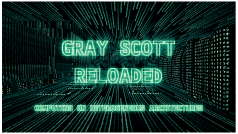
Programmation non-euclidienne

comment avoir les threads qui se touchent… avant l'infini ?

Vincent Lafage

[1]IJCLab, Laboratoire de Physique des 2 Infinis Irène Joliot-Curie
Université Paris-Saclay



July 1st 2024

Why parallelize ? end of Moore's law

- MOORE's Law : Gordon MOORE's observation (1965)
  « Cramming More Components onto Integrated Circuits. » :
  The number of transistors
  in a dense integrated circuit (IC)
  doubles about every two years.
  (even before microprocessors)

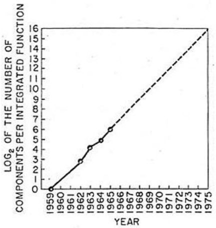


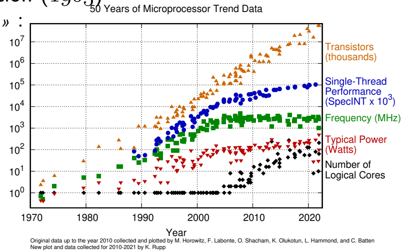
Fig. 2 Number of components per Integrated function for minimum cost per component extrapolated vs time.

+ 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, almost 20 years)
- 1971 ⇒ 10 µm, 2012 ⇒ 22 nm, 2014 ⇒ 14 nm, 10 nm in (slow) progress (Intel). TSMC, Samsung : 7 nm, 5 nm factories. 3 nm and beyond down to 1.4 nm in Intel roadmap. Tunnel effect ⇒ **Quantum Wall**
- **Money Wall**

Why parallelize ? end of Moore's law

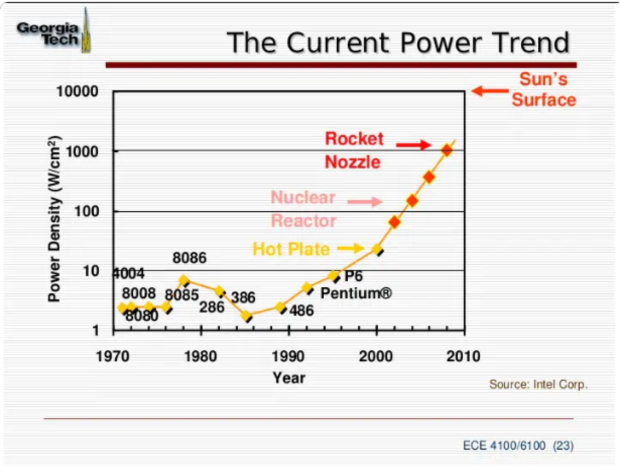
- MOORE's Law : Gordon MOORE's observation (1965) :
  « Cramming More Components onto Integrated Circuits. » :
  The number of transistors
  in a dense integrated circuit (IC)
  doubles about every two years.
  (even before microprocessors)



+ 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, almost 20 years)
- 1971 ⇒ 10 µm, 2012 ⇒ 22 nm, 2014 ⇒ 14 nm, 10 nm in (slow) progress (Intel). TSMC, Samsung : 7 nm, 5 nm factories. 3 nm and beyond down to 1.4 nm in Intel roadmap. Tunnel effect ⇒ **Quantum Wall**
- **Money Wall**

Why parallelize ? Frequency/Power Wall



Introduction to Multicore architecture, Tao Zhang – Oct. 21, 2010

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

« The number of computations per joule of energy dissipated doubled about every 1.57 years. »

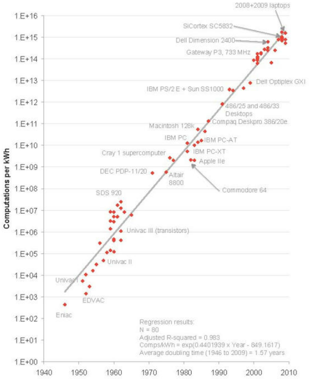
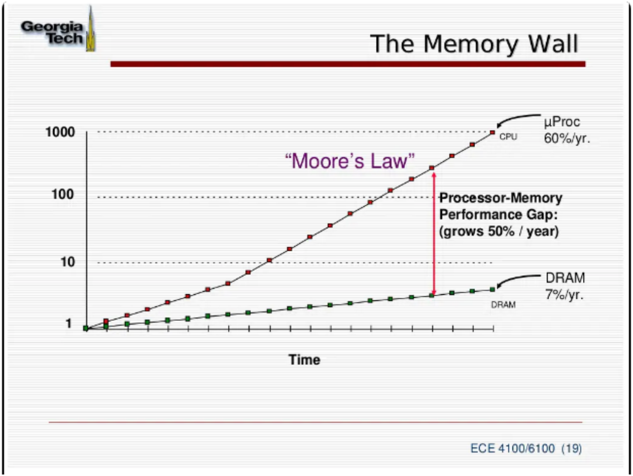
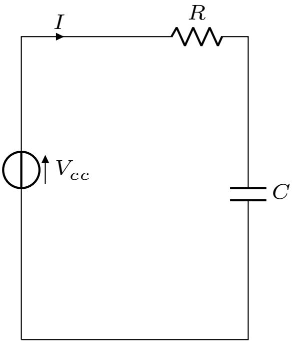


Figure – Koomey's law, 2010

Why parallelize ? Yet another Wall...



Introduction to Multicore architecture, Tao ZHANG – Oct. 21, 2010



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

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...

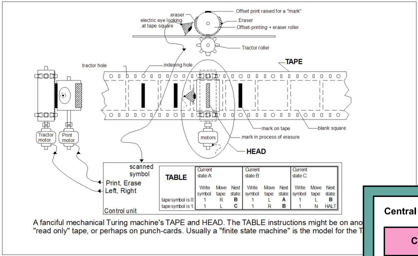
"If the only tool you have is a hammer, you tend to see every problem as a nail."

MASLOW*'s gavel*

$\Rightarrow$
"If the only tool you have is a GPU, you tend to see every problem as a matrix product."

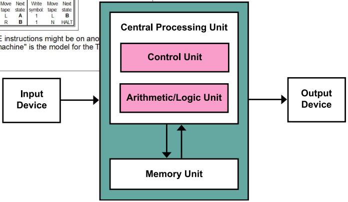
Architecture

- TURING Machine

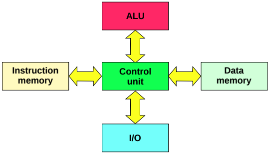


- VON NEUMANN architecture (Princeton architecture)

  ⇒ VON NEUMANN bottleneck



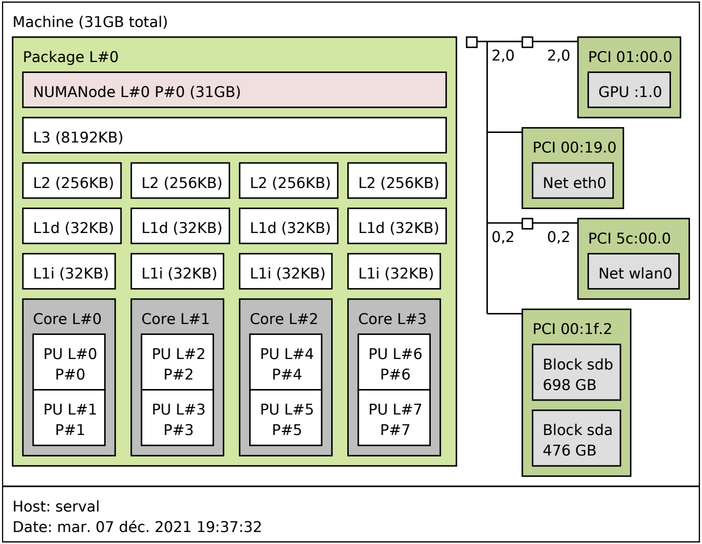
- Harvard architecture



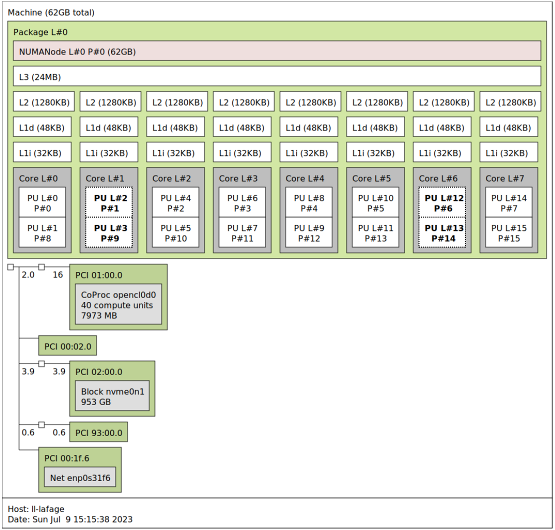
Know your tool

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



Know your tool `hwloc-ls`



Why parallelize ?

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)
$\Rightarrow$ what we usually call an iterative process

— iterations are independent of the previous ones
$\Rightarrow$ 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 :
⇒ 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
  - ⇒ 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

Pure = deterministic + without side effects

- 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 / static constexpr` global)
- a pure function can only call pure functions
- no exceptions

REFERENTIAL TRANSPARENCY :
⇒ the expression can be replaced with its corresponding value (and vice-versa) without changing the program's behavior.
⇒ 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)** :
⇒ 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.

Mixing functional paradigm (purity) with object-oriented paradigm will **strongly** change your object-oriented style

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.

⇒ 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 global variable ⇒ side effect...

`Fortran'23`

A pure procedure changes variables outside its scope only through its arguments. This allows it to be used in parallel constructs, where concurrency issues would otherwise prevent use.
A simple procedure is a pure procedure that in addition is restricted to reference variables outside its scope only through its arguments. ⇒ It represents an entirely local calculation.
If all the intent in arguments are constants and there are no intent inout arguments, it may be performed by the compiler at compile time.
A simple procedure must satisfy all the requirements of a pure procedure. In addition,

- it must not reference a variable by use or host association,
- it must not reference a variable in a common block,
- all its dummy procedures must be simple,
- all its internal procedures must be simple,
- all procedures it references must be simple,
- when used in a context that requires it to be simple, its interface must be explicit and specify that it is simple, and
- it must not contain a entry statement.

- All the intrinsic functions are simple.
- All the module functions in all of the intrinsic modules are simple.

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 : $\forall(a, b, c), (a + b) + c = a + (b + c)$ **BUT** $\exists(a, b, c), (a \oplus b) \oplus c \neq a \oplus (b \oplus c)$
$\Rightarrow$ out-of-order operation might change ever so slightly the result
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 (particularly on heterogeneous architectures)...

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. (*no data race*)

reentrant $\not\Rightarrow$ **thread-safe**
thread-safe $\not\Rightarrow$ **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 task has to wait for the completion of another one