Profiling Applications in `C/C++` and `Python`

IPPP Computing Club Seminar

Hitham Hassan

IPPP, Dept. of Physics
Durham University

31 October 2022

Profiling v.s. Benchmarking

Profiling: measuring and analyzing where time is consumed in running your application. Output is typically a table of time spent per “part” of the code.

Benchmarking: observe the behaviour of a system under load, with different input or on different systems. Output is typically one number (total time taken, events per second etc.)

Many tools exist for both, we focus today on profiling code in `Python` and `C/C++` with only a selection of such tools — **alternatives exist**.

Why Profile?

Profiling is **indispensably** useful for diagnosing slow code.

With a breakdown of where runtime is spent in an application, one can identify inefficient code and identify where to **focus performance development** efforts.

Additionally, profiling allows you to identify which parts of the code are **not** used — should these be part of your application?

Similarly to unit tests, profiling can (and should) be used for **instructing development**.

Profiling in C/C++ and Python

Today we focus on **profiling** in both C++ and Python for an application outside of physics.

We will profile code produced to calculate the complement of a nucleotide sequence in genomics and biology.

We will discuss methods for profiling first and then progress to the interactive part of the demonstration.

On the git repository there are implementations in C/C++ and Python — feel free to choose which you prefer to work with.

Profiling `C`/`C`++ Code

Valgrind — Profiling for C/C++ I

Valgrind is a robust profiling and memory debugging tool applicable to many programming languages including C++ and Python — we focus on C++ (for Python , more suitable tools exist).

Compiling C++ with **debug symbols** (specifying the -g flag when compiling) allows Valgrind to be used with your executable.

Valgrind works effectively as a virtual machine, recompiling your code dynamically at runtime when used.

This incurs a significant performance penalty — your code will run **very slowly** though the output of such an analysis can be **invaluable**.

Valgrind — Profiling for C/C++ II

Valgrind can be installed with your standard package manager (for Linux/MacOS) and comes equipped with **tools** which can be specified when running.

For an executable exec, we can use the callgrind tool for profiling (alternative tools exist for memory debugging and detection of memory leaks):

```
valgrind --tool=callgrind ./exec
```

This will run the executable and produce a detailed report named (by default) callgrind.out.x with x the **process ID**.

Many tools have been developed[1] to view the output; I use kcachegrind (available also from your package manager).

[1] Or — for the brave — you can write your own.

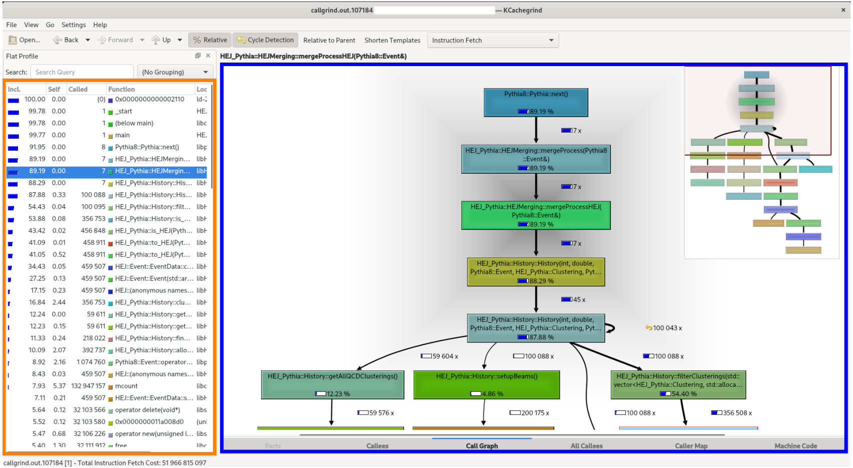
Valgrind — Profiling for C/C++ III

The output displayed in kcachegrind is organised in order of the most computing time used, the methods called are listed in the left sidebar.

The main body of kcachegrind displays where and **how** these methods are called; the **number** of calls and the **percentage** of runtime occupied by all calls to the method. There are a selection of views including:

1. **Callees**: a list of the methods called with the "parent" methods that called them listed.
2. **Call graph**: a graphical representation of the callers and callees showing method and module hierarchy and connections.
3. **Caller map**: a map showing the share of memory map of the applications, most useful for **multithreaded** applications.

kcachegrind Output Example


callgrind.out.107184 — KCachegrind
File View Go Settings Help
Open... Back Forward Up % Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat Profile
Search: Search Query
(No Grouping)
HEJ_Pythia::HEJMerging::mergeProcessHEJ(Pythia8::Event&)
Pythia8::Pythia::next()
89.19 %
7 x
HEJ_Pythia::HEJMerging::mergeProcess(Pythia8::Event&)
89.19 %
7 x
HEJ_Pythia::HEJMerging::mergeProcessHEJ(Pythia8::Event&)
89.19 %
7 x
HEJ_Pythia::History::History(int, double, Pythia8::Event, HEJ_Pythia::Clustering, Pyt...
88.29 %
45 x
HEJ_Pythia::History::History(int, double, Pythia8::Event, HEJ_Pythia::Clustering, Pyt...
87.88 %
100 043 x
59 604 x
100 088 x
100 088 x
HEJ_Pythia::History::getAllQCDClusterings()
12.23 %
HEJ_Pythia::History::setupBeams()
4.86 %
HEJ_Pythia::History::filterClusterings(std::vector<HEJ_Pythia::Clustering, std::alloca...
54.40 %
59 576 x
200 175 x
100 088 x
356 508 x
Parts
Callees
Call Graph
All Callees
Caller Map
Machine Code

Incl.	Self	Called	Function	Loc
100.00	0.00	(0)	0x0000000000002110	ld-2
99.78	0.00	1	_start	HE.
99.78	0.00	1	(below main)	libc
99.77	0.00	1	main	HE.
91.95	0.00	8	Pythia8::Pythia::next()	libp
89.19	0.00	7	HEJ_Pythia::HEJMergin…	libH
89.19	0.00	7	HEJ_Pythia::HEJMergin…	libH
88.29	0.00	7	HEJ_Pythia::History::His…	libH
87.88	0.33	100 088	HEJ_Pythia::History::His…	libH
54.43	0.04	100 095	HEJ_Pythia::History::filt…	libH
53.88	0.08	356 753	HEJ_Pythia::History::is_…	libH
43.42	0.02	456 848	HEJ_Pythia::is_HEJ(Pyth…	libH
41.09	0.01	458 911	HEJ_Pythia::to_HEJ(Pyt…	libH
41.05	0.52	458 911	HEJ_Pythia::to_HEJ(Pyt…	libH
34.43	0.05	459 507	HEJ::Event::EventData::c…	libH
27.25	0.13	459 507	HEJ::Event::Event(std::ar…	libH
17.15	0.23	459 507	HEJ::(anonymous names…	libH
16.84	2.44	356 753	HEJ_Pythia::History::clu…	libH
12.24	0.00	59 611	HEJ_Pythia::History::get…	libH
12.23	0.15	59 611	HEJ_Pythia::History::get…	libH
11.33	0.24	218 022	HEJ_Pythia::History::fin…	libH
10.09	2.07	392 737	HEJ_Pythia::History::allo…	libH
8.92	2.16	1 074 760	Pythia8::Event::operator…	libp
8.43	0.03	459 507	HEJ::(anonymous names…	libH
7.93	5.37	132 947 157	mcount	libc
7.11	0.21	459 507	HEJ::Event::EventData::s…	libH
5.64	0.12	32 103 566	operator delete(void*)	libs
5.52	0.12	32 103 580	0x000000011a008d0	(unl
5.47	0.68	32 106 226	operator new(unsigned l…	libs
5.40	1.30	32 111 912	free	libc

callgrind.out.107184 [1] - Total Instruction Fetch Cost: 51 966 815 097

Figure: An example of output from kcachegrind, run on a small sample of events from HEJ+Pythia (my research project). The orange box highlights the sidebar, the blue box highlights the main output area — currently set to view the call graph.

Valgrind — Some Hints and Tips for Usage

Running Valgrind will **significantly** reduce performance; using without any tools will reduce speed by roughly a factor 4-5.

If your application generally runs for long times, attempt to reduce the total time before using the profiling tool — e.g. use a small event sample if you are profiling a Monte Carlo event generator.

Equally, ensure the runtime is long enough that **valid conclusions** can be drawn from the profiling analysis; running an event generator on just one event is not likely to give representative behaviour.

memcheck is another useful valgrind tool for diagnosing memory management problems (e.g. **leaks**, **allocation** and **freeing** errors/optimisation) though this tool does have limits.

Profiling Python Code

cProfile — Profiling for Python I

cProfile should be available with your Python installation (if not pip is your friend), its usage is remarkably simple.

The library pstats makes the profiling data easier to **visualise**, we will use both of these.

Implementation is simple, execute the methods you want to profile in a with block (shown on next slide), or from the command line:

```
python -m cProfile [-o output_file] (-m module | script.py)
```

Output can be printed to the console in a table format or to a .prof file.

Libraries such as snakeviz may be used to view the output in browser.

cProfile and pstats — Example Implementation

```
import cProfile
import pstats

with cProfile.Profile() as pr:
      some_long_method()

# Produce statistics for profiler.
data = pstats.Stats(pr)

# Sort statistics in order of most time spent.
stats.sort_stats(pstats.SortKey.TIME)

# Sort statistics in order of most time spent.
stats.print_stats()

# Dump the statistics to a file for visualisation.
stats.dump_stats(filename="profiling_data.prof")
```

An Application to Biology(?)

Crash Course — High School Biology I

In biochemistry, there are four organic compounds called *nucleotides* from which DNA is constructed: **A**denine, **C**ytosine, **G**uanine and **T**hymine[2].

Each nucleotide bonds to a specific other nucleotide; A bonds to T, C bonds to G and vice versa.

From these base molecules we can define nucleotide sequences e.g. **ACTGCGAA**.

The *complement* of a sequence is the *unique* sequence that bonds perfectly to it e.g. for the above: **TGACGCTT**.

A *gene* is formed from a nucleotide sequence bonded to its complement and can thus be determined from just one sequence.

[2] For RNA, **U**racil takes the place of **T**hymine

Crash Course — High School Biology II

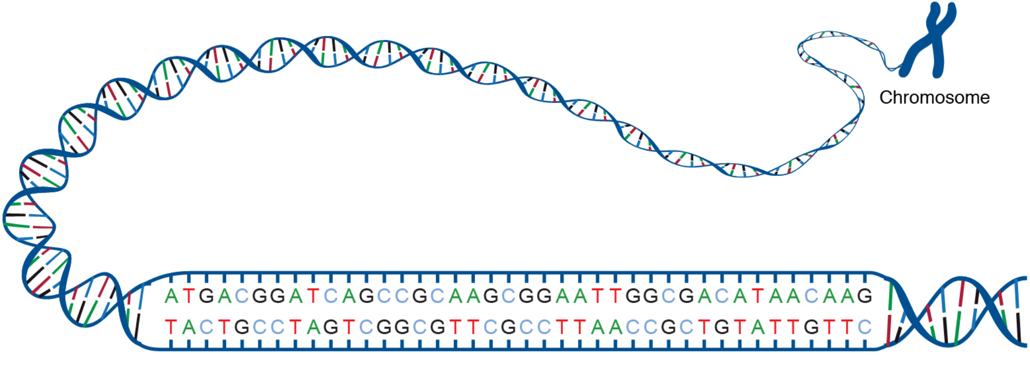

Chromosome
ATGACGGATCAGCCGCAAGCGGAATTGGCGACATAACAAG
TACTGCCTAGTCGGCGTTCGCCTTAACCGCTGTATTGTTC

Figure: A schematic of nucleotide sequences in genes, figure courtesy of: NIH.

The Task

On the repo, you will find an implementation in `C++` and `Python` of a nucleotide sequence complement finder.

In both cases, nucleotide **sequences** are provided as **input** and their **complements** are produced as **output**.

There are two implementations of each, one labelled **original** and the other labelled **optimised**.

Profile the most relevant/interesting implementation for your interests and try to identify where the performance[3] pitfalls are!

[3] The `C++` compiler is better than anyone could be at optimising code, I have had to deactivate this optimisation for the purposes of this exercise...

While You Work — A More Detailed Look at Nucleotide Structure

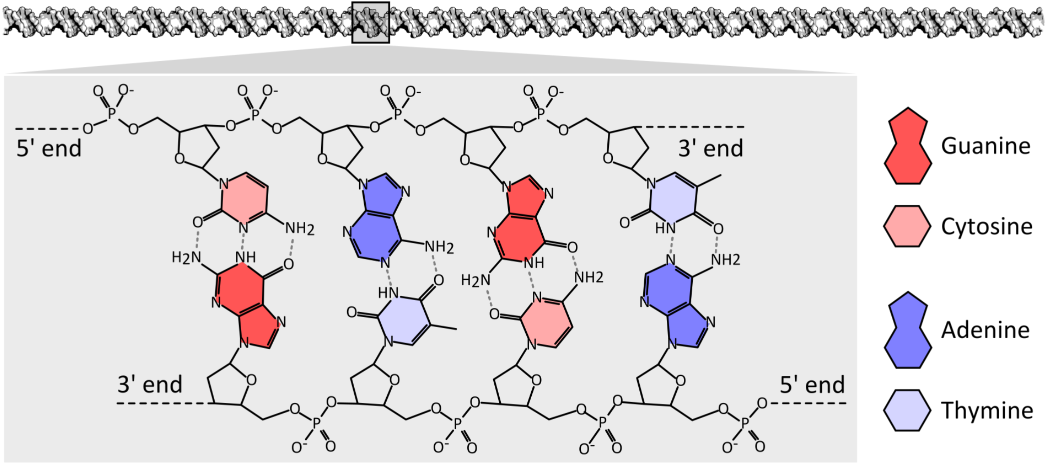


Figure: A more in-depth view of the molecular structure nucleotide sequences in genes for the curious, figure courtesy of: Thomas Shafee.

Following Up from Profiling Analyses I

Results of a profiling analysis will largely be specific to the implementation.

There are many usual culprits when slow performance is observed:

1. **Redundant operations**: do you perform operations multiple times or in different places when they can be performed in the same place?

2. **Many function calls**: function calls are computationally expensive, if you have code that looks like:

   ```
   for (int j=1; j<10; ++j)
     some_function()
   ```

   then this can incur a significant performance penalty since you call `some_function()` 10 times.
   Where reasonable, write functions that use loops rather than loops that call functions.

Following Up from Profiling Analyses II

3. **Unoptimised memory management (`C/C++`)**: Ensure you pass parameters by reference rather than by value wherever you can, produce fewer copies of variables in your code and reduce memory allocation by setting the memory you require.

4. **Unoptimised conditionals**: Using many `if` and `else` conditionals is inefficent, consider the logic of your divisions and whether a more efficient `switch` statement may be used instead.

5. **Unoptimised loops**: Consider if you need to use `while` loops (esp. for `Python`) or can reformulate to a `for` loop. Consider if you can break from a loop early e.g.

   If you are using a loop to find an integer larger than five in a large vector/array, you can exit the loop with `break` once it is found[4]

[4] or just use `std::any_of` in `C++` ... or a similar `Python` tool.

Following Up from Profiling Analyses III

6. **Not using libraries**: For **many** prevalent tasks (e.g. interpolation, root finding, string transformations) built-for-purpose libraries are often available that have been better optimised than something you could write.

   This includes often basic libraries such as the `C++` standard (`std`) library or vanilla `Python`.

   A good rule of thumb is to ask yourself if what you are trying to do feels far more complicated than it should be.

7. **Unoptimised algorithms**: For many "secondary" tasks in our code often the most obvious route is the least efficient e.g. searching using a **linear** rather than a **binary** approach if you have an ordered container (see backup slide).

8. **Using the wrong tool**: If performance really is the bottleneck for your application, should it be written in `Python` to start with?

Our Response - C++ Implementation

src/Main.cpp

Allocating the memory we know we will need and using std::vector::emplace_back rather than std::vector::push_back — it is useful to reserve the memory we need if we know it in advance (which here we do).

src/Gene.cpp

1. Removed unnecessary copies.
2. Optimised the calculation of the complement sequence by removing duplicate/extra loops.
3. Used a switch statement rather than multiple conditionals to transform the sequence string.
4. Used the transform method from the C++ std library — far more optimised than our own ever could be!

Our Response - `Python` Implementation

Implemented entire process in one function.

Made use of native `Python` functionality (e.g. `str.join`)

Removed superfluous loops and unnecessary copies.

Removed unnecessary calculations in loop iterations i.e.

```
for j in range(len(seq)) ...
```

Final Points

Profiling code is indispensably useful, consider how the slight improvements we have seen today would scale for applications running millions of such calculations.

The response to a profiling analysis is mostly specific to the implementation, though many common culprits can contribute.

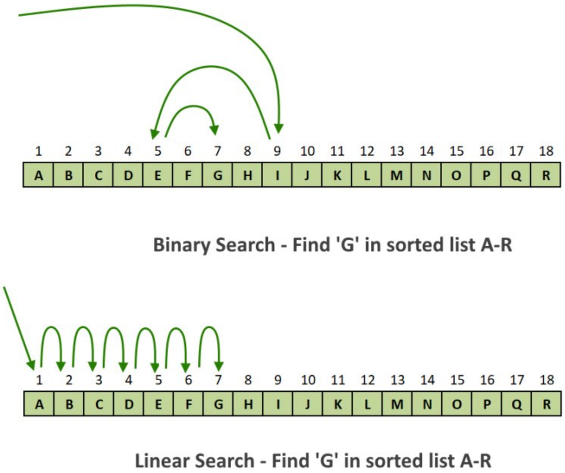
Often optimisation can be as simple as changing the tools one uses e.g. using libraries or changing the framework altogether (`Python` → `C++` for example).

Inefficient data structures can also contribute to slow code, consider packaging your data differently to better allow for efficient algorithms and searches (e.g. ordering if possible).

One can fall into a “benchmarker’s trap”, incomplete profiling on a small scale can give misleading results. Better to use standardised tools.

Thank You

Backup — Linear v.s. Binary Searches



Binary Search - Find 'G' in sorted list A-R

Linear Search - Find 'G' in sorted list A-R

Figure: An illustration of binary v.s. linear searching. One can note immediately that linear searching converges as $\mathcal{O}(n)$ while binary searching scales as $\mathcal{O}(\log n)$ for large n. Figure courtesy of: Tutorial Horizon.

Backup — The BioPython Library

As to be expected, there is a Python library that performs the procedure we implement in (almost) a single line...

```
from Bio.Seq import Seq



seq = Seq("GCTTACGT")
print seq.complement()
```

This would of course be the best implementation in Python as it is purpose-built for such procedures.