optimization of large code base

Sébastien Ponce

July 9, 2019

1 Foreword

This exercise is originally part of the CERN school of computing set of exercises. So you may find useful information there, in particular in this course

We will play here with (a small subset of) the LHCb first level trigger code. The code is reading simulated 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 to 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.

This exercise can in principle be run on any machine running linux. However you need a few dependencies (namely singularity, cvmfs, kcachegrind and ideally mutrace) and, for the second part only, a multicore machine with ideally 8 cores or more to play with multithreading.

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

In order to avoid debugging the setup of your machine, the easiest is to setup singularity and cvmfs. In this part, we detail the steps to set things up if you do not have these tools yet on your machine. You will also install a few other utilities, namely kcachegrind, mutrace and htop in case you do not have them.

3.1 singularity

Installation of singularity is trivial. Depending on your distribution, just run one of

```
sudo apt-get install singularity-container
sudo yum install singularity-container
```

or something equivalent. No configuration is needed.

3.2 cvmfs

Installing cvmfs is described here. Here are the main lines :

- first add the cvmfs repo to your local list of repos
  - on debian derivatives :
    ```
    sudo add-apt-repository 'deb http://cvmrepo.web.cern.ch/cvmrepo/apt stable main'
    sudo apt-get update
    ```
  - on redhat derivatives :
    ```
    sudo yum-config-manager --add-repo http://cvmrepo.web.cern.ch/cvmrepo/yum
    sudo yum-config-manager --enable cvmrepo
    ```
  - for other distributions, do the equivalent
- then install the software with one of
  ```
  sudo apt-get install cvmfs cvmfs-config-default
  sudo yum install cvmfs cvmfs-config-default
  ```
  or something equivalent.
- finally configure it
  - create a base setup with
    ```
    sudo cvmfs_config setup
    ```
  - edit or create `/etc/cvmfs/default.local`
  - put these lines in it
    ```
    CMVFS_REPOSITORIES=sft.cern.ch,cernvm-prod.cern.ch
    CVMFS_HTTP_PROXY="DIRECT"
    ```
  - check it works by running
    ```
    cvmfs_config probe
    ```
  - if it does not, try to restart autofs:
    ```
    sudo service autofs restart.
    ```

3.3 Other tools

Install kcachegrind and mutrace according to your distribution. Typically:

```
sudo apt-get install kcachegrind mutrace htop
sudo yum install kcachegrind mutrace htop
```

3.4 exercise

- open a terminal and go to the directory where you want to work
- start a singularity session

  ```
  singularity exec --bind /cvmfs --home $HOME:/home /cvmfs/cernvm-prod.cern.ch/cvm4 bash
  ```

- in the previous line, we map your home directory to /home (cvmfs image won't let us map it to a better place) and run the bash shell. Adapt the shell if needed.
- clone the exercises' code in your home directory. If you clone it somewhere else, keep it in mind for later.

  ```
  git clone http://cern.ch/sponce/CSC/exercises/exercises.git
  cd exercises
  git checkout LHCbParisWorkshop
  ```

- setup the environment to use a gcc 8.3 as a compiler

  ```
  source /cvmfs/sft.cern.ch/lcg/releases/gcc/8.3.0/x86_64-centos7/setup.sh
  ```

- go to exercise4 and compile the example code using cmake

  ```
  cd exercise4
  mkdir build
  cd build
  cmake -DCMAKE_BUILD_TYPE=Release ..
  make -j 3
  ```

- 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

  ```
  time ./FakeLHCb LHCbbig.mdf ../geoData.bin
  % 21.95s user 0.51s system 99% cpu 22.513 total
  ```

Original duration -

4 General exploration with 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 10
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 humanly readable but you want to use `kcachegrind` for nice visualization of the data.

In order to use kcachegrind, you want to be on the host machine, outside of singularity. So open a new terminal on the host, go to your working directory and type

```
kcachegrind callgrind.out.<xxxx>
```

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.
- enable "Relative" more to the left, still in the top bar. This allows to show percentages relative to the parent instead of raw numbers of instructions
- make sure functions are listed in "Incl." order in the left pane by clicking on the "Incl." column name. Then select the top function (should be "clone" or "start_thread"). This pane shows in 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 algorithm 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`. 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” in the top bar.

Instructions spent in `buildHitsFromRawBank` initially -

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

Look at the calls 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<LHCb::ChannelID...>::push_back`.

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 on the box of the `push_back` function in the bottom right pane. It should send you to the right line of code. Take care to not double click. Clicking once shows where the function is called without 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.

Now understand the lines of code around (lines 409 to 414). Find out where `ChannelIDs` is created 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. Go back to your singularity shell and type :

```
# on the physical machine
make -j 10
valgrind --tool=callgrind --dump-instr=yes ./FakeLHCb LHCbsmall.mdf ../geoData.bin
# on your virtual machine
scp csc-cc-pm-<nn>:~/exercises/exercise4/build/callgrind.out.<yyyy> .
kcachegrind callgrind.out.<yyyy>
```

Instruction spent in `buildHitsFromRawBank` after fixing -

Gain in percentage -

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

Looking at the new graph view for `buildHitsFromRawBank`, let's see whether we 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 line 548 of the source code 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 physical machine
make -j 10
```

```
valgrind --tool=callgrind --dump-instr=yes ./FakeLHCb LHCbsmall.mdf ../geoData.bin
# on your virtual machine
scp csc-cc-pm-<nn>:~/exercises/exercise4/build/callgrind.out.<yyyy> .
kcachegrind callgrind.out.<yyyy>
```

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 to use to something higher than 1.

Let's first check how many cores we have on our machine. Just type lscpu and look at the "CPU(s)" line.

```
lscpu
```

Number of cores on the machine ? -

Put this number in the code, rebuild and run

```
vim ../FakeLHCb.cpp # change NBTHREADS
make -j 10
./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

Initial behavior of multi-threaded code -

We can try to understand what is hapenning with gdb :

```
gdb --args ./FakeLHCb LHCbsmall.mdf ../geoData.bin
run
```

You may get an idea of where the problem lies by checking which line of code the different threads are executing at the moment of the crash. Combined with code inspection, ... good luck !

```
thread apply all backtrace
```

I would personnaly rather try to 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 ! Il 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
```

The output can be very verbose. Depending on your shell buffer, you may even have to redirect its output to a file (use `>& helgrind.log` after the command line). Start from the top and read carefully 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::lock_guard` to solve the issue. Recompile and run again.

```
make -j 10
./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`. 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 the 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` to make the compiler aware and sort out the compiler error. 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 these methods also have to become const as they are called in a threaded context. 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::lock_guard` as was done for FetchDataFromFile.

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

```
make -j 10
./FakeLHCb LHCbsmall.mdf ../geoData.bin
```

7 analysing thread usage

We will now analyze the efficiency of our multithreaded approach. As we've already measured the code in single threaded mode and as we know our number of cores, we have an idea of the ideal speedup we could achieve.

Ideal speedup we aim for -

Before benchmarking, we need to rebuild the whole code in release mode

```
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j 10
```

Now let's measure the actual speedup achieved. From a simple extrapolation, what is the time it would have taken to run on that new input in single thread case ?

Estimated time in single-threaded case -

Let's see how much it takes in multi-threaded mode :

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

Time in multi-threaded case -

Actual Speedup -

In order to understand what is happening, open another shell and launch `htop` in it, with update every .5s (-d 5 does exactly that)

```
htop -d 5
```

Now rerun your measurement in the original shell while watching `htop`

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

`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, as it was stuck on a lock.

How much red do you see roughly ? -

Does it more or less match the missing speedup ? -

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. However, you won't have mutrace in your singularity environment, so use a shell on the host machine ant type :

```
mutrace ./FakeLHCb ../LHCbEvents.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
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 :

- where is it used ?
- what's the scope of each usage ?
- find an easy way to solve the race condition

What change solves the race condition ? -

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

```
make -j 10
time ./FakeLHCb ../LHCbEvents.mdf ../geoData.bin
```

New time in multi-threaded case -

New Speedup -