Optimization of the trigger software of the LHCb experiment

Thesis requirements specification

Péter Kardos

2018-2019

red: instruction from uni
blue: my vague ideas/question about things i don't know much about

1 Background

Here you describe in what context your thesis is to be done.
CERN is a nuclear research organization which runs the world's largest particle accelerator to discover, identify and verify elementary particles and laws of physics. Along the accelerator tube, there are multiple points where high energy particle collisions occur. At one of these points are the devices of the LHCb experiment located, of which the online data-processing software is the subject of this thesis work. Sensors generate a large amount of data about collisions, but only a small fraction of it is interesting and is worth storing for further processing. Interesting events must be filtered for in real-time, thus the performance of the so called trigger code that selects the important events is of critical importance.

- a few words about what data is measured
- what data we reconstruct from it in real-time? (particle paths and...?)
- what events are found interesting?

The filtering happens in several steps. Initially fast dedicated hardware electronics is collecting the data from the detector and building so called raw events. These are then decoded and analyzed in a large farm of standard servers where particles trajectories are reconstructed and analyzed in order to decide whether the "event" is worth recording or not based on caracteristics like momentum and type of tracks. This process will happen at a rate of 30MHz from 2021 on, distributed across 1000 nodes leaving only 30us for each event reconstruction on average.

What prerequisites are valid, what is the goal of the project from the supervisors point of view, what is available and has been done before, under what circumstances should the work be done.
- what do we currently have?
    - single core trigger software working I assume?
    - do we have test data stored offline?
- so we want it at least 3 times faster, but
    - do we benefit from making it 10 times faster?

In order to achieve this ambitious goal, the current software of the LHCb experiment needs to be reviewed to take full benefit of modern processors features, particularly concerning many cores and low level parallelization, like vectorization and superscalar processors.
As the main code is dating from early 2000s, the underlying framework itself (named Gaudi) was not prepared for this. However a lot of effort has been put in modernizing it in the past years, and it is now time to adapt the LHCb code accordingly. The overall goal is to achieve a speedup of a factor 6 compared to the original code, factor 3 compared to what we have today, after the first round of optimizations.
A failure to reach this goal would mean to lower the rate of interesting events reaching offline analysis and thus limiting the physics potential of the overall experiment.

2 Description of the task

what should be done
what tasks/parts does the work consists of
how we go about it

- what hardware are we aiming for?
  - simple multi-core CPU (16 threads, 100 watts)
  - distributed systems (100-800 threads, 20 kW)
  - gpgpu (80k or more threads, 1000 watts)

this is based on how fast we want it and if it's even possible to parallelize it that much
The main task of this thesis work will be to participate in the optimization of the LHCb software, in particular in the areas of the trigger system, level 1 and level 2. Mains directions are :

- run benchmarks to measure efficiency of the current code base
- analysing the current limitations using appropriate tools like calgrind, perf or vtune
- make proposals to improve the code efficiency. This can be rewrite of selected subparts, optimizations of data structures, new algorithms for some processing, etc, etc
- implement a few of these optimizations
- measure improvements achieved
- share the acquired knowledge with the rest of the collaboration so that others can apply same methods on other pieces of code.

[Note, not to be kept] The work will not that much consist in identifying parallelism as we have 30KHz of fully independent events to handle, so we are facing what is called in the litterature embarassing parallelism, that I would call trivial parallelism. The problem is that the code that can run in parallel all these events needs to be thread safe and efficient, which it is not at all. [end of note]
The resulting code has to be able to run on manycore systems (40 to ¿100 cores) running standard Intel or AMD processors and a recent linux distribution.
Developing the application is not a linear, but an iterative process, so the below points might repeat or overlap.

2.1 Getting familiar with code base

Understanding the architecture of the code, learning to navigate among source files, getting familiar with software development environment and compilation process.

2.2 Identifying parallelizable points

Finding parts in the sequential code which can be made parallel. Ideally, the bulk of the computationally intensive code should be running in parallel.

2.3 Sketching a parallel architecture

Determining how algorithms will make use of vectorized instructions and multiple threads. Data layout has to be adjusted accordingly. Additionally, data layout has to be cache-friendly.

2.4 Implementing the parallel system

The sketched system is implemented in code, modifications to the imagined architecture are made where necessary.

2.5 Regression testing

The optimized system has to produce the same results as the previous one.

2.6 Profile performance

The performance of the optimized system is evaluated, and the system is modified to mitigate design and implementation flaws made earlier, thereby increasing performance.

3 Methods

- systems
- tools
- methods
- how is it evaluated

Programming environment:

- C++, latest specification
- Python
- Multi-core systems

Methods:

- AVX2 and/or AVX512 vectorization
- Multi-threading
- Cache and data layout optimization
- vectorization using external libraries like VC or VCL

Evaluation:

- Cachegrind
- Intel Parallel Studio XE
- perf
- Manual timing measurements

The goal is to optimize and make code faster, which requires high performance, low level languages such as C++ and a constant performance evaluation using various profilers.

4 Relevant courses

- Parallel and Distributed Computing
- High Performance Programming

[note to be dropped]LHCb and CERN are providing internally several courses that may be beneficial, in particular around C++, performance optimization and vectorization. We will see how you can benefit from them. One goal of this work can also be to deliver one of these courses at the end.
I also see that you've listed some physics courses. This won't be useful. We stay at the pure computing level. [end of note]
Depending on what I will exactly do, I may use these to understand what I'm coding:

- Mathematical Methods of Physics II (hilbert spaces, metric spaces, manifolds, vector space, lie algebra)(if I pass, haha)
- Applied Mathematics (perturbation theory, diff eq's, integral eq's)
- Computational Physics (numerical integration, differentiation)
- Quantum Physics (schrödinger eq, wave functions and solutions, bra-ket, state collapse, hydrogen atom)

5 Delimitations

- stuff that won't be done
- stuff that is done only if enough time
- stuff that is skipped if too little time
! my stay is way longer than time needed, nothing will be left out probably
! we can say there was no time if we don't want to do it :)

e.g. developing all-new algorithms and formulas is out of scope we will concentrate on software optimizations, so no change to the physics algorithms is foreseen. Also we will not enter the GPU area, this is handled by another subset of the team.

6 Time plan

- largely based on sec 2
- 5 or 8 months of work (30 or 45 cred thesis)
- a task max 1 month long
- graphical plan encouraged (i.e. task graph)
- graph may have conditionals?

- learning the environment and the tools -¿ 2 months
- benchmarking and drawing conclusions on the pieces to improve -¿ 2months

- implementing an improved version of a given piece of code -¿ 3 months
- validating results and fine tuning optimization -¿ 1 months

and we have free space for going further/spending more time/targetting more code.

7 References

I'm gonna come up with something.