Accelerating CORSIKA8 using massively parallel platforms

Overview, requirements, design patterns and recommendations

A. Augusto Alves Jr

Presented at CORSIKA-devel meeting - KIT, Karlshuhe

December 3, 2020

Outline

- Basic concepts
- Programming environments
- Design patterns

Why to deploy parallelism and concurrency?

Basically, there are three walls that can't be overcame

- Power: it is not possible anymore to increase clock frequency without unreasonably increasing power consumption and design complexity. Think about cooling management, for example.
- Memory: memory chips became very cheap overtime. Albeit that, accessing memory is becoming increasingly costly (multi-layered caches L1, L2, slow bus...).
- ILP: in theory, instruction level is already deployed. Actually, recently some processor speculation has been disabled due breaches in security (Meltdown & Spectre).

The current roadmap to increase overall efficiency is to deploy concurrency and parallelism: multiprocess, multithread and vectorization.

Concurrency and parallelism

Concurrency is ability to execute different parts of a program or an algorithm in out-of-order or in partial order, without affecting the final outcome. Parallelism means the execution of concurrent tasks simultaneously.

- Concurrent routines can be executed in parallel.
- Design of concurrent programs and algorithms is challenging and requires reliable techniques for coordinating instruction execution, data exchange, memory allocation and execution scheduling to minimize response time and maximise throughput.
- Potential problems: race conditions, deadlocks, resource starvation etc.

Thread and thread-safety

In Linux a (CPU) thread is a lightweight process: it has a private stack but shared heap, data segments and code. A piece of code (design) is said "thread safe" when can be executed or accessed by multiple threads, without triggering disasters.

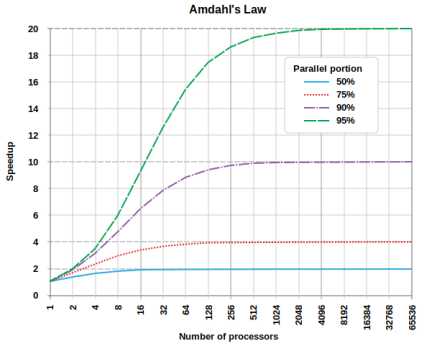
- Nothing is thread safe unless explicitly stated.
- STL containers (c++17 and earlier standards) are not thread safe.
- TBB containers are mostly thread-safe.
- Many system calls are not.
- Most of the libraries available are not.

Amdahl's Law

Describes the theoretical speedup in the execution a task with fixed problem size when more resources (threads) are thrown on it:

$$F(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Where F is the speedup factor, s is the number of threads and p is the parallelizable portion. Note that regardless of the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.



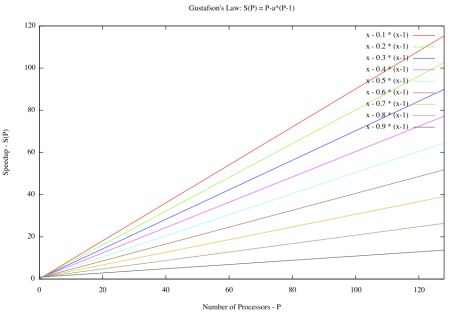
Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIP Conference Proceedings, vol. 30 (Atlantic City, N.I., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

Gustafson's Law

Describes the theoretical speedup in the execution a task with fixed execution time when more resources (threads) are thrown on it:

$$F(s) = N + (1 - N)s$$

Where F is the speedup, N is the number of threads (instead of just one) and s is the serial part of the program. Gustafson's law deals with programs designed to work with larger problems sizes in order to fully exploit the resources as they become available. Therefore, if more threads are available, larger problems can be solved within the same time.



John L. Gustafson. 1988. Reevaluating Amdahl's law. Commun. ACM 31, 5 (May 1988), 532–533. DOI:https://doi.org/10.1145/42411.42415

Strong vs. Weak scaling

Strong scaling:

- A problem with fixed size that is processed by an increasing number of threads.
- Better modeled by Amdahl's Law.

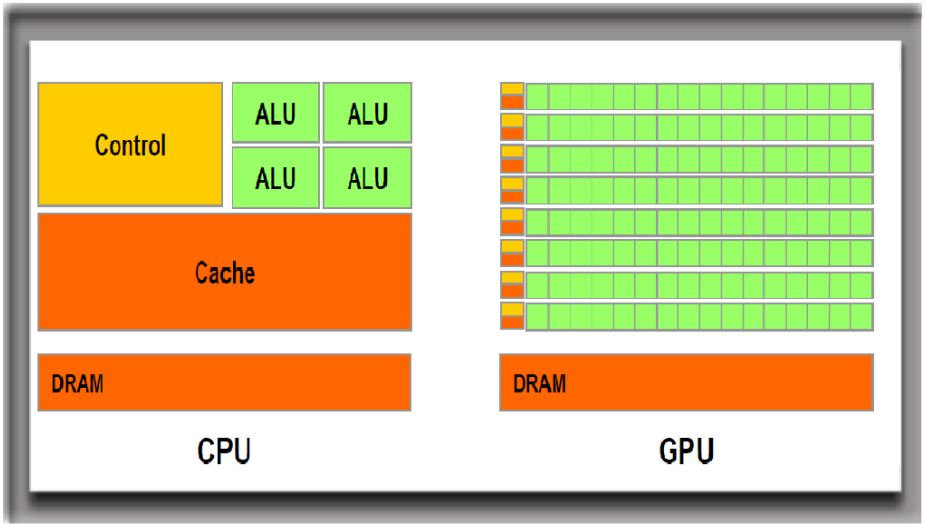
Weak scaling:

- Each thread available contributes with some work. Adding threads also adds work.
- Solving problems with increasingly problem size, at same time.
- Better modeled by Gustafson's Law.

CPUs and GPUs

- The CPU carries out all the arithmetic and computing functions of a computer. Principal components of a CPU: some few arithmetic logic unit (ALU), some registers and a huge control unit. Threads are expensive to create and destroy.
- The GPU (graphics processing unit) is specialized processor designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer. Principal components of a CPU: many arithmetic logic units (ALU), many registers and some smaller control units, managing groups of cores. Threads are cheap to create and destroy.

CPUs and GPUs



Tools and programming environments

There is a quite comprehensive set of computing models and programming environments for parallel computing in different platforms:

- Library level: TBB, PThreads, STL (multithread facility, since C++11).
- Language level: OpenMP, OpenACC.
- Environments/computing model: CUDA, OpenCL.

Most of these alternatives are multiplatform, but not all. There are substantial differences in scope as well.

Designing parallel and thread safe code

Key questions to ask before writing code:

- How big is the parallelizable portion of the problem?
- Is it more consistent with strong or weak scaling?
- Is the task divisible into subproblems with different parallelizable portions (type and size)?
- What are the platforms available? Do they match the necessary resources ?
- Are the subproblems expressible in parallelizable design patterns?
- Is the surround code thread safe?
- How to migrate code from one platform to other? Example: from CPU to GPUs?

Designing software to run in parallel is fully different than adapting existing serial code to run in parallel.

Designing parallel and thread safe code

There are many STL algorithms that are largely parallelizable. To cite some not very known or used:

- reductions : mapping a collection of objects into a single object.
- transforms : mapping a collection of objects into another collection of objects.
- for_each : applying a predicate to each object of a collection.
- scan : scanning a collection performing a partial reduction.
- partition : reorder a collection according to a predicate.

Parallel implementations for such algorithms are available in Thrust, TBB, CUB, STD/STL and so on.

Once a design patterns is chosen to solve a certain problem, lets say "visitor", the operations performed at patterns level and sublevel should be implemented using generic parallelizable algorithms.

Example: `https://github.com/MultithreadCorner/Hydra`

Designing parallel and thread safe code

Things to do:

- Deploy design patterns.
- Reflect carefully about data handling and containers.
- Deploy policy based design for management of parallelism.
- Use the stl algorithms.

Things to avoid doing:

- Writing code using open parallel directives (OpenMP or OpenACC).
- Avoid writing your own kernels in CUDA and OpenCL. Difficult to maintain, requires skilled people to develop and just prone to problems of all sorts.