# Parallel Computing

Fabrice Roy[1]    Vincent Lafage[2]

[1]Observatoire de Paris, Site de Meudon
Université Paris Sciences & Lettres
[2]IJCLab, Laboratoire de Physique des 2 Infinis Irène Joliot-Curie
Université Paris-Saclay

9 December 2021

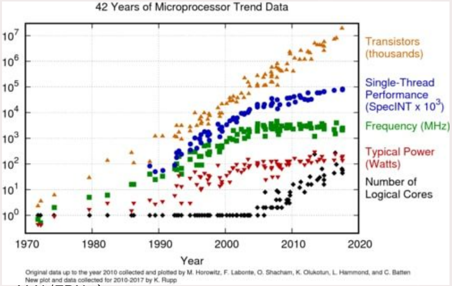
## Any prior experience?

- parallel programming? C/POSIX threads? OpenMP?
- Fortran? Jurassic Fortran? Archeo Fortran? Modern Fortran?
- C++? '98/'03? '11+?

# Why parallelize?

## end of Moore's law

- MOORE's Law
  Gordon MOORE's observation (1965):
  *The number of transistors*
  *in a dense integrated circuit (IC)*
  *doubles about every two years.*
  (even before microprocessors)

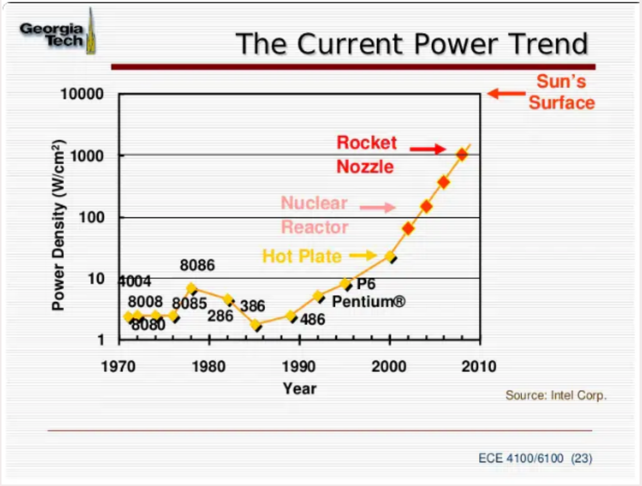


+ registers
+ memory cache
+ processor instructions
+ bus size (4 bits → 64 bits)
+ memory management (MMU)
+ processing units (one, then many ALU/FPU, vector ALU/FPU...)
+ pipeline depth (superscalars *cf* Pentium ca 1993)
+ complex branch predictor / out-of-order execution unit

- **Heat/Power Wall**: $\mathcal{P} = \alpha \cdot C \cdot V_{dd}^{2} \cdot f + V_{dd} \cdot I_{st} + V_{dd} \cdot I_{leak}$
- **Frequency Wall**: « Free lunch is over » (already for 15 years)
- 1971 ⇒ 10 µm, 2012 ⇒ 22 nm, 2014 ⇒ 14 nm, 10 nm in (slow) progress (Intel). TSMC, Samsung: 7 nm and 5 nm factories (*but maybe not exactly the same measurement*). 3 nm using GAAFET under development. Tunnel effect ⇒ **Quantum Wall**
- **Money Wall**

# Why parallelize?

## Frequency/Power Wall

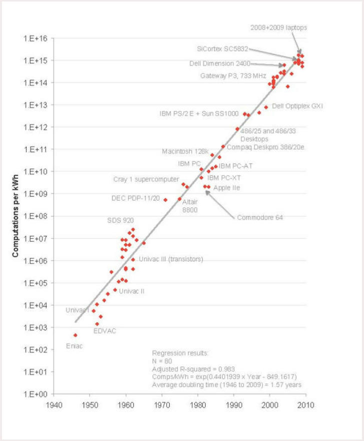


# Why parallelize?

## in the era of climate change

Information technologies : growing part of a rare, expensive & dirty energy.
1.6 MW for the first room of IN2P3 Computing Centre: 0,5 to 1 M€/yr
Moving from PFlops to Exascale requires a breakthrough...

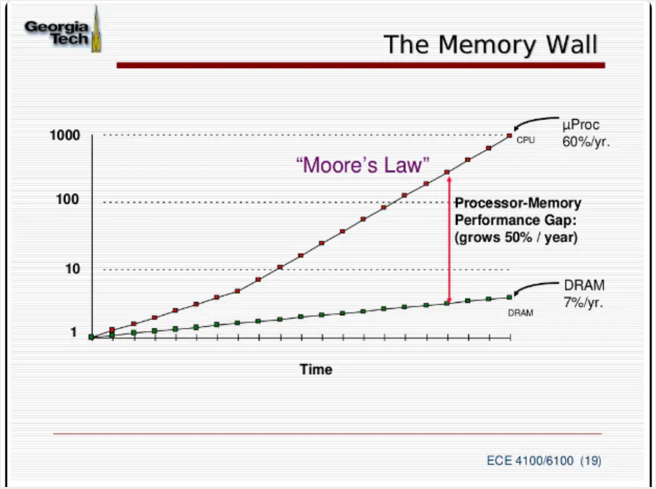
- moving to a **better W/MIPS ratio** (or W/MFLOPS):
  Intel XScale[1], 600 MHz, 0.5 W
  *5 × slower, 80 × cheaper in energy!*
- **reduce frequency**, using more cores



[1] ARM ancestor

## Why parallelize?

### Yet another Wall...



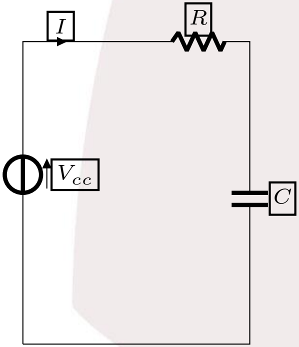
Georgia Tech

The Memory Wall

ECE 4100/6100 (19)

# Why parallelize?

## Memory Wall



Data is moved through wires

Wires/memory behave like an RC circuit

Trade-off:

- Longer response time $\tau = RC$ (“latency”)
- Higher current $I$ ($\Rightarrow$ more power)

Physics says:

*Communication is slow, power-hungry, or both*

Hierarchy of memories

- Small amount of fast memory close to CPU
- Large amount of slow memory far from CPU

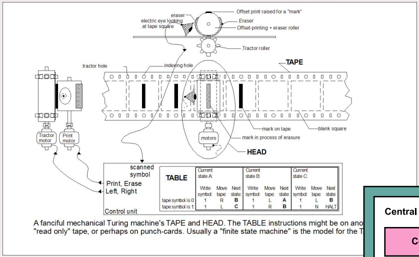
CPU register « Level 1 cache « Level 2 cache « Level 3 cache « Main memory « Disk « Internet

## Why parallelize?
### Memory Wall

We must feed the CPU $\Rightarrow$ some problems will be **memory bound**.
The distinction between **memory bound** and **CPU bound** algorithms can often be related to their **arithmetic intensity**:
for $N$-sized problem, how many operations?

- dotproducts: $\mathcal{O}(N)$ data, $\mathcal{O}(N)$ ops
  convolution
- matrix-vector products: $\mathcal{O}(N(N+1))$ data, $\mathcal{O}(N^2)$ ops
- matrix-matrix products: $\mathcal{O}(2N^2)$ data, $\mathcal{O}(N^3)$ ops
  matrix inversion, diagonalisation, Fourier/Bessel transform...

# Architecture

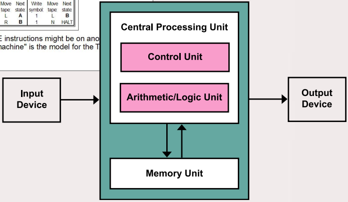


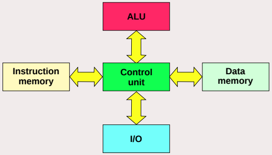
| | Current state A: | | | Current state B: | | | Current state C: | | |
|---|---|---|---|---|---|---|---|---|---|
| | Write symbol: | Move tape: | Next state: | Write symbol: | Move tape: | Next state: | Write symbol: | Move tape: | Next state: |
| tape symbol is 0: | 1 | R | **B** | 1 | L | **A** | 1 | L | **B** |
| tape symbol is 1: | 1 | L | **C** | 1 | R | **B** | 1 | N | HALT |

A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on ano
"read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the T

- Turing Machine
- von Neumann architecture (Princeton architecture)

  ⇒ von Neumann bottleneck
- Harvard architecture

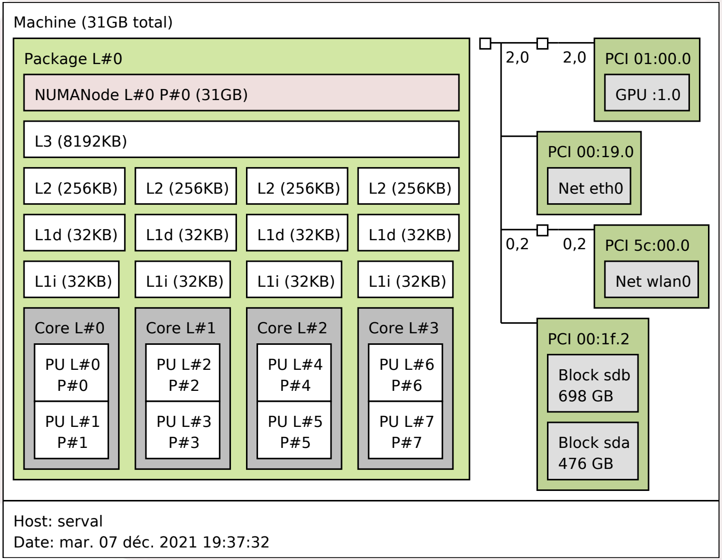




## Know your tool

| | |
|---|---|
| execute typical instruction | 1 ns |
| fetch from L1 cache memory | 0.5 ns |
| branch misprediction | 5 ns |
| fetch from L2 cache memory | 7 ns |
| Mutex lock/unlock | 25 ns |
| fetch from main memory | 100 ns |
| send 2K bytes over 1Gbps network | 20 000 ns |
| read 1MB sequentially from memory | 250 000 ns |
| fetch from new disk location (seek) | 8 000 000 ns |
| read 1MB sequentially from disk | 20 000 000 ns |
| send packet US to Europe and back | 150 000 000 ns |

# Know your tool `hwloc-ls`



## Sequentiality

**IMPERATIVE PROGRAMMING** = programming sequence of instructions/subtasks to the processor

*program as an ordered shopping list, as an ordered recipe*

**SEQUENTIALITY** is essential to programming

## Concurrency

With only one processor, **tasks** will get executed one after the other.
Often this order is compulsory: permuting tasks would change the result
... sometimes this order is contingent: permuting tasks wouldn't change the result

If we can identify all these permutable tasks,

- we could run those **OUT OF SEQUENCE**
- we could run those **CONCURRENTLY** on multiple processors, or execution units

(exhibiting concurrency in a program is an industrialization process).

# Task & Thread

Logical level: we want to identify **TASKS** and among them, order-independent tasks.
Physical level: we want to assign tasks to execution **THREADS**.
Multitasking can occur on one processor:

- **time sharing** of processing ressource among threads
- **context switching** between threads

If we have a multiprocessor, some/each processor can be assigned one or many threads

| | | |
|---|---|---|
| **PARALLEL** programming (a.k.a multiprocessing) | = | **CONCURRENT** programming on a **MULTIPROCESSOR** (a.k.a. multiprogramming) |

two kinds of loops:

— iterations depends on the previous one(s)

what we usually call an iterative process

— iterations are independent of the previous ones

more duplication (or N-uplication) than iteration

$\Rightarrow$ embarassingly parallel = lowest possible concurrency = as decoupled as possible
$\Rightarrow$ delightfuly parallel!
very common in particle physics: each event is independent and can be processed on a separate processor / in a separate process

$\Rightarrow$ **DISTRIBUTED** processing

## Purity

When we apply the same function on a collection of objects, the collection of result is independent of the order of application of the function.
To ensure that this is right we need **PURE** functions:
$\Rightarrow$ computer functions that are as close as possible to mathematical functions

- the function return values are identical for identical arguments
  (no variation with local static variables, non-local variables, mutable reference arguments or input streams.) i.e. its evaluation relies on a **DETERMINISTIC ALGORITHM**: given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states
  - $\Rightarrow$ function are *referentially transparent* (see below)
- the function application has no **SIDE EFFECTS**: no mutation of local static variables, non-local variables, mutable reference arguments or input/output streams

## Purity

- input arguments must be *immutable*: `C++` `const`, `Fortran` `intent (in)`,...
- evaluation must not rely on (mutable) global variables
  (e.g. in `Fortran`, it shouldn't rely on `COMMON` variables, but it can rely on module `parameters` or `protected` variables.
  In `C++`, you can use `const / constexpr` global)
- a pure function can only call pure functions

**REFERENTIAL TRANSPARENCY**:
$\Rightarrow$ the expression can be replaced with its corresponding value (and vice-versa) without changing the program's behavior.
$\Rightarrow$ allows **MEMOIZATION**:
optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result
a specific type of **LOOKUP TABLE (LUT)**:
$\Rightarrow$ a collection / an array of precomputed results that one reuses instead of recomputing.

Lookup tables are usually initialised at start, while memoization fills it on the fly.

## Side effects

### what happens when the function is not pure...

- **Input/Output**: displaying something occur in a given order, storing data to disk (can be seen as a global object)
- **hardware related behavior**: depends on the interaction with environment, which is a global variable
- **time dependency**: time is a global variable
- **exceptions**: your function is not returning a value of the expected type, likely because of limited definition domain for the arguments.
  A mathematical function is not only pure, it also aims at *totality* (maximal expansion of the definition domain)
- most random number generators rely on a hidden state changing on each call.

$\Rightarrow$ in the long run, no computer function can ever be called pure: running a computer requires energy and increases the entropy of the Universe, which is a side effect...

## short warning and advertisement for tomorrow

**CAVEAT !!!**
Floating point evaluation are usually dependent on the order of evaluation: floating point operations are NOT associative, contrarily to the real number corresponding operation: $(a+b)+c \neq a+(b+c)$
Subtle side-effects introduced by the languages, compilers and optimization options...

- `C` strictly conforms to your order of computation
- `Fortran`, *i.e.* FORmula TRANslator, tries to somehow optimize your computation: mathematically equivalent, numerically not strictly equivalent

Some purity check by compiler are rather formal.

## What could go wrong?

**REENTRANCY**

a subroutine is called reentrant if

- multiple invocations can safely run concurrently on multiple processors,
- or on a single processor system, where a reentrant procedure can be interrupted in the middle of its execution and then safely be called again ("re-entered") before its previous invocations complete execution.

- Reentrant code may not hold any static or global non-constant data without synchronization.
- Reentrant code may not modify itself without synchronization.
- Reentrant code may not call non-reentrant computer programs or routines.

**THREAD SAFETY**

Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction.

$$\text{reentrant} \nRightarrow \text{thread-safe}$$
$$\text{thread-safe} \nRightarrow \text{reentrant}$$

`https://en.wikipedia.org/wiki/Reentrancy_(computing)`

`https://stackoverflow.com/questions/856823/threadsafe-vs-re-entrant`

## When some task takes the lead

**CRITICAL SECTION** is a part of code where concurrent accesses to shared resources would lead to erroneous behavior.
$\Rightarrow$ we need to protect these accesses
**Lock** / **mutex** (mutual exclusion), protected object
(atomic instruction)

During a critical section, we loose all benefits of the multiprocessor.

!!!Warning!!!: dead lock
synchronization point, or rendez-vous:

sometimes one tasks has to wait for the completion of another one

## OpenMP

http://icps.u-strasbg.fr/~bastoul/teaching/openmp/bastoul_cours_openmp.pdf
http://www.idris.fr/formations/openmp/
http://www.idris.fr/media/formations/openmp/idris_openmp_cours-v2.11.pdf
http://www.idris.fr/media/eng/formations/openmp/idris_openmp_cours-eng-v2.11.pdf

http://www.idris.fr/media/formations/openmp/idris_openmp_tp-v2.9.pdf
http://www.idris.fr/media/eng/formations/openmp/idris_openmp_tp-eng-v2.9.pdf
http://www.idris.fr/media/formations/openmp/openmp_tp-v2.9.tar.gz

https://gitlab.in2p3.fr/lafage/phynubeparallel

```
git clone https://gitlab.in2p3.fr/lafage/phynubeparallel.git
```