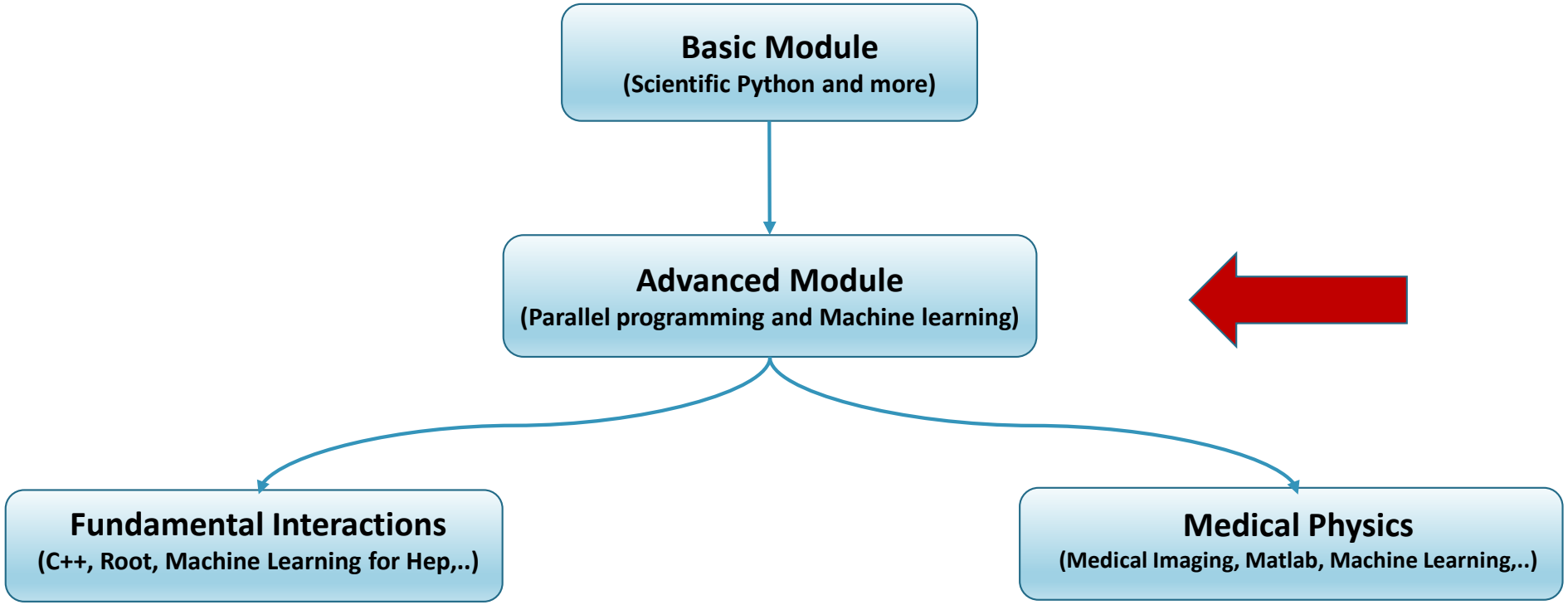
Computer architecture from a performance point of view: from serial to parallel

Computing Methods for Experimental Physics and Data Analysis

gianluca.lamanna@unipi.it

Where we are?



- Modularity
  - ➞Each module is worth 3 credits
  - ➞6 credits: basic+advanced
  - ➞9 credits: basic+advanced+fundamental interactions
  - ➞9 credits: basic+advanced+medical physics

Where we are?

- Advanced code development
  - Unit testing, continuous integration, static analysis, documentation
- Advanced Python
  - Errors, exceptions, iterators and generators, decorators
  - Profiling and optimization
- Parallel computing 
  - Computer architectures, memory, scaling laws, CPUs and GPUs
  - Parallel programming: concurrency and parallelism, threading in Python
- Machine learning
  - Classification and regression: boosted decision trees and multilayer perceptrons
  - Deep learning: neural networks, the keras library
  - Supervised and unsupervised training, reinforcement learning
  - Tensorflow

Introduction

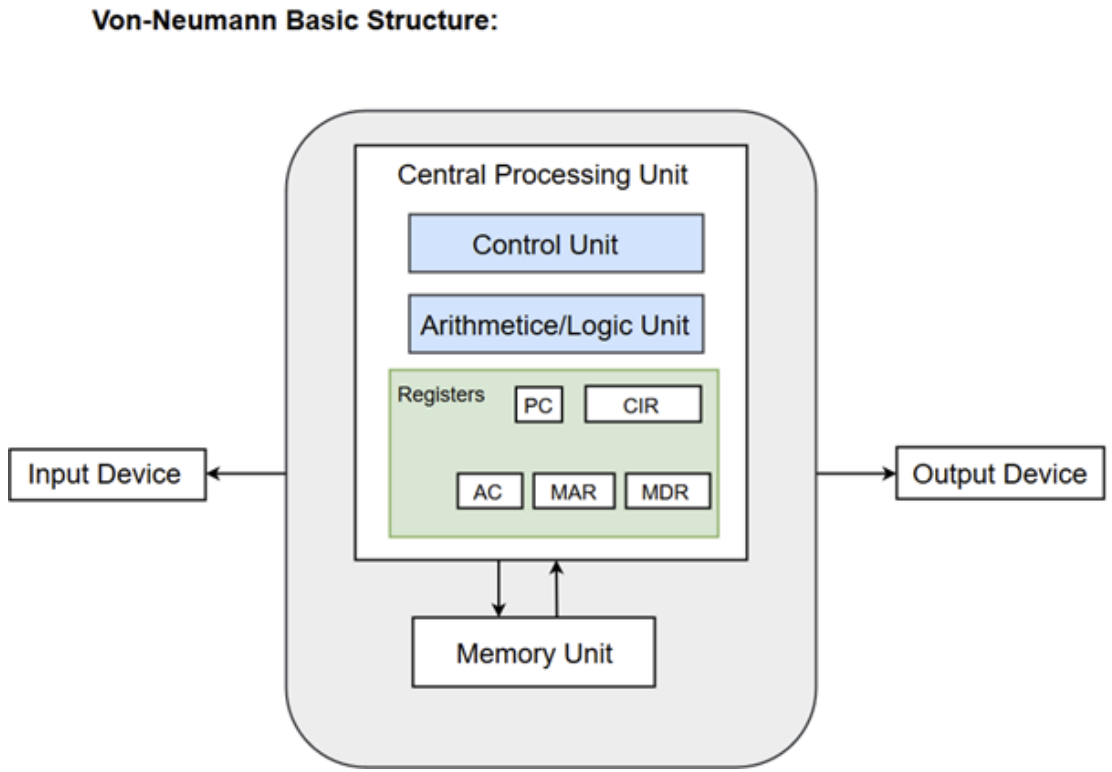
- In these lessons we will introduce the basic concepts of ***parallel programming***
- Thursday 21/10 «Computer architecture from a performance point of view» & «Concurrency and Parallelism in Python»
- Monday 25/10 : «Examples of Parallel programming in Python» (→Hands-on) & «Introduction to GPU programming (1)»
- Thursday 28/10 : «Introduction to GPU programming (2)» & «PyCUDA» (→Hands-on)
- Thursday 4/11 Morning: «Examples of more advanced GPU programming» (TBC)

Goal of this lecture

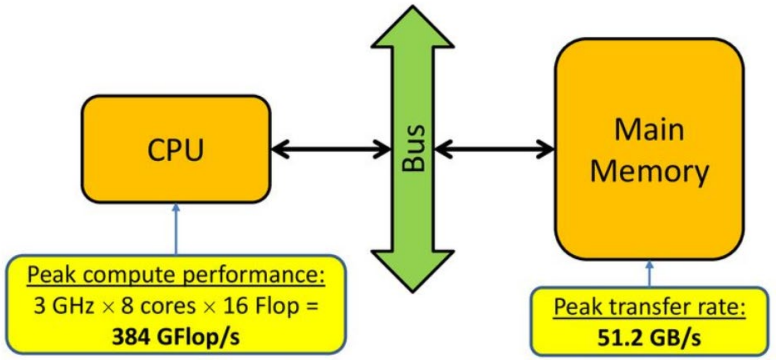
- Introduce key concepts of computer architecture
  - ➡ Identify which are the characteristics of the processors that influence most the computing performance
- Identify the limits of the standard architecture: Dennard and Moore scaling
- Flynn's taxonomy
- Concurrency and parallelism
- The limits of parallelism: Amdahl and Gustafson laws

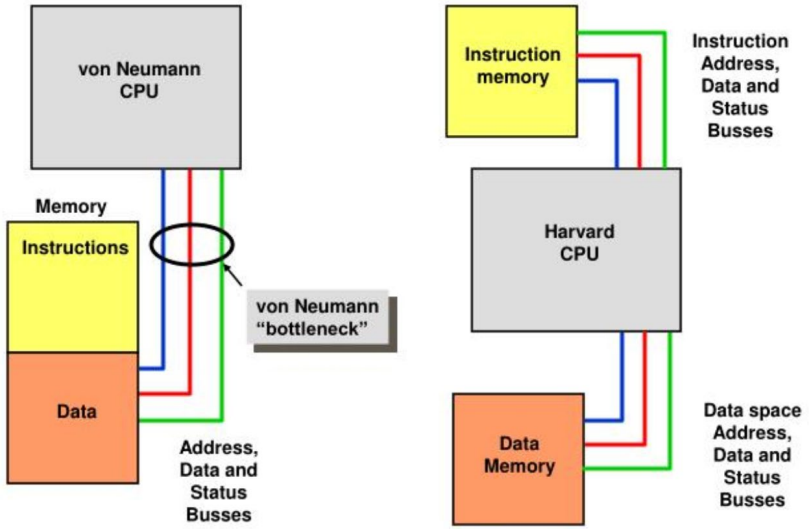
Von Neumann architecture

- Introduced in 1945 by John Von Neumann
- 5 elements:
  - Processing unit (arithmetic logic unit)
  - Control unit (instruction pool)
  - Memory
  - Bus
  - I/O
- Early idea by Alan Turing (1936)
  - Universal Turing Machine (UTM)



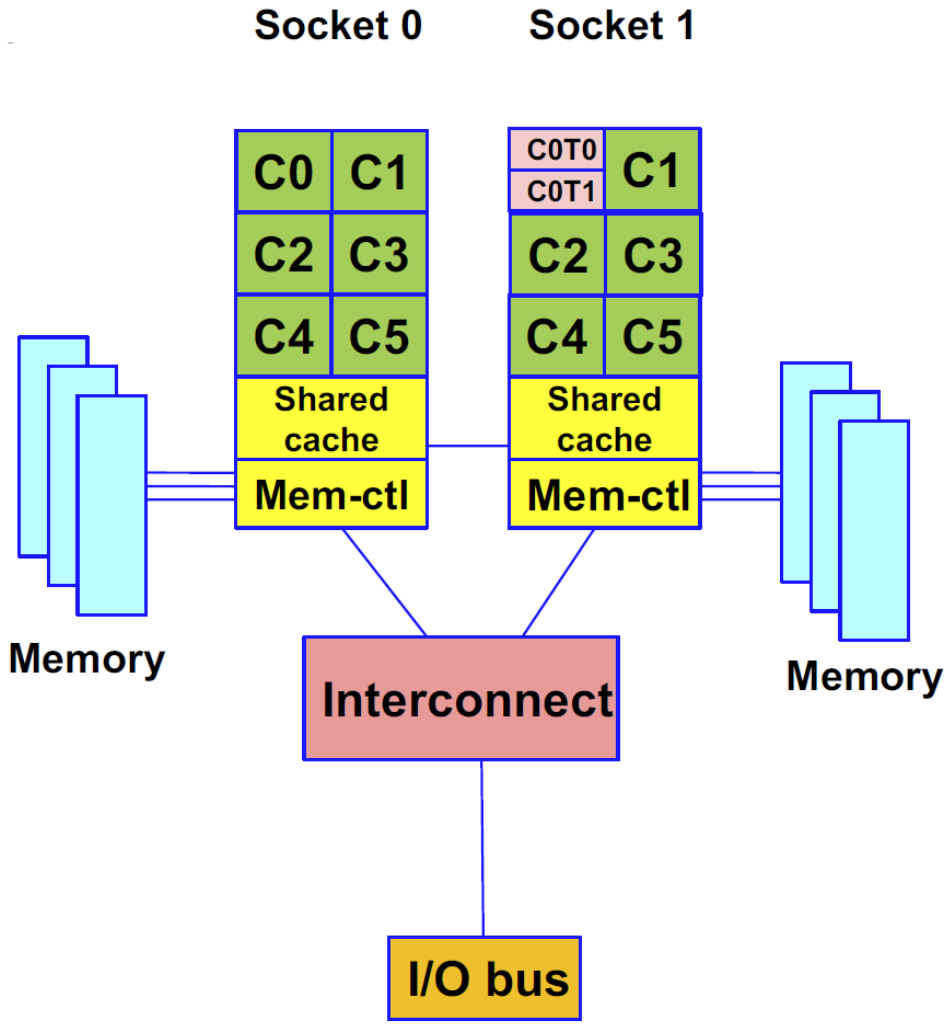
Von Neumann Bottleneck





- The bottleneck comes from the sharing of the same bus for data and instruction
  - ➡ Limited throughput of data and instruction transfer compared to memory size
- Several strategies to mitigate:
  - ➡ Caching and memory gerarchy on chip
  - ➡ Separate access to data and instructions → Hardvard Architecture
  - ➡ Branch prediction

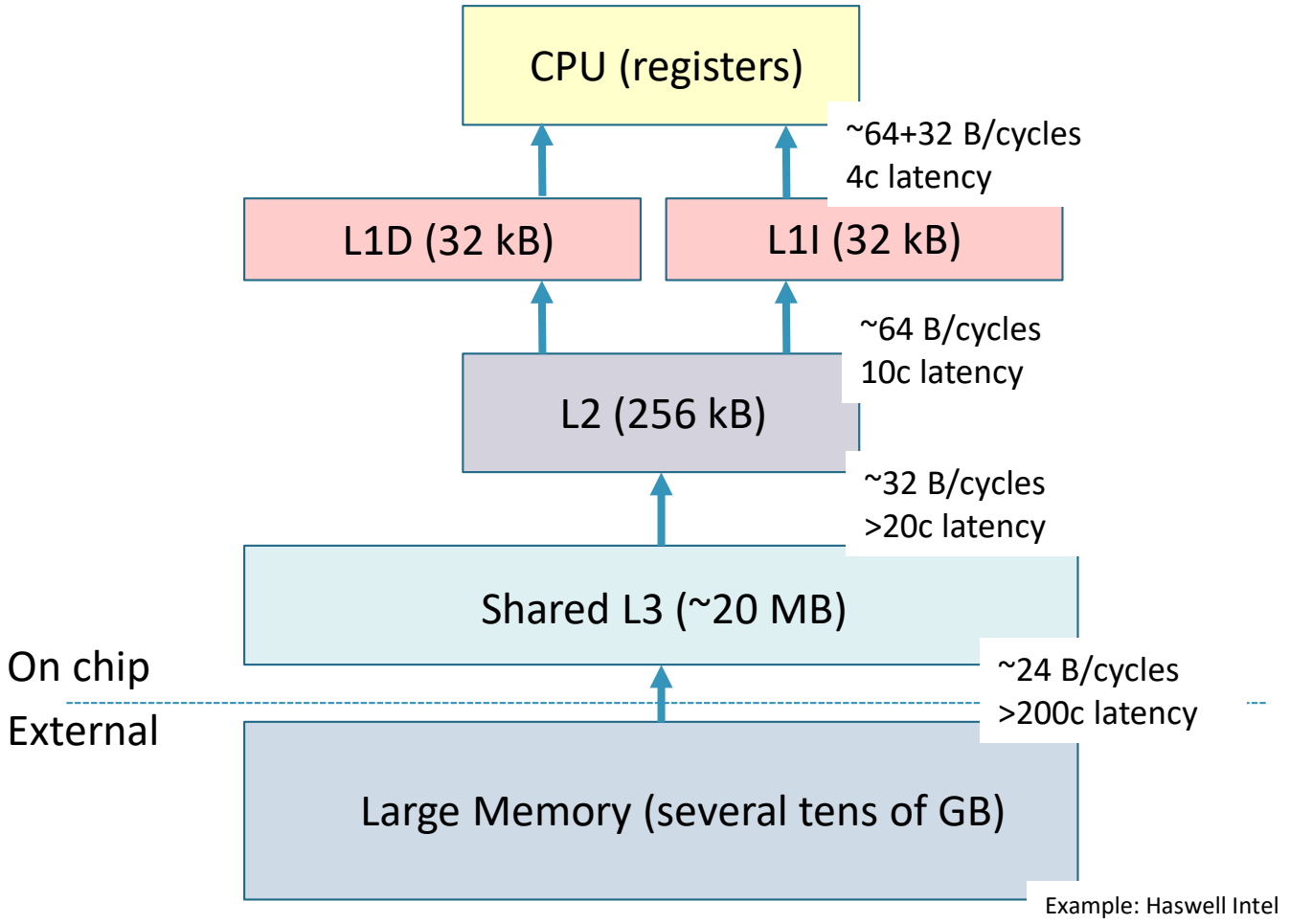
Simple server architecture



- In a server multiple components interacts during the program execution
  - ➡ Processors/cores
    - I-cache, D-cache
  - ➡ Shared Caches
    - For instruction and data
  - ➡ Memory controllers
  - ➡ I/O subsystems
    - Storage, network, peripherals
- Example: NUMA architecture (non-uniform memory access)

Memory

- Multilevel cache
  - ➡ Tradeoff between hit-rate and latency
  - ➡ Specialized caches, victim cache, trace cache, Write coalescing cache,…
- The hierachy of both data access and instruction fetching is fundamental in present computer architecture



Seven dimensions of Performance

- The «modern» PC performance depends on (at least) seven characteristics:
  - Hardware vectors
  - Superscalars
  - Pipelining
  - Hardware multithreading
  - Multiple cores
  - Multiple sockets
  - Multiple nodes

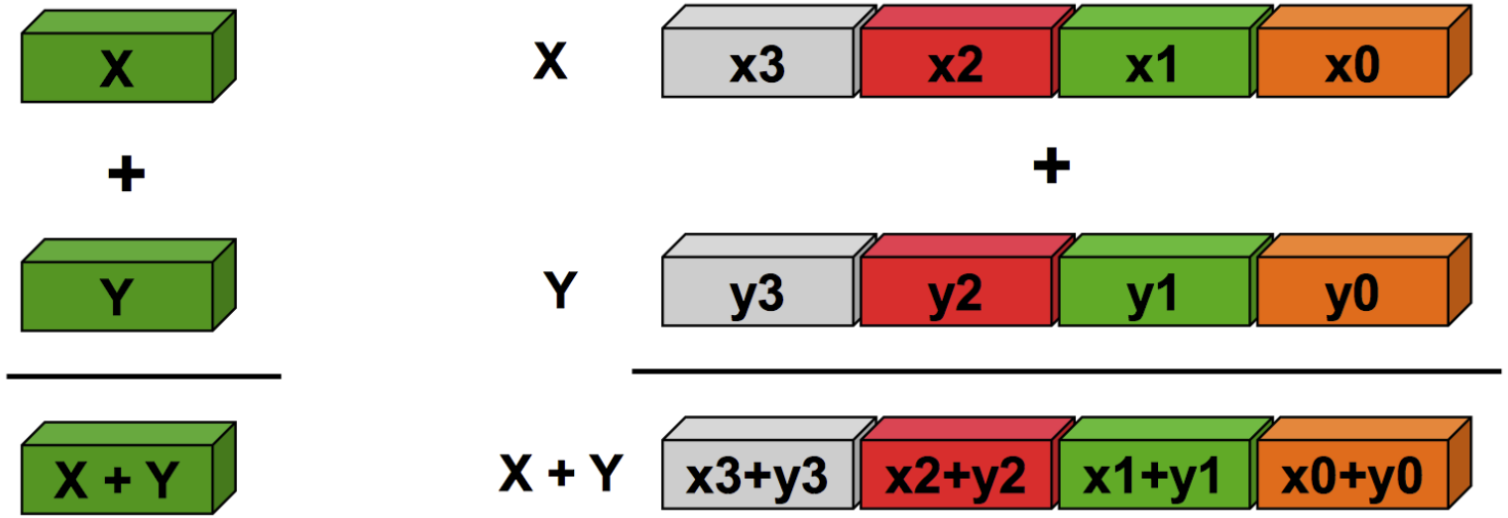
Data and Instruction level

Task level

Application level

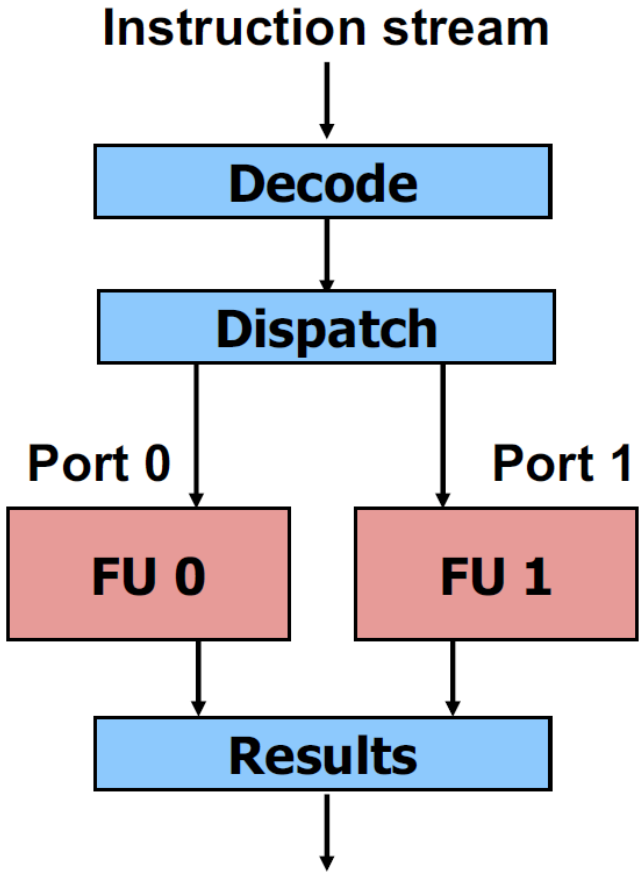
Vector processors

- Modern processors implement registers for vectorialization (SSE/SSE2 and AVX)
- Scalar mode:
  - ➞One operation produces one result
- SIMD (Single Instruction Multiple Data) is a simple way to parallelize
  - ➞One operation produces multiple results



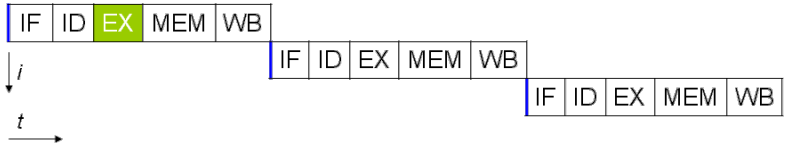
Superscalars

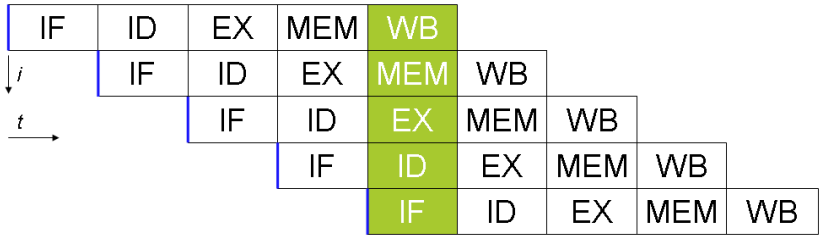
- Architecture between pure «scalar» and pure «vector»
  - ➡Several hardware units can execute different operation on different data at the same time
- Functional Units (FU) can have identical or different computing capabilities
  - ➡Decoder and Dispatcher must have the capability to manage two instruction in on clock cicle
- Usefull for *Branch Prediction*
  - ➡Execute at the same time different branches i an algorithm then choose the correct one



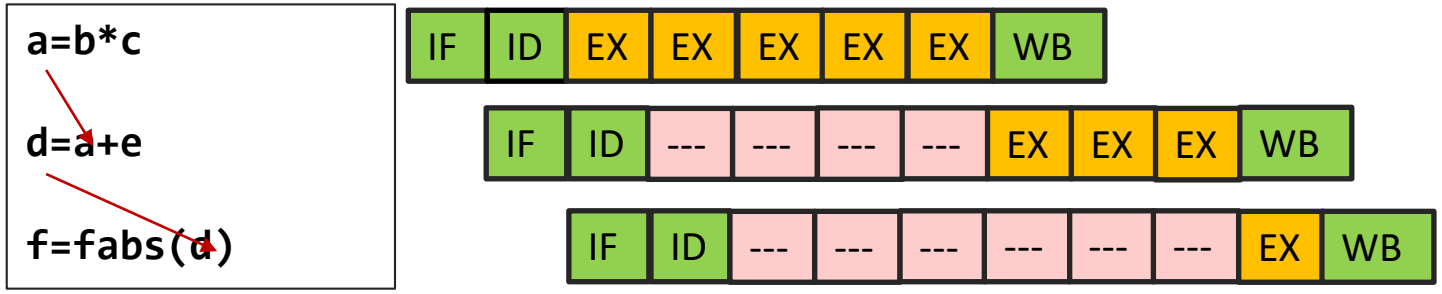
Pipelining

- Integer/logic instructions have a *latency*
  - 1 cycle: ADD, AND, SHL, ROR, FABS
  - 3 cycles: IMUL, FADD
  - 5 cycles: FMUL
  - 13-23 cycles: IDEV
  - 27: FSQRT
- The pipeline is an important ingredient in modern processors:
  - Capability to execute different stage of consecutive instructions at the same time





- It isn't always possible to fully exploit the pipeline

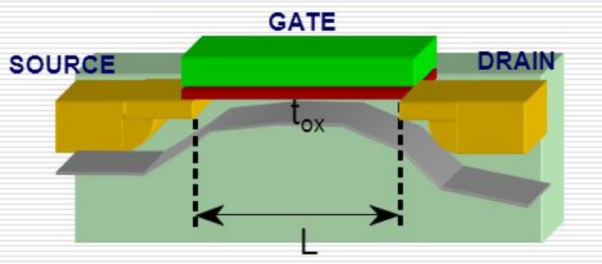


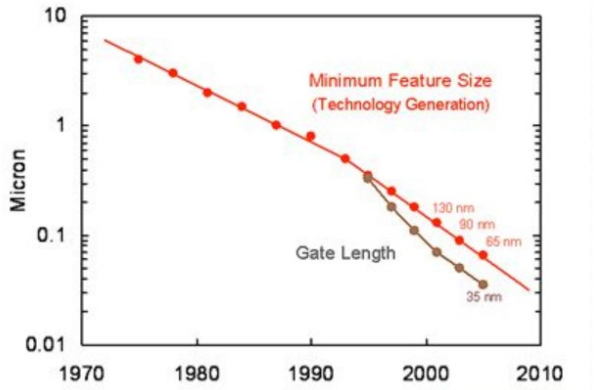
Recap

- Superscalars, Pipelining and Vectorialization are methods to exploit some «parallelism» at the instruction and data level: ILP
  - Probably OOO (Out-of-order) execution should be included in this category
- The possible improvement thanks to ILP depends on problem and data structures
  - 1x-10x for Superscalars and Pipeline
  - 2x,4x,8x,16x for the vectorialization
- These methods show «saturation» because they are limited by the CPU resources available
  - Pentium 4: 30 pipeline stages (nowadays 10-15 maximum)
  - ARM A57 (Apple A7/A8): 9 ports/6 instructions superscalar
  - Intel Tiger Lake: vector of 512 bits for a subset of AVX512 instructions

… the point is: can CPU resources grow indefinitely?

Dennard scaling





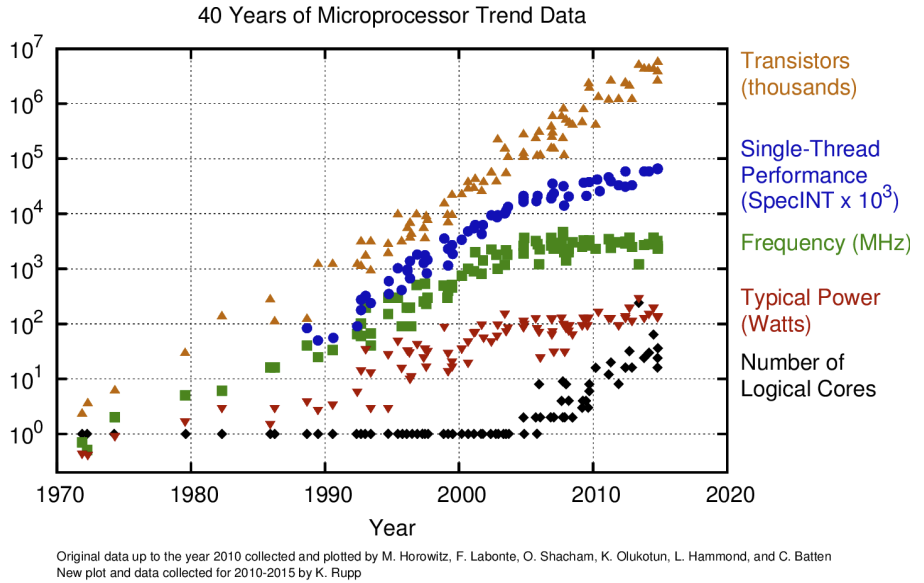
Device or Circuit Parameter	*Scaling Factor*
Device dimension t_{ox}, L, W	$1/k$
Doping concentration N_a	k
Voltage V	$1/k$
Current I	$1/k$
Capacitance eA/t	$1/k$
Delay time per circuit VC/I	$1/k$
Power dissipation per circuit VI	$1/k^2$
Power density VI/A	1

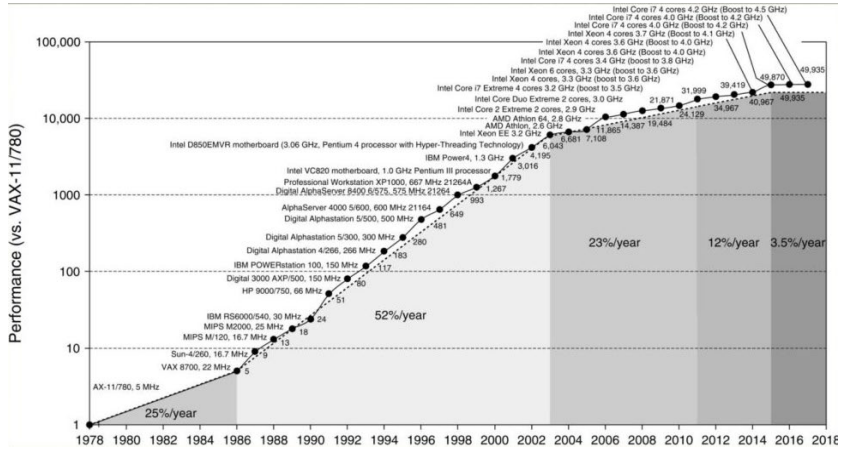
Table I: Scaling Results for Circuit Performance (from Dennard)

- Aka MOSFET scaling
  - ➡ Dennard scaling after an article from Dennard et al. in 1974 in IEEE Journal of Solid State Circuits
- In each generation of CMOS based IC the power consumption remains the same
- Breakdown of Dennard scaling around 2006
  - ➡ With very small integration it is not true anymore that the power consumption is the same, due to increasing in current leakage
  - ➡ The increasing of the speed of the transistors switching (frequency) is not anymore linear with the performance of the CPU
- Energy consumption has become more important to users
  - ➡ For mobile, IoT, and for large clouds
- Processors have reached their power limit
  - ➡ Thermal dissipation is maxed out (chips turn off to avoid overheating!)
  - ➡ Even with better packaging: heat and battery are limits

Moore scaling

- The Moore's «law» is the empirical observation that the number of transistors doubles about each two years
  - ➡ The performance of CPU doubles each 18 months
- Moore's prediction was verified for decades
- Around 2005 is starts to show saturation
- Closely related to Dennard scaling





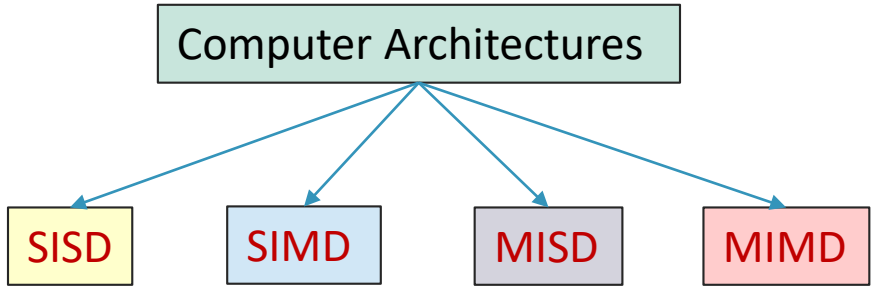
Hardware parallelism

- How to avoid saturation?
- Instruction level parallelism achieved significant performance advantages
  - ➡ But the performance are related to clock speed
  - ➡ Increasing in ILP is still possibile but the complexity of CPU is more than linear → diminishing return in efficiency
- We need a next level in parallelism
  - ➡ Task level!

Flynn's taxonomy

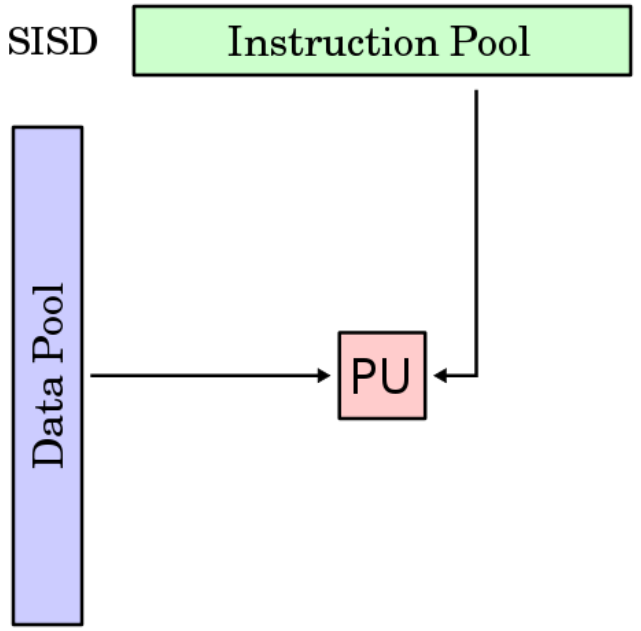
- Classification of computers architectures based on the number of data streams and instructions streams

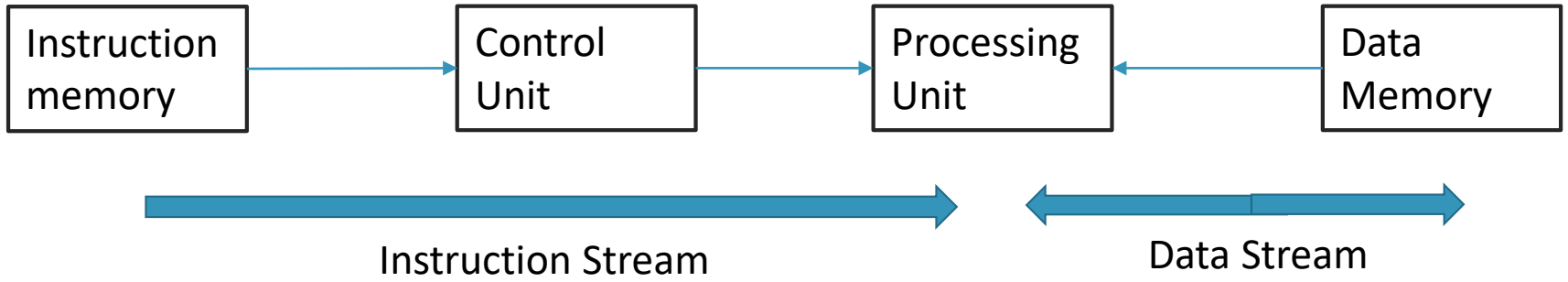
- Single Instruction Single Data (SISD): Traditional sequential computing
- Singe Instruction Multiple Data (SIMD)
- Multiple Instructions Single Data (MISD)
- Multiple Instructions Multiple Data (MIMD)



SISD: Single Instruction Single Data

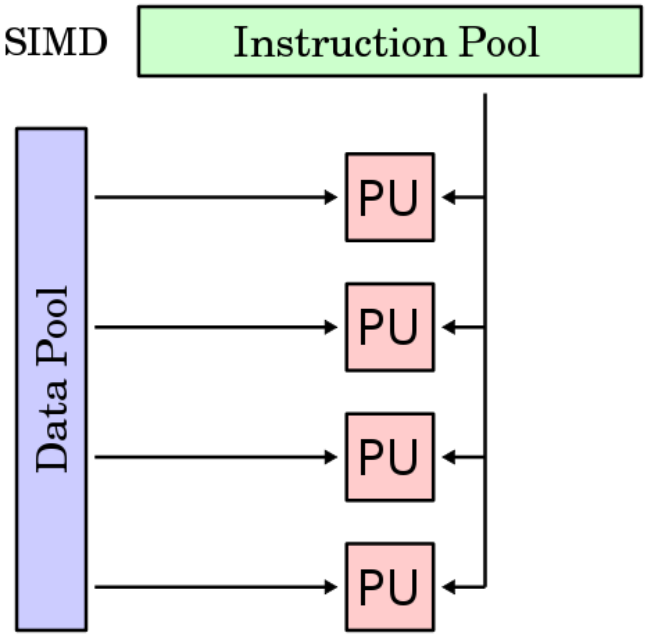
- Only one instruction operates for each time slot on one data
  - ➔Sequential processing

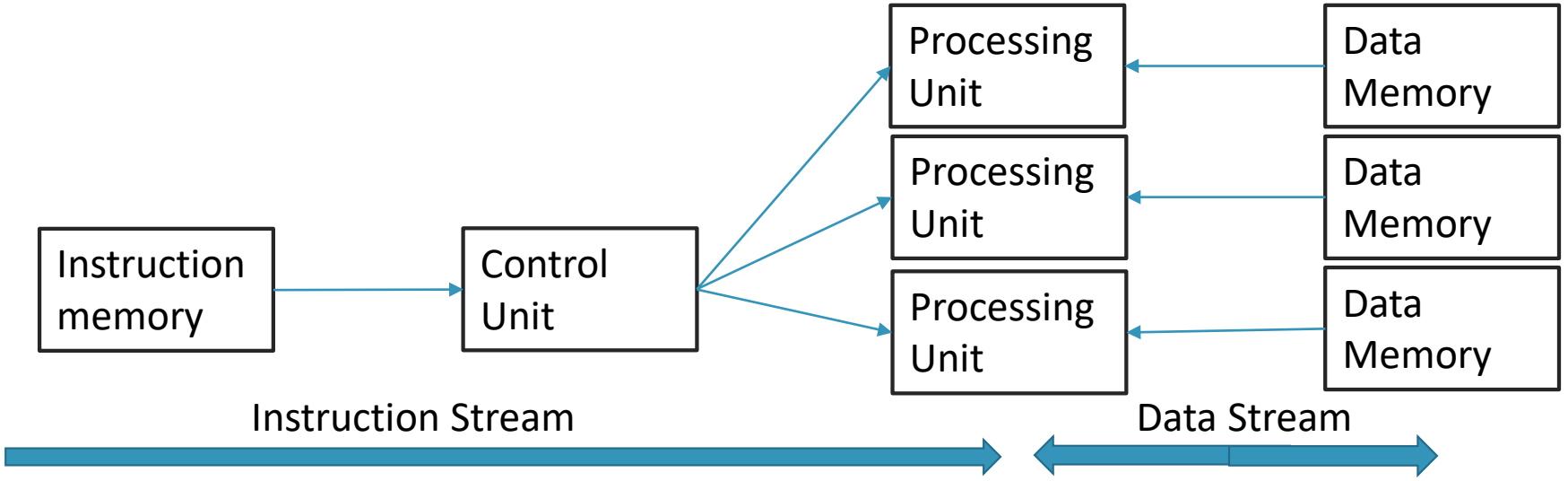




SIMD: Single Instruction Multiple Data

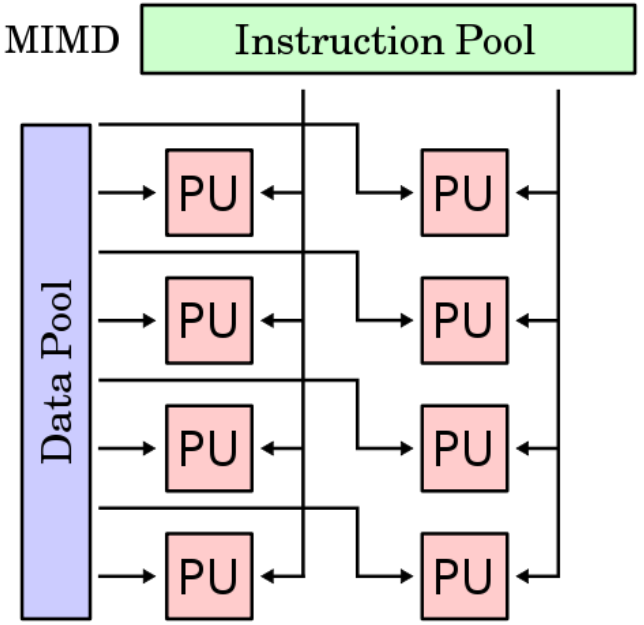
- At one time one instruction operates on multiple data
  - ➡ Very similar to vector processors
    - Although in the vector architecture the parallelism is obtained with a pipeline, while in SIMD the operations are really parallel on vector's element.
  - ➡ Array processors
  - ➡ Most modern processors contain one or more SIMD sections

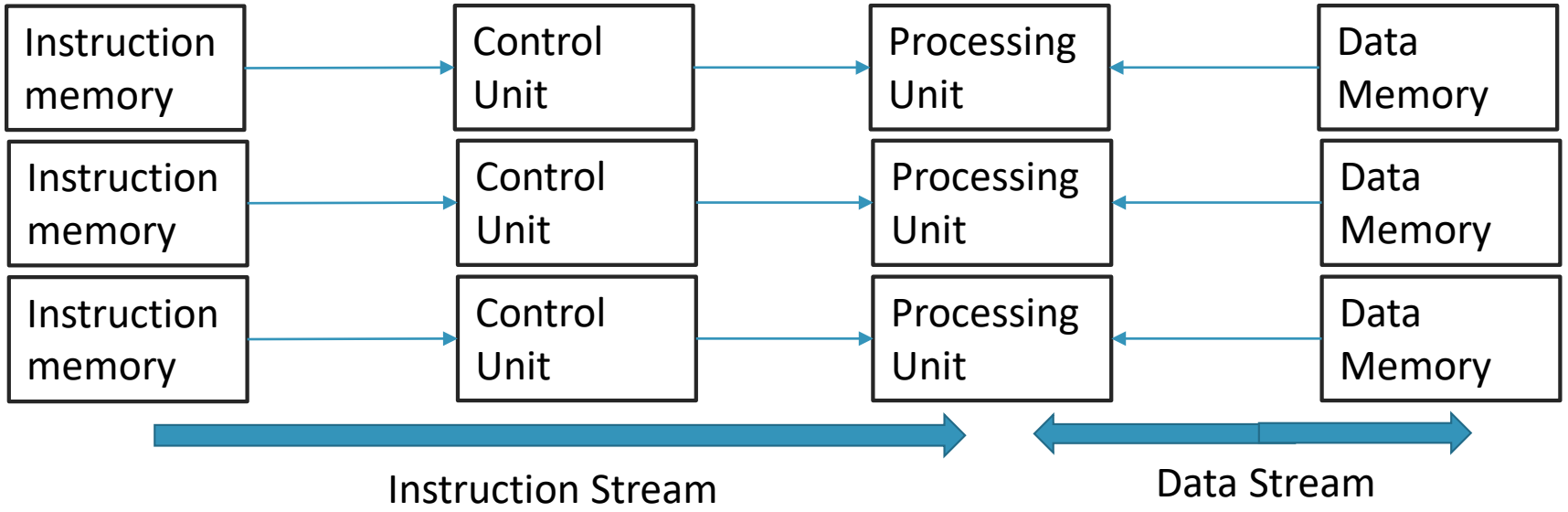




MIMD: Multiple Instruction Multiple Data

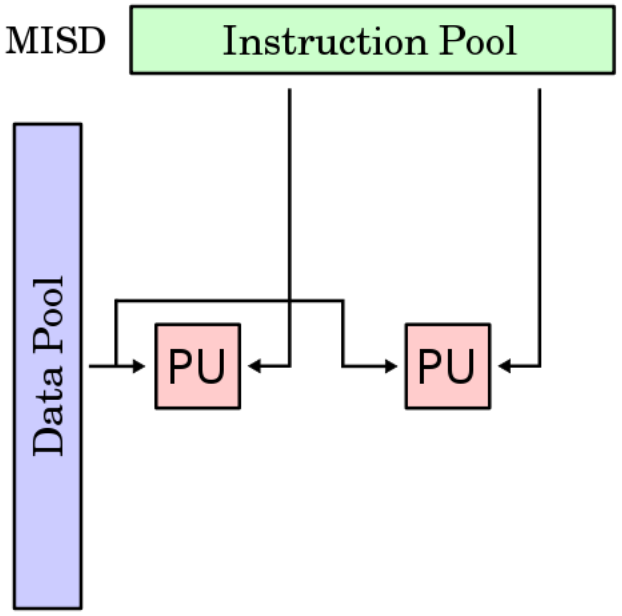
- Multiple instructions streams operate on multiple data stream
  - ➡Most of supercomputers are organized as MIMD architecture
  - ➡Multi-core superscalar, multi-processors and distributed systems

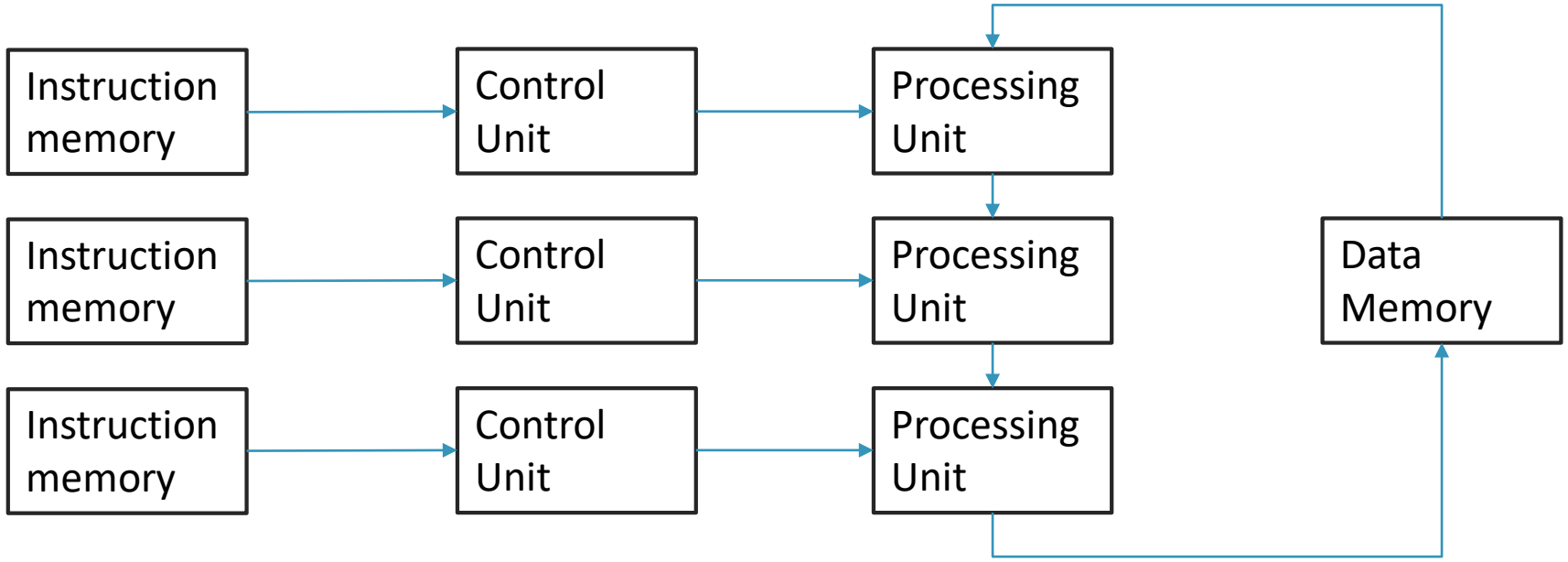




MISD: Multiple Instruction Single Data

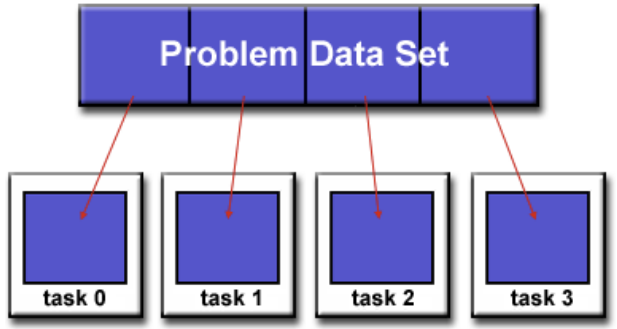
- Not commonly seen
  - ➡Sometime the systolic array is seen as MISD
  - ➡Usually is an architecture used for fault tollerance and not for computing

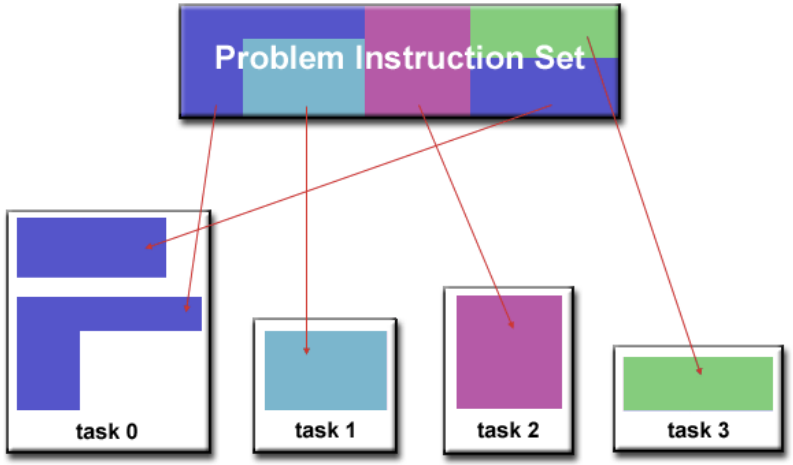




Logic partitioning and decomposition

- The choose of the architecture depends on the problem
- Domain decomposition
  - ➡Single program, multiple data
  - ➡ decomposition based on Input domain, output domain, both
- Functional decomposition
  - ➡Multiple programs, multiple data
  - ➡Independent tasks
  - ➡Pipeling





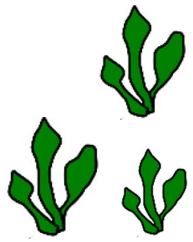
Multiprocessor Execution Model

- A specific architecture is suitable for a specific problem, but all needs «multiprocessors»
- Examples:
  - Each processor has its own PC and executes an independent stream of instructions (MIMD)
  - Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations

- Two ways to use a multiprocessor:
  - Deliver high throughput for independent jobs via job-level parallelism
  - Improve the run time of a single program that has been specially designed to run on a multiprocessor - a parallel-processing program

Sequential processing

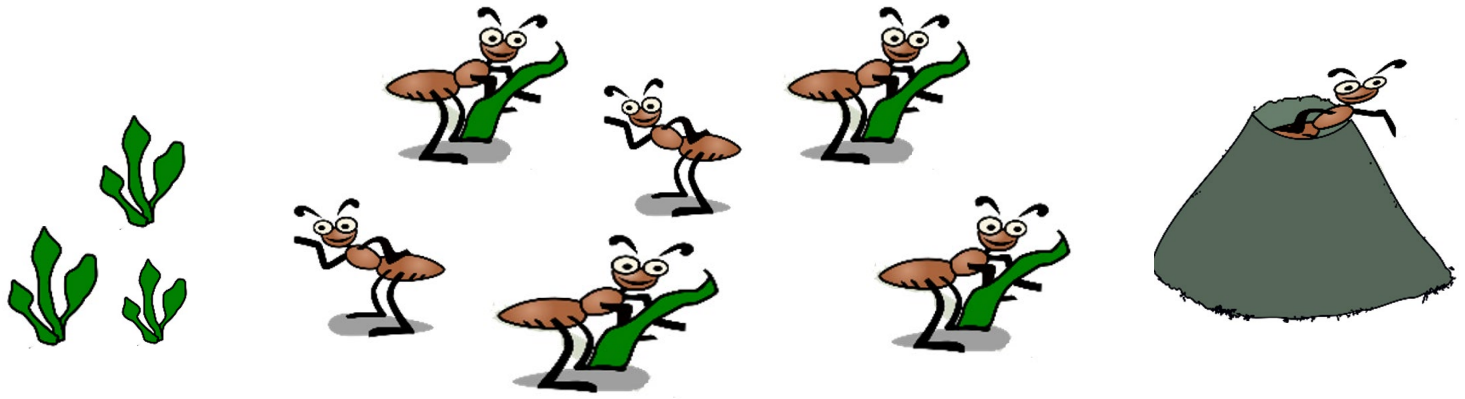
- Only one “thread” of execution
  - ➡ One step follows another in sequence
  - ➡ One processor is all that is needed to run the algorithm
- Thread definition: It is the smallest of a program that can be managed independently by a scheduler (typically in the operating system).
  - ➡ A thread is a component of a process
  - ➡ Multiple threads can exist within one process
  - ➡ Systems with a single processor generally implement multithreading by time slicing (software threads)



Concurrent Processing

- A system in which:
  - ➡ Multiple tasks can be executed at the same time
  - ➡ The tasks may be duplicates of each other, or distinct tasks
  - ➡ The overall time to perform the series of tasks is reduced

- Advantages:
  - ➡ Concurrent processes can reduce duplication.
  - ➡ The overall runtime of the algorithm can be significantly reduced.
  - ➡ More real-world problems can be solved than with sequential algorithms alone.
- Disadvantages
  - ➡ Runtime is not always reduced, so careful planning is required
  - ➡ Concurrent algorithms can be more complex than sequential algorithms
  - ➡ Shared data can be corrupted
  - ➡ Communication between tasks is needed

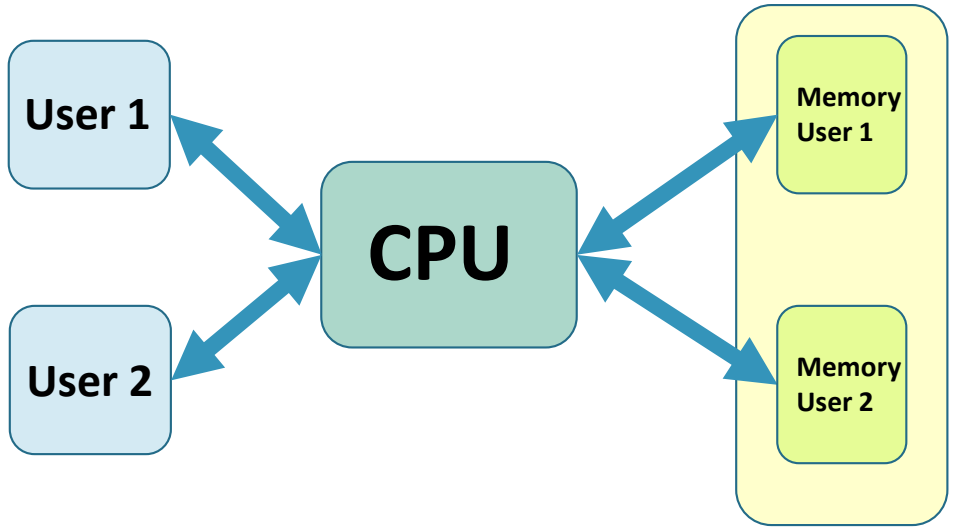


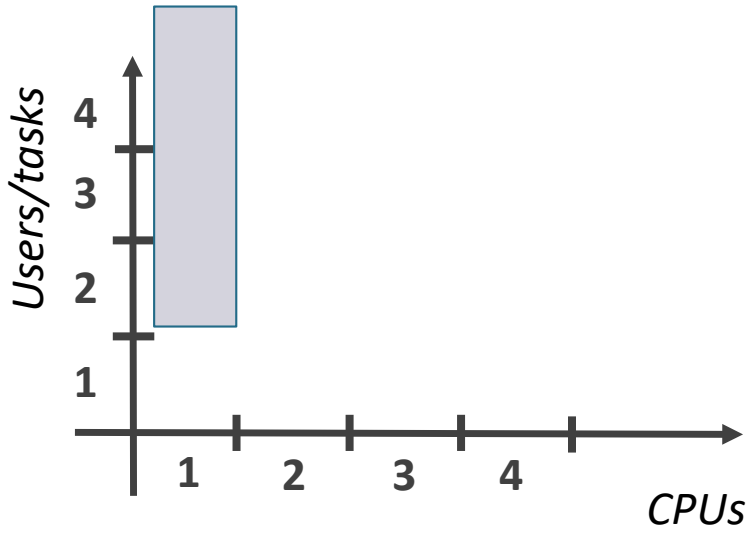
Types of concurrent processing

- Multiprogramming
- Multiprocessing
- Multitasking
- Distributed Systems

Multiprogramming

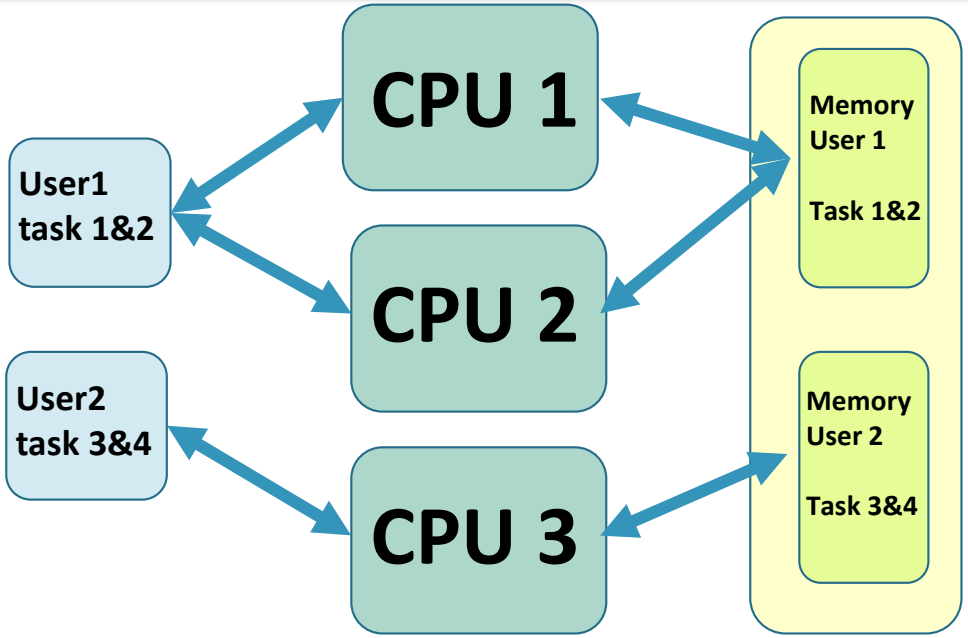
- Multiprogramming
  - ➔ Share a single CPU among many users or tasks.
  - ➔ May have a time-shared algorithm or a priority algorithm for determining which task to run next
  - ➔ Give the illusion of simultaneous processing through rapid swapping of tasks (interleaving).

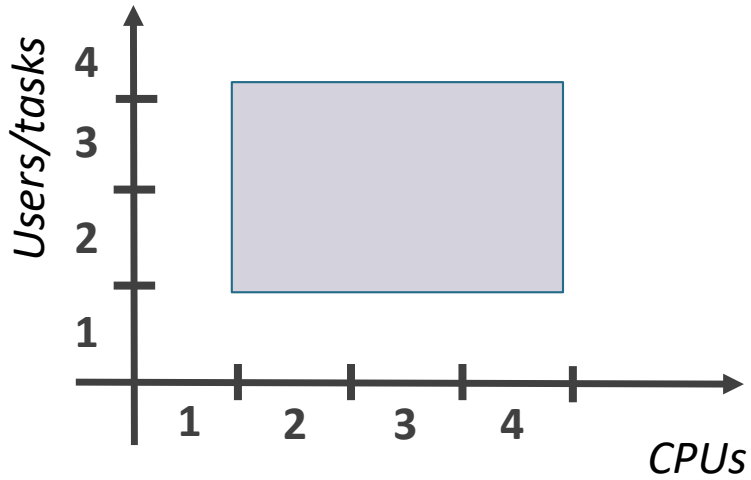




Multiprocessing

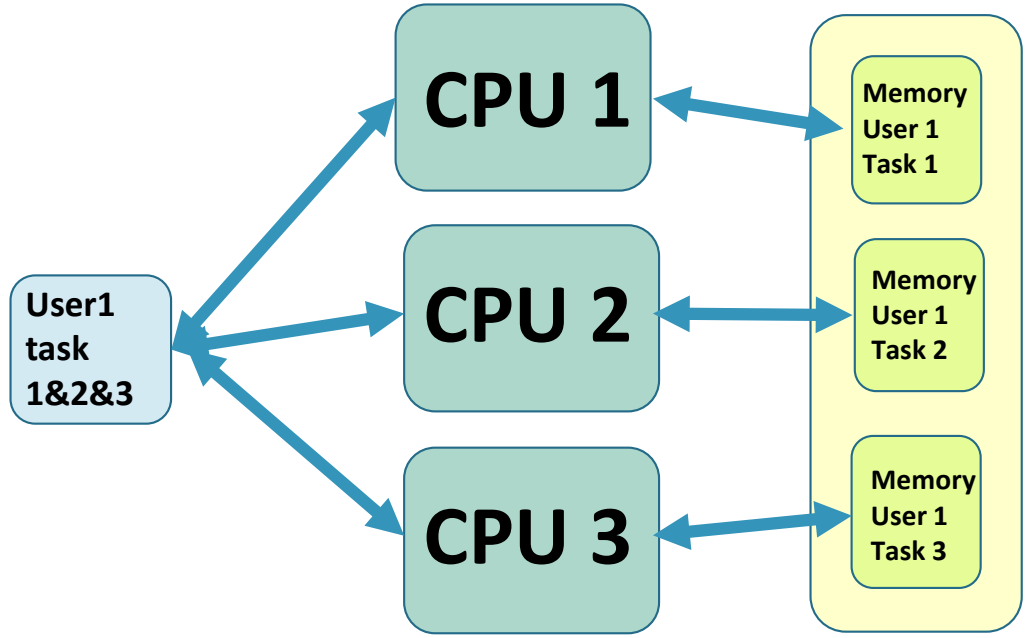
- Multiprocessing
  - Executes multiple tasks at the same time
  - Uses multiple processors to accomplish the tasks
  - Each processor may also timeshare among several tasks
  - Has a shared memory that is used by all the tasks

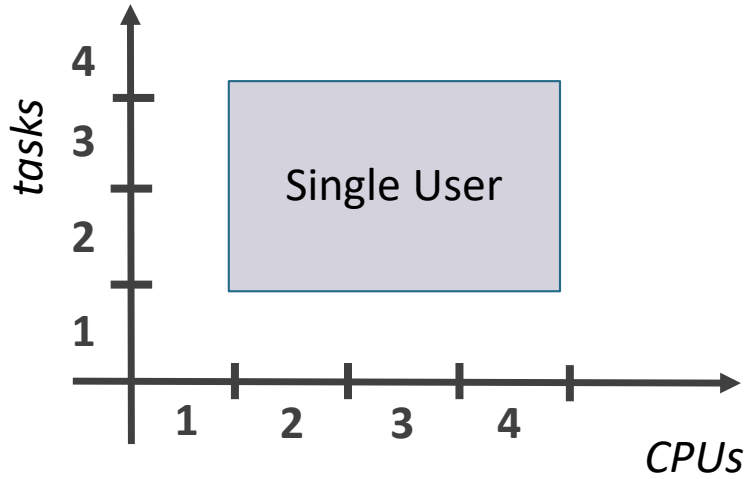




Multitasking

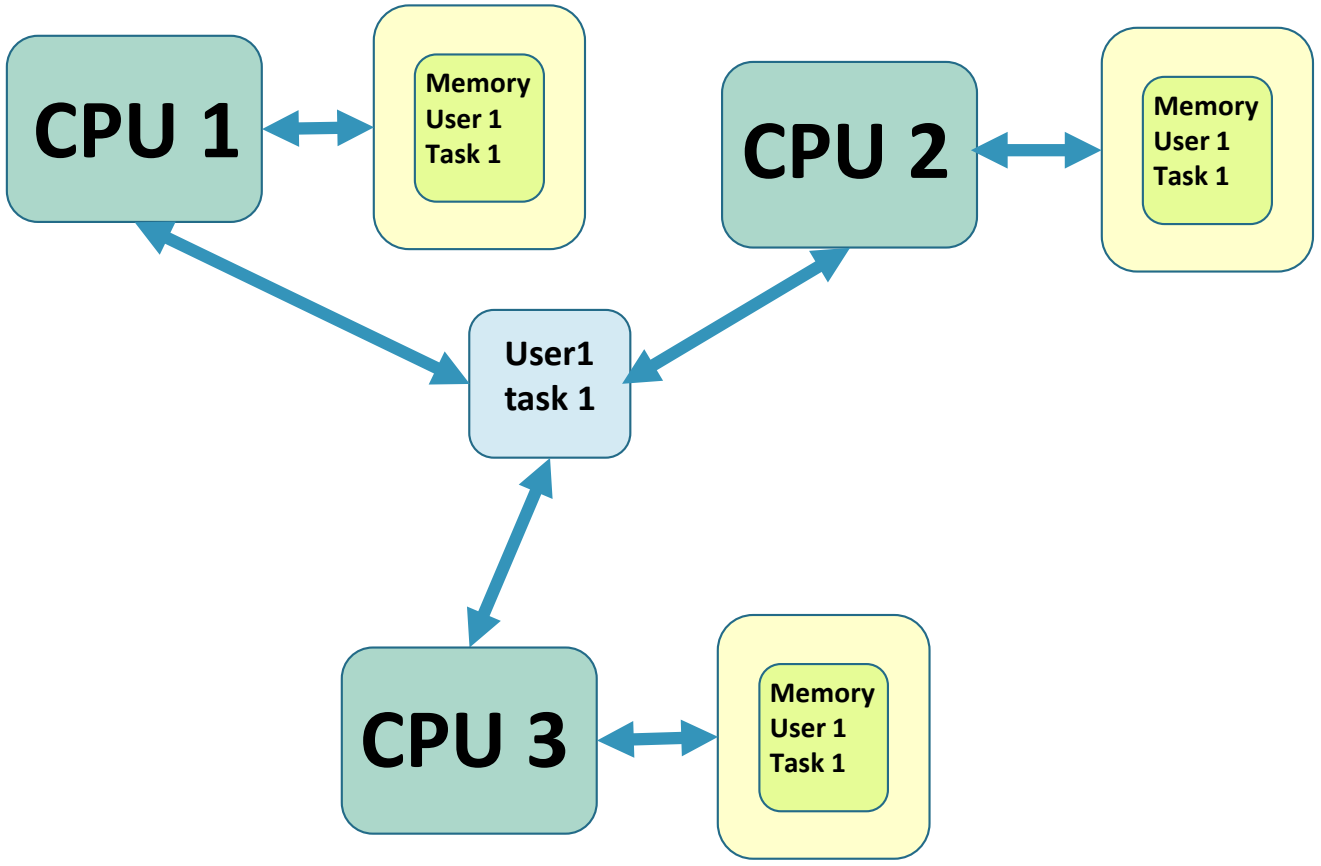
- Multitasking
  - A single user can have multiple tasks running at the same time.
  - Can be done with one or more processors.
  - Used to be rare and for only expensive multiprocessing systems, but now most modern operating systems can do it.





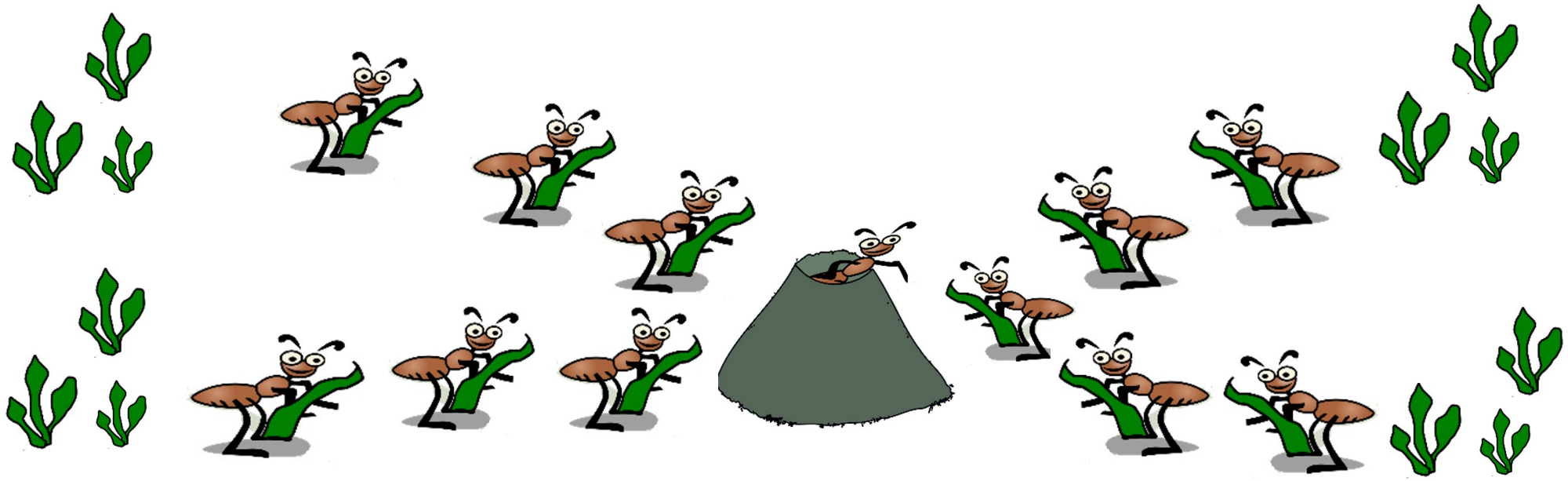
Distributed systems

- Distributed systems
  - Multiple computers working together with no central program “in charge.”
  - No bottlenecks from sharing processors
  - No central point of failure
  - Complexity
  - Communication overhead
  - Distributed control



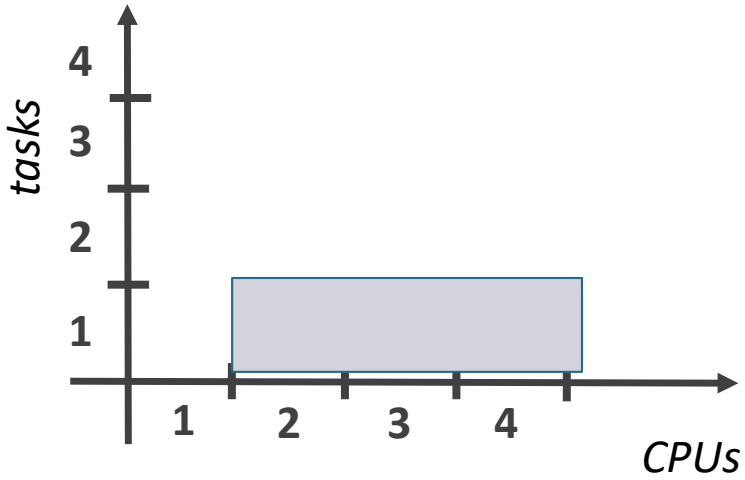
Parallelism vs Concurrency

- Concurrency is the execution of multiple tasks at the same time, regardless of the number of processors.
- Parallelism is the execution on multiple processors on the same task.
  - Breaking the task into meaningful pieces
  - Doing the work on many processors
  - Coordinating and putting the pieces back together.



Parallelization

- For a wide class of algorithms parallelization is the most powerfull way to decrease execution time (not complexity)
  - ➡ Example: a problem with O(NlogN) complexity (for instance Quicksort) on logN processors will take the time needed by O(N) algorithms
  - ➡ Example: a problem with O(N^2) complexity (for instance binary search) on N processors will take the time needed by O(N) algorithms
- Parallelization is not free.
- Processors must be controlled and coordinated.
  - ➡ We need a way to govern which processor does what work; this involves extra work.
- Often the program must be written in a special programming language for parallel systems.
- Often, a parallelized program for one machine (with, say, 2K processors) is not optimal on other machines (with, say, 2L processors).



Speedup and Efficiency

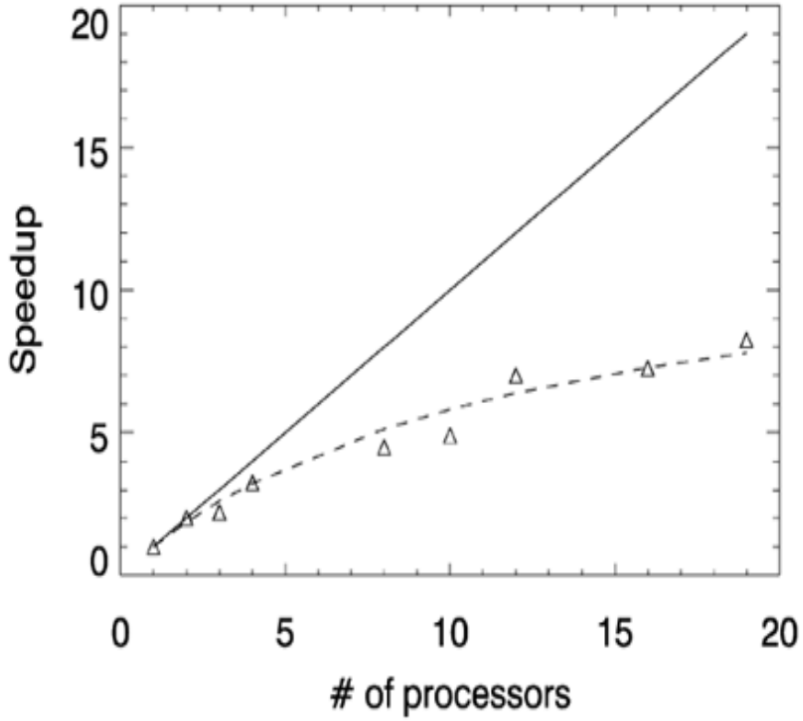
- How much gain can we get from parallelizing an algorithm?
- Let's define the «speedup» as (where n is the number of processors)

$$S_n = T_{serial}/T_{parallel}(n)$$

- For an perfect parallel algorithm $S_n = n$
  - ➡ Pratically impossible, even if for very specific case could be also $S_n > n$ (superlinear case)
- The efficiency is defined as

$$E = S_n/n$$

  - ➡ It is a measure of how well our algorithm is using the processors



Cost and Scalability

- Cost: the number of CPU required

$$c = nT_p(n) = \frac{T_1}{E}$$

- Scalability: capability to remain efficient with the increasing of the number of processors

Amdhal's law (1967)

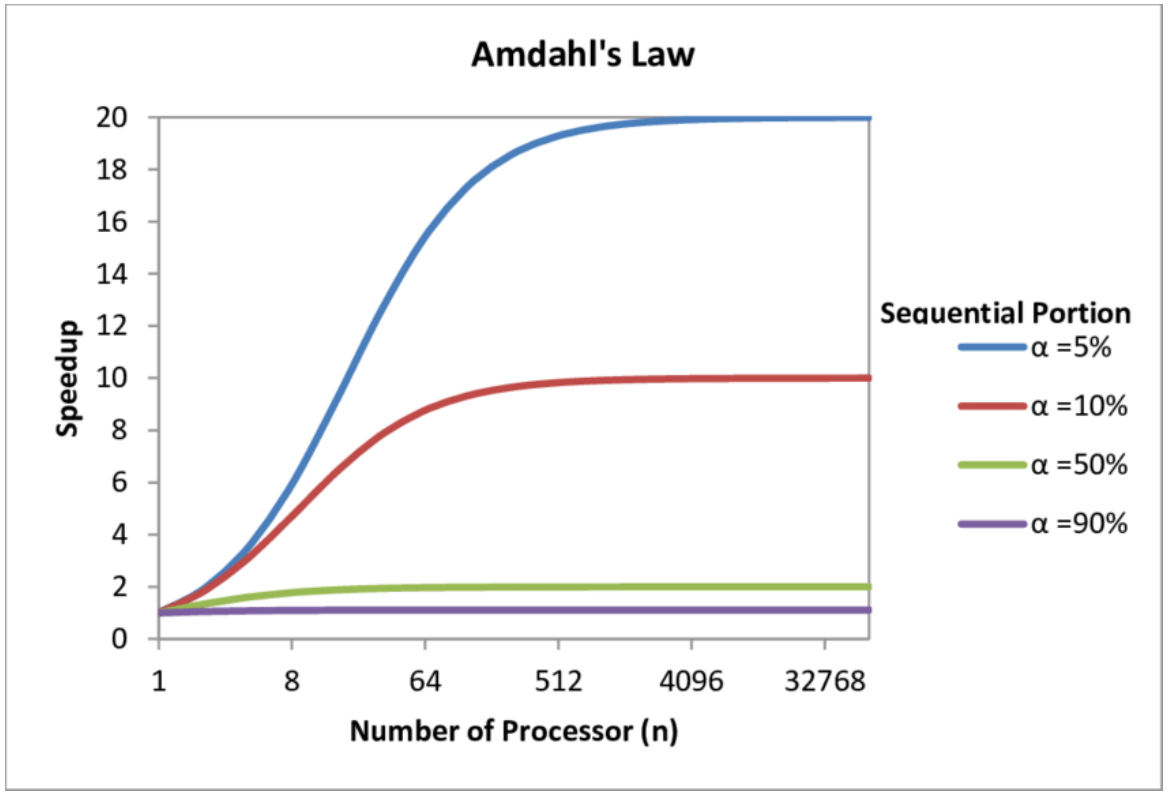
- If only one part (P_K) of the code can be improuved, the maximum improvement is given by

$$1/\sum \frac{P_K}{S_k}$$

- Where k is the part of the code and S_k is the speedup of the part-k
- In the case of parallel programming

$$S_n = \frac{n}{nF + (1 - F)}$$

- Where F is the fraction of the code that can not be parallelized
  - If n→∞ the speedup is $S_n = \frac{1}{F}$
  - For instance if the fraction of serial code is 10% (F=0.10) the maximum speedup is «only» 10 (regardless the number of processors)
  - Apparently the parallelism is usefull only for «embarassingly parallel» problems, with a small number of processors



Overhead of parallelization

- Load balancing
  - In case of several tasks in parallel the execution time of each task must be similar
  - Otherwise the total time is dominated by the slower task
  - Some processor could be inefficently IDLE
  - It's not easy to design a priori a good load balancing
- Synchronization
  - If the tasks use the same memory (shared memory) to exchange data a logic of lock-unlock must be designed
  - This involves a waste of time
- Comunication latency
  - If data must be moved between processors the overhead due to data transmission can be really relevant

Limits of Amdhal's law

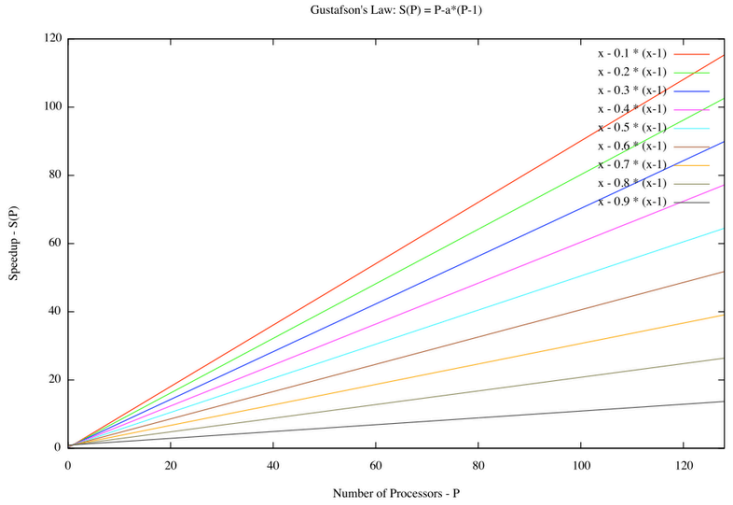
- Apparently the Amdahl's law puts important limits to the advantages of parallel computing.
- But there are importants caveat to this law:
  - Amdahl assumes that the best solution is always the best serial algorithm. Often some problem must be solved in parallel
  - Some architectural design can help parallel processing (for instance the caching)
  - Amdahl assumes that the dimension of the problem is always the same with the increasing of the number of processors. But more processors often means that wider problem can be addressed.

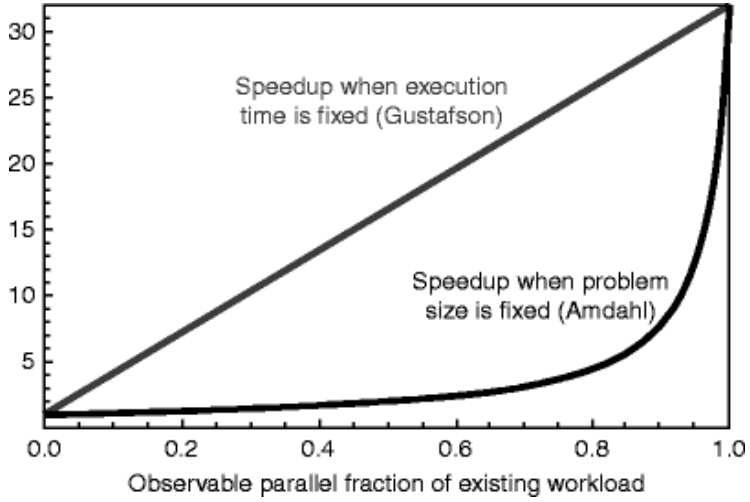
Gustafson's law (1988)

- Let's assume s is the time of the serial part (and p is the time parallel part)
- Let's assume that the problem grows with the number of processor (N) and that the serial part remains always the same
- Under these assumptions the speedup is given by

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

- The speedup is linear with N





Concurrency in Python

- See next lecture

Parallelism and GPU

- It’s the main argument of the lectures in the second part of this group

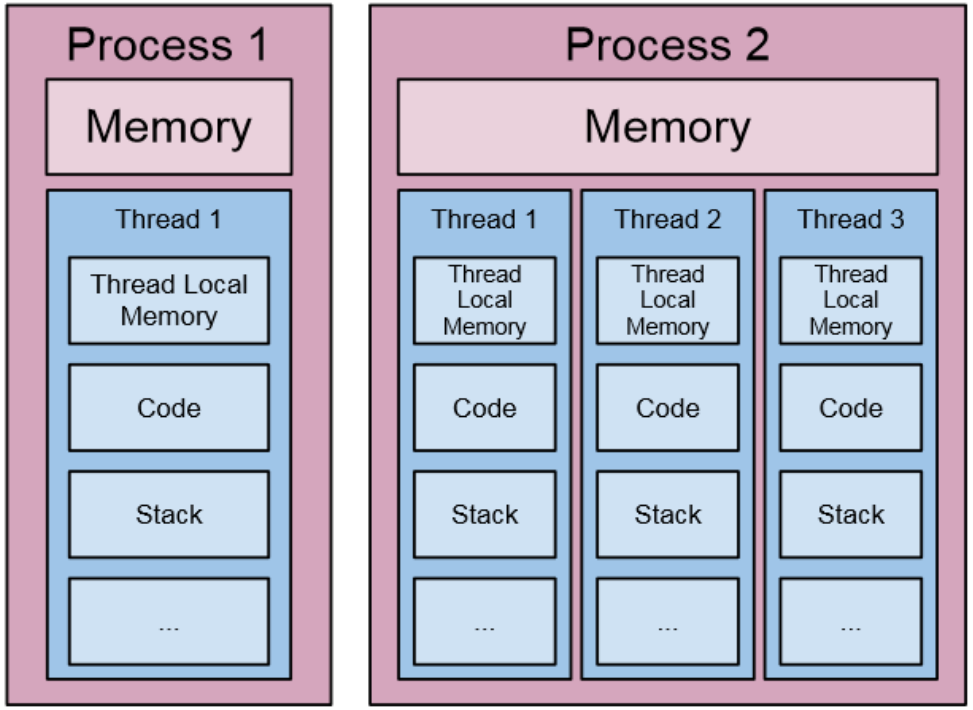
Multithreading and multiprocessing in Python

Computing Methods for Experimental Physics and Data Analysis

gianluca.lamanna@unipi.it

Threads and processes

- Threads and processes are the way to use concurrency in python
- Python implements a very simple thread-safe mechanism: Global Interpreter Lock (GIL).
  - ➡ In order to prevent conflicts only one statement in one thread is executed at a time (single-threading)



The Global Interpreter Lock (GIL)

- The Global Interpreter Lock refers to the fact that the Python interpreter is not thread safe.
- There is a global lock that the current thread holds to safely access Python objects.
- Because only one thread can acquire Python Objects/C API, the interpreter regularly releases and reacquires the lock every 100 bytecode of instructions. The frequency at which the interpreter checks for thread switching is controlled by the sys.setcheckinterval()function.
  - ➡ In addition, the lock is released and reacquired around potentially blocking I/O operations.
- It is important to note that, because of the GIL, the CPU-bound applications won't be helped by threads.
  - ➡ In Python, it is recommended to either use processes, or create a mixture of processes and threads.

Process: pros and cons

- A process is an instance of a program
- Managed by operating system
  - → Memory space allocated by the kernel
- Two processes can execute code simultaneously in the same python program
- Separated memory space
- Takes advantage of multiple cores and CPUs
- Child processes are killable
- Avoid GIL limitations

- Relatively high overhead
  - → Open and close processes takes more time
- Sharing information between processes is very slow
- Model not adaptable to parallelism

Threads: pros and cons

- Processes produce threads (sub-processes) to handle sub tasks
  - ➡ Threads live inside the process and share the same memory space
- Can use shared memory
  - ➡ Threads communication
- Lightweight
- Very small overhead
- Great option for I/O bound application
- Subject to GIL (although there are workarounds)
- Not killable
- Potential of race condition
- Same memory space

When to use threads vs processes?

- **Processes** speed up Python operations that are CPU intensive because they benefit from multiple cores and avoid the GIL.
- **Threads** are best for IO tasks or tasks involving external systems because threads can combine their work more efficiently. Processes need to pickle their results to combine them which takes time.
  - ➡ Threads provide no benefit in python for CPU intensive tasks because of the GIL.

Things to be affraid of! (not only in python…)

- Starvation
  - ➡ a task is costantly denied necessary resource
  - ➡ The task can never finish (starves)
- Deadlock
  - ➡ Usually a deadlock occurs when two or more tasks wait cyclically for each other.