Profiling

Problematic

"My program is too slow, what can I do ?"

Parallelize it ?

- No ! (or "Not now ! later !")

First, understand where you "lose" most of the time
This process is call "profiling"

How to speed up my program ?

- find "hotspots"

“90 percents of the time is spent in 10 percents of the code.”

- Do "high level optimization" : (re)-design for speed only the "hotspots"

- low level optimizations :

assembly, SIMD, optimize cache memory

- Parallelize it

Finding hotspots

Remember the "KISS" Principe : "Keep it simple, (and) stupid"

Optimizing something seems to go against the KISS Principe...

Donald Knuth says :
"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times ; premature optimization is the root of all evil (or at least most of it) in programming."

So, keep the code simple, then optimize only what you need.

(and remember also : "90% of the time is spent in 10% of the code". And 10% is sometimes overestimated...)

Language choice

Some programming language are thought to be "fast" (e.g. C) and other to be "simple" (e.g. python)

Theoretically, the fastest language is the assembly language, but :

- very complicated
- not portable at all
- compiler optimise well the code, and can reach very good performances
- in "speed designed" languages, it's possible to add ASM part

Compiler are more and more smart, and can optimize easy things for you (-O options in gcc/g++/clang)
It can :

- move variables into registers,
- inlinize
- propagate constants
- remove useless instruction (for example, variable never read)
- unloop
- derecursive functions
- ...

floats : –ffast-math : break standard (rounding standard for example), but usually not important for you.

-march=active : use the instruction set of your computer, not the instruction set compatible with all the processors of the family.

Finding hotspots

Easy way to do this : use a profiler.

Usually, there is a profiler with the compiler suite you use

For example, with GCC/G++, there is "gprof"

There are more "low level" general profilers. We will also see Intel's "Vtune"

Where the time can be lost ?

(1) Are you sure it is a "CPU " problem ?

Example : A "syscall" (= "system call") can be slow, or can "block"

File operations, network operations, mutex, some memory allocations are "syscall"

Solution :

- redesign with less system operation...

example, put ("cache") things into memory before writing them on the disk, or sending them

Or this can be a timing / synchronisation issue in the code

For example : you should never use "sleep" operations to wait something...

Where the time can be lost ?

(2) You are sure it's a "CPU" problem, i.e. there are "Too many" CPU instructions

- find a better algorithms, and optimize them

Of course, this depends a lot of what you are doing
But there are some classic stuff :

- precompute things if possible
- lazy computing
- memorize (dynamic algorithms..)
- use less memory, try even to work with "bits"

"CPU" problem...

Note that a CPU can loose time in "non visible" things

- a function "call" can takes more since some registers has to be saved
- branches that cannot be guessed loose the pipeline
- memory access not in the cache

gprof usage :

compile with -pg

run it (and wait it ends)

launch gprof <program>

(example IRL)

problems with gprof

- -pg with -O do strange things
- its very roughly (but usually very useful at the beginning)

Once you find the "hotspots", redesign the part of the code

It its already the best you can do (in algorithmic point of view), then go to low level optimizations

Low level optimizations

Idea : "Look at the assembly code" of hotspots

You can use a more precise profiler

- For example "Vtune" by Intel for x86, using hardware things.
- Or things by "valgrind"

(example IRL)

Understand where the CPU "loose" time

"Caches miss" :
gprof does not report this, but one can use valgrind
command : valgrind –tool=cachegrind <prog>

Solutions :

- reorganize the memory or the order of access
- use less memory (ex : bytes instead of 64bit int, bits)
- if you have 2D arrays (matrix), sometime "transpose" is better
- if you have booleans, consider to use "bit" operations

Other problem : "Unpredictable things"

- avoid unpredictable branches

Optimize in "assembly"

Usually, compiler do goods things in standard assembly.

But usually, it cannot profit of SIMD instructions (SSE,AVX...) nor GPU

Some libraries are already optimized for SIMD (and/or GPU) (especially "maths" stuffs : FFT, matrix calculus, image and video processing, neural network)

Also you can program directly with SIMD or GPU

- Look at "intel intrinsic" for SIMD on x86
- or (e.g.) Cuda or SYCL or Vulkan for GPU programming

Parallelization

In fact, may be complicated for only a "small" benefit, if you do not envisage to use a computing cluster
(usually 2 or 4 "real" cores in most of laptop computers)

Some languages have primitives for it.
In C/C++, you can either use "threads" (low level) or more high level things : OpenMP or MPI.

But you will have a new problem to manage : synchronisation between threads.
Bad synchronization can :

- more "complicated" bugs to track
- you can lose more time than you win

It's why most researcher use solutions like MPI or OpenMP.

- The parallelization is more/very interesting if you can/want to use big computers/clusters

Computing clusters

It's common to use computing clusters in research
But, not very often used to prove "math" theorems
You are the next generation, change this !

At the LIP : $\sim$ 10 computers with at least 32 cores
One machine with 96 cores and 1.5 TB of ram

At the ENS : the "CBPSMN"
$\sim$ 6000 cores
(used mostly by physicists, biologists and chemisists)

National : `https://www.edari.fr/` , Grid5000 ....