir build
  cd build
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```

- open another shell on the same machine and run htop inside it

  ```
  ssh <userName>@csc-2025-<nn>
  htop
  ```

  `htop` tells you which cores are running something. You should see the cores appearing in 4 columns of 8. First column corresponds to physical cores in CPU 0, second to logical cores in CPU 0. Third and fourth columns are the physical and logical cores of CPU 1. You can this way see the activity on the machine, including from the other student on the same node.

- compile using all cores in original shell, letting htop run

  ```
  make -j 20
  ```

  You should see the columns corresponding to your CPU going quite busy. Check these are the expected ones and you did not put the wrong CPU.

- prepare a reasonably big input file from the one in the git repository

  ```
  touch LHCbbig.mdf
  for i in `seq 10`; do cat ../LHCbEvents.mdf >> LHCbbig.mdf; done
  ```

- run the code and time the original status. Note that you are running single threaded so a single core will get busy, although the core used may fluctuate over time

  ```
  time ./FakeLHCb LHCbbig.mdf ../geoData.bin
  ```

Original duration -

3.2 share files with your local machine

This last step allows to mount the remote machine files on your local machine so that they appear to be local.

- on your local machine, open a terminal
- create the same directory[1] in /tmp as you did on the CERN server and mount it via sshfs

  ```
  cd /tmp
  mkdir -p <username>
  sshfs <username>@csc-2025-<nn>:/tmp/<username> <username>
  ```

- you should now see your remote files as if they were local

  ```
  ls <username>
  ```

[1] in case or impossibility, you can use any directory but you will have to use `sed` or any find/replace tool on the callgrind output files later and change all paths from the ones on the CERN machine to the ones on your local machine

4 General exploration with callgrind

4.1 Running `callgrind`

Callgrind allows to record the execution of a program and build statistics on where instructions were spent, in terms of functions and lines of code. By default, it only deals with functions, but the `--dump-instr=yes` allows to also have details per line of code. Last but not least, we will recompile in debug mode so that functions names are readable.

One of the "features" of the valgrind family is that it will slow down the execution by a factor 20 typically. So it's a good idea to reduce the amount of events we run under callgrind. The good news is that the mdf format of LHCb allows to just cut the input file blindly :-)

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j 20
head -c 5000000 ../LHCbEvents.mdf > LHCbsmall.mdf
valgrind --tool=callgrind --dump-instr=yes ./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

After some 73 events, you should get an output file in the current directory called callgrind.out.xxxx where xxxx is the process id. This file is human readable but you want to use `q/kcachegrind` for nice visualization of the data. `qcachegrind` is the QT based gui, while `kcachegrind` uses KDE libraries for the same look and feel. In the rest of the document, I'll use kcachegrind by default although both are working the same.

4.2 installing `kcachegrind`

`kcachegrind` is a graphical tool. In order for it to be responsive despite the network latency with CERN, we will run it locally, so you'll have to install it. There are mainly 2 possibilities :

- for linux users, just use your native package manager (yum/apt/dnf/...) and install either qcachegrind or kcachegrind depending on what is available.
- for others, miniconda is the way to go :
  - go to `https://docs.conda.io/en/latest/miniconda.html`
  - download the suitable installer and run it, answer yes when needed
  - install qcachegrind using conda :

    ```
    ~/miniconda3/bin/conda install qcachegrind
    ```

4.3 Exploring callgrind's output

Open a new terminal on your local machine and launch kcachegrind on the callgrind output file. Remember it's mounted on your local /tmp thanks to sshfs.

```
cd /tmp/username
kcachegrind tcsccourse/exercises/exercise4/build/callgrind.out.*
```

Note in case you do not use sshfs

In case you could not mount the remote machine files via sshfs, here are some more instructions to get the callgrind output file locally.
Create the same subdirectory as on the CERN server and clone the git repository of the exercise into it, as you did on the server. Again it is important to have the same path on the local machine and on the server or you won't be able to look at the source code in kcachegrind. If not possible, see note 1 on page 3.

```
cd /tmp
mkdir <username>; cd <username>
git clone https://gitlab.cern.ch/sponce/tcsccourse.git
```

Then get the output file of callgrind from the CERN machine and open kcachegrind locally (not necessarily from the command line).

```
scp userName@tcsc-2023-<nn>:/tmp/<username>/
    tcsccourse/exercises/exercise4/build/callgrind.out.* .
kcachegrind callgrind.out.*
```

The kcachegrind environment is quite complex. Let's first setup a couple of things :

- check that "Instruction Fetch" is selected in the top bar, on the most right drop-down menu. This decides which type of measurement you want to look at. Note that we are not measuring time here, but number of instructions. In some cases, it can make a difference.
- check that the "Relative", or the "%" button is enabled on the left, still in the top bar. This allows to show percentages relative to the parent instead of raw numbers of instructions
- select the "Shorten Templates" button if the top bar, this will make templated function names more readable
- make sure functions are listed in "Incl." order in the left pane by clicking on the "Incl." column name. Then get to the top of the list and select the top function (should be "clone" or "start_thread"). This pane shows where the instructions were spent in terms of functions and the selected item is the baseline for the 2 panes on the right. "Incl." means that it is showing the number of instructions spent in the given function and all the ones called from it. The second column "Self" allows to see the instructions spend purely in a given function, excluding the instructions spent in callees
- select the "Callee Map" tab in top right pane. This shows a graphical view of all functions called. Each function is depicted by a rectangle enclosed inside the rectangle of its caller. Each rectangle has its area proportional to the number of instructions spent inside the corresponding function
- select the "Call Graph" tab in the bottom right pane. This shows a graph of all function calls, with edges mentioning the number of calls and rectangles giving the percentage of instructions spent in each function. The color code matches the one of the "Callee Map" pane. Also the thickness of the edges is depending on the number of instructions involved in the call. Note that clicking on one function in this pane enlights it in the top right pane, at least in recent versions of the tools.

- right click in the back of the "Call Graph" pane, select "Graph" then "Min node cost" and finally "5%". This means that the graph will be reduced to functions that use at least 5% of the instructions of their parent. It basically allows to concentrate on the main functions. Just try "1%" to see the difference.

From all this, you should be able to get a rought idea of what the program does and where it is spending time.

Which operator() takes most of the instructions ? -

2 main subparts of it -

Which percentage of instructions is spent in `buildHitsFromRawBank` ? -

5 Optimizing single threaded version

5.1 Global view

Let's now concentrate on optimizing `buildHitsFromRawBank`, called in process. Double click on its box in bottom right pane (or click on it in the left pane) to select it as the base line. The graph and the "Callee Map" are now only showing this function (and its callers in the graph)

Check how many instructions are spent in this function by unselecting "Relative" or the "%" button in the top bar.

Instructions spent in `buildHitsFromRawBank` initially -

Go back to "Relative" mode and also click the "Relative to parent" or "cross" button next to it. Now `buildHitsFromRawBank` is 100% in the graph and the callees percentages are relative to it.

Look at the calls made by `buildHitsFromRawBank` in the graph. Check what they all have in common (but one actually).

Where do we spend instructions in `buildHitsFromRawBank` ? -

5.2 `std::vector::push_back`

Let's first analyze the calls to `std::vector::push_back`, starting with the specific case of `std::vector<>::push_back(LHCb::VPChannelID const&)`

What does `push_back` call internally ? Why ? -

Let's find out the corresponding lines of code. Select the "Source Code" tab in the top right pane. Then click once on the box of the `push_back` function in the bottom right pane. It should send you to the right line of code. If not, it's line 412. Take care to not double click. Clicking once shows where the function is called within the source code of the currently selected function (here `buildHitsFromRawBank`). Clicking twice selects `push_back` as the reference function, and will thus try to display its source code. But as you do not have the sources of the libstdc++, you will get an error message instead.

Good to know

If you're searching where a piece of code is located in a git repo, `git grep` is your friend

```
> git grep channelIDs.emplace_back
Rec/Pr/PrPixel/src/PrPixelHitManager.cpp:410: channelIDs.emplace_back();
```

Now understand the lines of code around (lines 394 to 413 and 368). Find out where `ChannelIDs` is declared and propose an improvement that would avoid the repeated calls to `realloc_insert`. For your education, there are typically up to 100 items in ChannelIDs for a regular event and each items holds no more than 100 ids.

What could improve the `push_back` speed ? -

While you are at it, looking around the line where `ChannelIDs` is declared, note that `xFractions` and `yFractions` may benefit from the same kind of improvements. And `hitsXVec`, `hitsYVec`, `hitsZVec` or `hitsIDVec` as well ! Fix them all.

Try to recompile and rerun valgrind. See the improvements in kcachegrind.

```
# on the remote machine
make -j 20
valgrind --tool=callgrind --dump-instr=yes ./FakeLHCb LHCbsmall.mdf ../geoData.bin
# on your local machine
cd /tmp/username
kcachegrind tcsccourse/exercises/exercise4/build/callgrind.out.yyy
```

Instruction spent in `buildHitsFromRawBank` after fixing -

Gain in percentage -

Let's check how this translates in execution time. We need to recompile in optimized mode.

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make clean
make -j 20
time ./FakeLHCb LHCbbig.mdf ../geoData.bin
```

New duration -

Gain of time in percentage -

5.3 [optional] `std::vector::vector` and `std::vector::~vector`

Note : prefer jumping to the threaded part and come back here afterward in doubt.

Go back to kcachegrind and look at the new graph view for `buildHitsFromRawBank` to see whether you can go further. Note the calls to the copy constructor of `std::vector` (more than 15000 !) and the 30000 calls to the `std::vector` destructor. Try to identify where they are done. If you do not manage, go to the end of the source code, find the calls to `storeOfflineClusters` around line 538 and look around.

Why is copy constructor of vector called there ? -

Analyze the usage of the vector's copies and see whether the copy can be avoided (of course it can !). Do the appropriate fix. Hint : the fix is a single char !

Rebuild and rerun callgrind

```
# on the remote machine
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j 20
valgrind --tool=callgrind --dump-instr=yes ./FakeLHCb LHCbsmall.mdf ../geoData.bin
# on your local machine
cd /tmp/username
kcachegrind tcsccourse/exercises/exercise4/build/callgrind.out.zzz
```

Check new number of instructions after this other fix.

Instruction spent in `buildHitsFromRawBank` now -

Gain in percentage since beginning -

5.4 [optional][advanced]Going even further

This part should only be done if you've finished the rest of the exercise, including the threading part.

Let's do yet another cycle. If we look at the new graph in kcachegrind, not much remains with the 5% cut we've applied. Let's switch to 2% and see what can be improved.

I can see quite some meat :

- we've not tackle `pixel_idx` reservation

- we can reduce the number of calls to reserve easily
- some `push-back` can be replaced by `emplace_back` calls avoiding copying
- change of data structures may be useful when you have `vector<vector<...>>`
- and probably many others. Try to achieve the highest gain you can !

6 Getting thread safe

In this part, we will try to run our reconstruction software in multi-threaded mode. All is ready in the main file "FakeLHCb.cpp", we only have to change the number of threads and use something higher than 1.

Let's first double check how many cores we have on our machine. Just type lscpu and look at the output.

```
lscpu
```

> Number of logical cores on the machine ? -

> Number of logical cores per CPU ? -

Put the number of logical cores per CPU in the code as the number of threads to be used, rebuild and run

```
vim ../FakeLHCb.cpp # change NBTHREADS to 16
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j 20
./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

> Initial behavior of multi-threaded code -

Let's see what helgrind has to say. Helgrind is another tool of the valgrind family able to detect potential race conditions in your code. Note the potential ! It can detect race conditions even when they did not occur. This is done by building a graph of dependencies of all locks and thread unsafe statements and analysing that graph for races.

Usage is as easy as callgrind :

```
valgrind --tool=helgrind ./FakeLHCb LHCbsmall.mdf ../geoData.bin >& helgrind.log
```

The output is be very verbose, hence the redirection to a file. Look at the file produced, starting from the top and read until the first mention of a race condition, something like :

```
==103497== Possible data race during write of size 8 at 0x7FEFFB168 by thread #3
==103497== Locks held: none
... [full stacktrace]
==103497== This conflicts with a previous write of size 8 by thread #2
==103497== Locks held: none
... [full stacktrace]
```

Where is our race condition (file and line) ? -

What is not thread safe ? -

In this particular case, there is no other choice than putting a lock to synchronize accesses to the resource. Use a `std::mutex` and a `std::scoped_lock` to solve the issue. Recompile and run again.

```
make -j 20
./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

Was it sufficient ? Of course not. You may replay the same game and solve the next issue, but we will rather go for a more global and more efficient solution : using constness.

Find out the 2 main methods we are calling in a threaded context. Hint : these are the 2 methods called in `mainLoop` located in the top level file `FakeLHCb.cpp`. Second hint : if you're not familiar with C++, you may miss them as they are operators. Ask a teacher if you do not find them easily.

2 main methods called in threaded code -

Once you found them, make them `const`. Do not forget to change both declaration and implementation. Recompile

```
make
```

You should get compiler errors. For once, they are welcome as they tell you where the race conditions are located.

Start with the ones in `FetchDataFromFile.cpp`. For this file, you have already solved the thread safety problem with a lock. So you are already thread safe, however you get errors as you're changing the status of members in a const method.

Remember that `const` only means "visibly const and race condition free". We have made sure this is the case via the mutex, so we can make use of `mutable` for all members used in the mutexed method. Now you can compile again

```
make
```

Once `FetchDataFromFile.cpp` compiles, look at the other errors, in `PrPixelTracking.cpp`. The compiler should complain that you call a non const method from the `operator()` that is now const. This means that this method should also become const as it is called in a threaded context. Do so and repeat this process of making const the necessary methods until you find the actual race condition, typically a write access to a class member in a const method. If you are not clear why this is a race condition, ask a teacher.

Where is the race condition in PrPixelTracking ? -

The right way to solve this case is clearly not to add a mutex. But let's play the game of learning from our mistakes and do it anyway. So edit PrPixelTracking, and solve the race condition using `std::mutex` and a `std::scoped_lock` as was done for FetchDataFromFile.

Recompile and see that the program runs fine now, without any crash

```
make -j 20
./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

7 analysing thread usage

We will now analyze the efficiency of our multithreaded approach. As we know our number of cores, we have an idea of the ideal speedup we could achieve.

Ideal speedup we aim for -

Now let's measure the actual speedup achieved. Let's switch to Release mode, rebuild and rerun

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j 20
time ./FakeLHCb LHCbbig.mdf ../geoData.bin
```

Time in multi-threaded case -

Actual Speedup -

Clearly not exactly meeting our expectations... In order to understand what is happening, look at htop while running if you did not yet.

`htop` tells you which cores are running with a color code explaining you what was running on each of them. From the documentation, here is the color code meaning :

- Blue: low priority processes
- Green: normal (user) processes
- Red: kernel time (kernel, iowait, irqs...)
- Orange: virt time (steal time + guest time)

So basically you want green. If you have red, this means you are spending your time in the kernel, typically switching context as the running thread on that core could not go further, being stuck on a lock. If you have black, it means the given core is not used

How much green do you see roughly ? -

We are most probably paying our naive locking here. But which one ? Although you may already have guessed, let's use `mutrace` to find out. `mutrace` is able to trace the mutexes of your code and tell you how many times they were locked and for how long. Its usage is trivial and no recompilation is needed

```
mutrace ./FakeLHCb LHCbbig.mdf ../geoData.bin
```

You got a list of mutexes with full stack trace of where they are used and a summary table telling what they costed.

Algorithm concerned	nb locks	time spent(ms)
PrPixelTracking		
FetchDataFromFile		

In case you have troubles reading the stack trace, remember that `c++filt` can demangle the C++ symbols for you, e.g. :

```
c++filt _ZNK15PrPixelTrackingclERKN4LHCb8RawEventE
# -> PrPixelTracking::operator()(LHCb::RawEvent const&) const
```

Which lock is problematic ? -

Now let's remove the problematic lock and solve the original race condition properly. First remove the mutex, the lock and the `mutable` statements you should have added. Then understand the usage of the member that creates the race condition :

Which member is problematic ? -

Where is it used ? -

What's the scope of each usage ? -

Once you understood the scope, you should find an easy way to solve the race condition acting on that scope.

What change solves the race condition ? -

Rebuild a last time, and see how fast it now runs

```
make -j 20
time ./FakeLHCb LHCbbig.mdf ../geoData.bin
```

it's actually a bit too fast to measure the speedup precisely ! Let's put 10 times more data to it and rerun. Also look at htop concurrently from the other shell.

```
for i in `seq 90`; do cat ../LHCbEvents.mdf >> LHCbbig.mdf; done
time ./FakeLHCb LHCbbig.mdf ../geoData.bin
```

New time in multi-threaded case (10x more data) -

New Speedup (take care of the factor 10) -

Still not entirely satisfactory, although much better than before. We won't go further in this exercise but it already illustrates how difficult it is to make old code performant on recent hardware.