Optization and Profiling

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda
alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on October 31, 2019

Optimization and profiling

- Optimization is the process of making your code more performant
- Profiling is the study of the resource cost of your code (CPU, RAM, Hard Disk, network . . .) possibly as function of time / input / execution status.
- Most important message from today's lesson:
  - Premature optimization is the root of all evil
- The correct workflow:
  - Write correct, redible, debugged, tested, documented code
  - Ignore performance at all
  - If the performance are ok: great, you are done!
  - If the performance are not ok: profile to find the bottlenecks.
  - Start the optimization from the problematic lines! Everything else is a waste of time.
  - Keep testing whenever you have a new version!

- Disclaimer: This is a naive description. More on this in the future lessons!

- Data are processed by CPUs (or GPU)
- CPUs have a small memory *cache* (or more of them: L1, L2, ...) of a few KB from which it gets the data
- Generally L1 is on the processor, L2 or L3 are shared among different processors, but this scheme may vary
- The connection between CPU and cache is called *bus backside* and is very fast
- Your data needs to be tranferred from the RAM to the cache, which happens through the *frontside bus*, slower than the backend bus
- Sometimes you need to retrieve data from the hard disk or the network: in that case the time required to retrieve the data is even (much) larger

General optimization goals

- As obvious as it is: do less operations! Use an optimal algorithm for doing your job
- Reduce as much as possible the slow operations involving hard drives and networks
- Make sure that the next data required by the CPU are already in the cache, and need not to be retrieved from the RAM throguh the slower frontside bus
- If possible use *vectorization*, that is let the CPU do multiple operations at once
- If the system has more than one processor, as most computers nowadays, try to parallelize the work on more of them - this will be covered in the next lessons!
- Python is a high level language and generally does not allow a direct control over the memory usage, or the cache - however there are ways to improve the speed of the code, as we will see!

What to avoid

- The CPU uses algorithms of *branch prediction* and *pipelining* in order to try to load the next instructions (and the required data) while processing the current one
- Whenever that fails you will get *branch misses* and/or *cache misses*, plus usually a number of *stalled cycle* on the frontside or backside bus
- Using data structures that keep data contiguosly in memory is better, as sparse data are more difficult to move into the cache into a single transfer
- That's why a list is worse than an array or a numpy array for numerical operations
- *context switches* and *cpu migrations* are managed by the OS, so there is not much you can do about that
- Memory allocations are also expensive, as the program is paused and wait for the OS to find a free memeory location (this, together with I/O operations, is a typical case where a context switch may happen)

Tools that we will use today

- Time profiling:
  - Simple 'print' statements
  - A time measuring decorator
  - The *timeit* Python module
  - The Unix *time* utility
  - *cProfile*
  - *line_profiler*
- Bytecode study:
  - *dis*
- CPU efficiency:
  - *perf*
- Memory profiling:
  - *heapy*

- Fractal set named after the mathematician Gaston Julia
- Take a complex function $f(z)$ and a real number R
- Apply repeatedly f to a complex number z. If the norm of the result is always smaller than R the number belong to the set
- In practice we can only test up to a number of iterations: after that the number will be considered as belonging to the set
- The function we will use is $f(z) = z^2 + c$, where c is the constant complex number $c = -0.62772 - 0.42193i$

Julia output