Threads

Concurrency: theoretical aspects

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Concurrency and parallelism

- Terminology
  - Concurrency
    - Multiple tasks (two or more) are performed in overlapping time intervals, without a specific order
    - Task execution seems contemporary (in parallel), but it is not
      - The effect is due to the CPU time-slicing, i.e., to the scheduler (context switching) that dedicates infinitesimal time units of the CPU to the various tasks

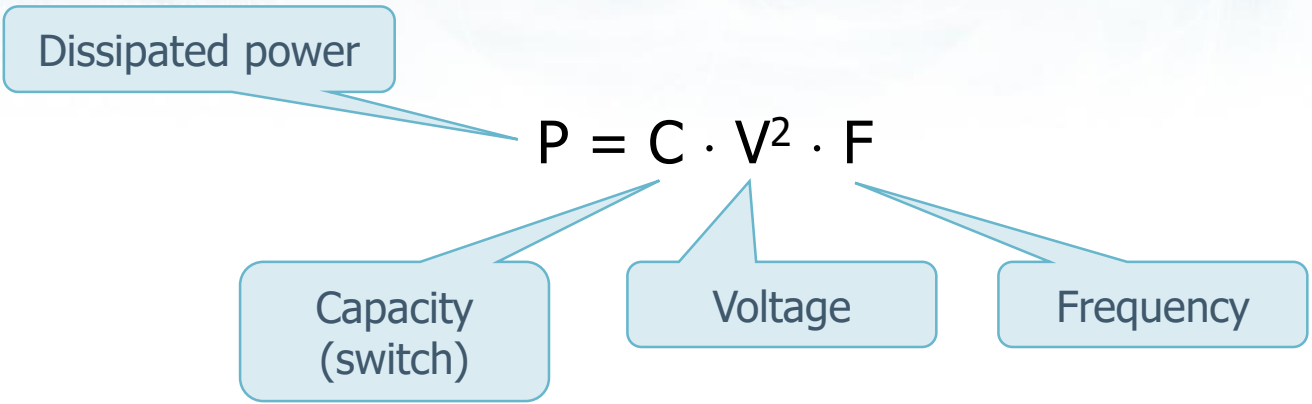
Concurrency and parallelism

- Parallelism
  - Multiple tasks (or different parts of the same task) are executed in the same time intervals in multi-processor or multi-core systems
  - It is allowed by dedicated hardware structures (multi-CPU or multi-core)
  - It is obtained by transforming a sequential execution flow into a parallel one

Concurrency and parallelism

- Concurrency and parallelism do not identify the same thing
  - Concurrency = manipulate many things at the same time
  - Parallelism = perform many things at the same time
  - These terms are often used with the same meaning
- These two concepts are old
  - But, much research has been done since 2004
  - In 2004, Intel stopped the development of Tejas and Jayhaw processors
    - The frequency increase (clock) of the processors has reached its limits due to excessive power consumption

Concurrency



- Due to the limits of frequency scaling
  - Parallel computing is now one of the main programming paradigms
  - Concurrent programming has introduced new challenges and pitfalls (bugs)

Concurrency and parallelism

- These changes have led to similar changes in the used paradigms
  - First Moore law (1965)
    - The number of transistors in the processors will double every 12 months
      - 24 months in the 80s
      - 18 months in the 90s
    - Example
      - May 1997, Pentium II, 7.5 millions of transistors, f_{CLOCK} = 300 MHz
      - November 2000, Pentium 4, 42 millions of transistors, f_{CLOCK} = 1.5 GHz

Concurrency and parallelism

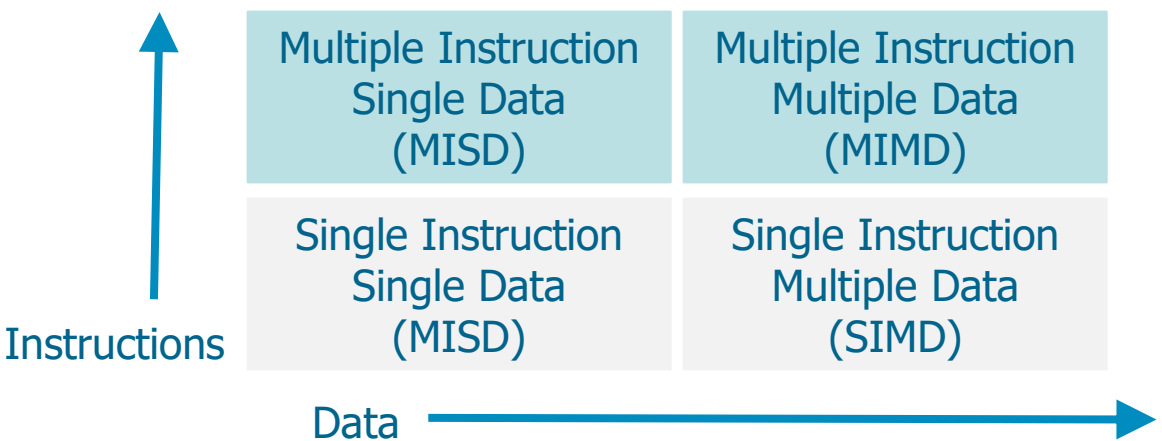
- Bill Dally law (NVIDIA, 2010, Forbes)
  - Following Moore's law no longer makes sense
  - We can increase the number of transistors and cores by 4 times every 3 years. By making each core working slightly more slowly, so more efficiently, we can increase performance more than triple, while maintaining the same total consumption

Parallel architectures

- There are different levels of parallelism
  - Bit-level
    - Word length determines the efficiency of an instruction (e.g., 8 bit versus 16 bit adder)
  - Instruction-level
    - Use of multi-stage pipelines for the execution of an instruction flow (e.g., fetch, decode, execute)
  - Task-level
    - Different computations are executed in parallel (e.g., sorting and matrix product executed in parallel)

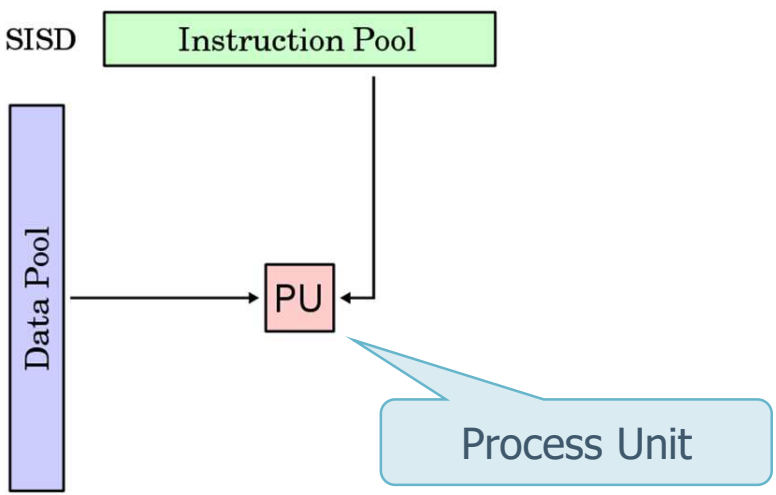
Parallel architectures

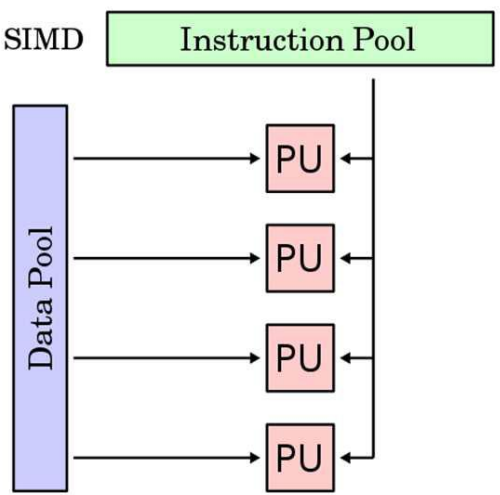
- The first classification for parallel architectures was introduced by Flynn (1966)
  - Partially outdated
    - Many architectures are mixed or not directly classifiable
  - Still widely used
    - Simple and easy to understand



Parallel architectures

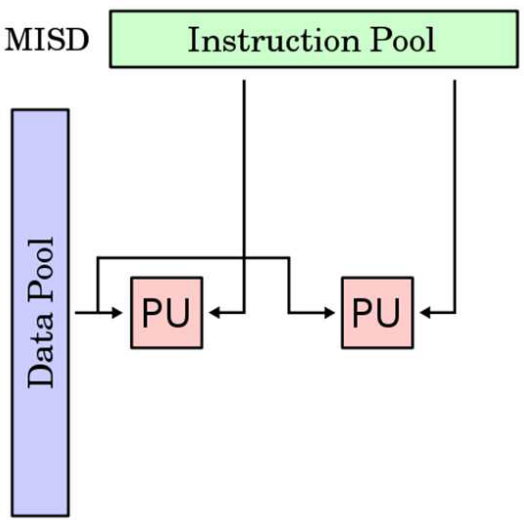
- Single Instruction Single Data (SISD)
  - Classical scheme
  - No parallelism
- Single Instruction Multiple Data (SIMD)
  - A single instruction operates on multiple data flows

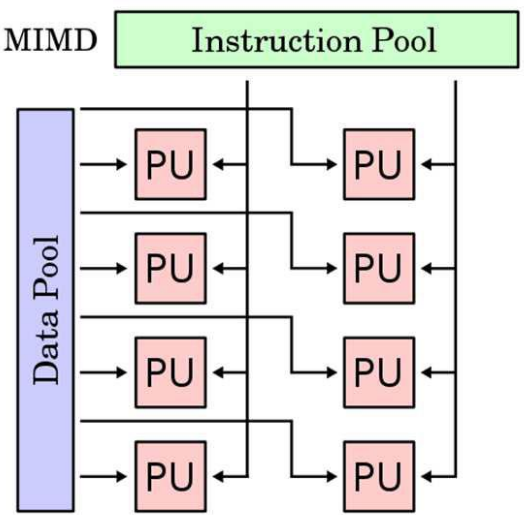




Parallel architectures

- Multiple Instruction Single Data (MISD)
  - Multiple instructions operate on a single data flow
- Multiple Instruction Multiple Data (MIMD)
  - Multiple instructions operate on a multiple data flow





Speed-up

- The main purpose of parallelism is to increase efficiency
  - How do we evaluate efficiency?
  - Possible evaluation
    - Time
    - Memory

Speed-up

- There are different metrics for calculating execution times
  - User time
    - Total time that the CPU uses to perform a given task at the user-level
      - It does not take into account the time "lost" in management, e.g., I/O operations, routine execution at the kernel level, etc.

Speed-up

- CPU time
  - Total time dedicated by the CPU to the execution of a task
    - This time includes I/O times, task management by the kernel, etc.
  - On a parallel architecture it will be necessary to evaluate the time dedicated to the task by all the calculation units
    - In most cases, this time will be longer than the time of the same sequential process running on a single processor

Speed-up

- Wall-clock time
  - Also called elapsed time
  - Actual time required to complete a given task
  - That is
    - Wall clock time = task finish time – task starting time
  - This is equivalent to compute the whole execution time regardless of whether the execution is carried out on single or multi-processor systems
  - Taking this metric as a reference
    - $speedup = \frac{Wall\text{-}clock\ time\ sequential\ algorithm}{Wall\text{-}clock\ time\ parallel\ algorithm}$

Speed-up

- Doubling the number of processing elements should halve the wall-clock time
  - Advantages should be linear
- This behavior is obtained
  - Rarely
    - If a process cannot be parallelized, increasing the number of processors/cores does not change the run time
  - Only for a small number of processors/cores
    - After an initial linear dependence, the speed-up curve has a horizontal asymptote

Amdhal law

- The theoretical advantages obtained by concurrency were first analyzed by Amdhal
- The Amdhal law (1967) specifies the theoretical improvement obtainable by parallelism
  - $speedup = \frac{1}{S+\frac{1-S}{n}}$

S = percentage of the elapsed time for the execution of the sequential part of a program (not parallelizable)

n = number of processors or cores

Amdhal law

- To Ahmdal law should be added the overhead related to the management of n threads (or processes)
  - $speedup = \frac{1}{S+\frac{1-S}{n}+H(n)}$

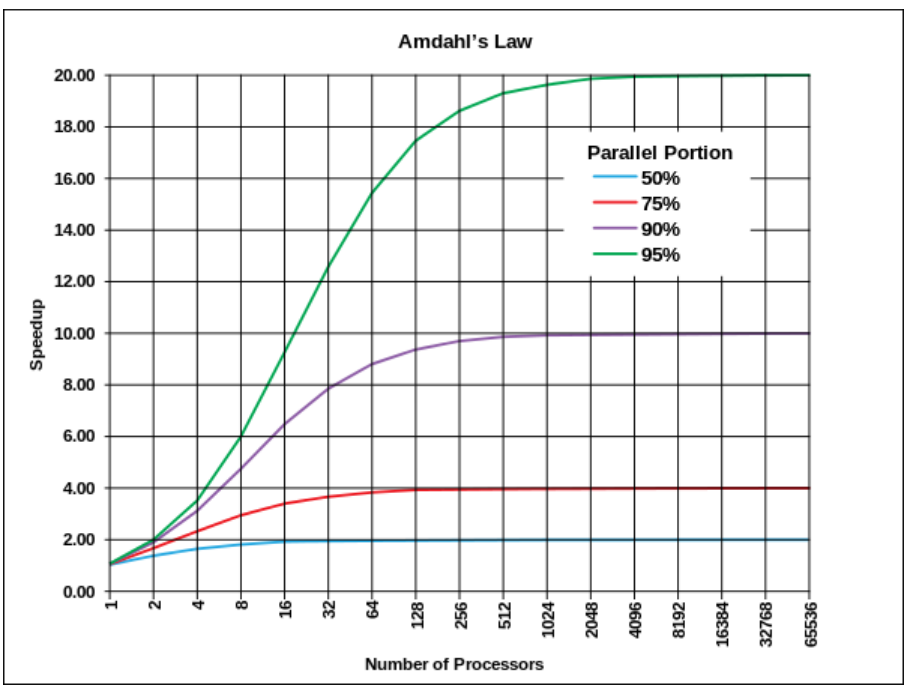
H(n) management overhead: of the operating system and inter-thread synchronization

- In the most optimistic hypothesis H(n)=0, and when the number of processors increases
  - $speedup = \lim_{n\to\infty} \frac{1}{S+\frac{1-S}{n}} = \frac{1}{S}$

S = 10% → $speedup_{max}$ = 10
S = 20% → $speedup_{max}$ = 5
...
S = 50% → $speedup_{max}$ = 2
...

Amdhal law

- Amdahl law (1967)
  - Small program segments intrinsically sequential limit the total speed-up that can be obtained



Amdhal law

- Corollary to Amdhal law
  - Decreasing the serializable part and increasing the parallelizable part is more important than increasing the number of processors
- Following the Amhdal law, the use of parallelism has been subjected to criticism for many years

Amdhal law

- Limits of the Amhdal law
  - The limits do not only depend on the availability of CPU cycles, but also on other factors
    - Multi-core systems can have multiple caches for lowering memory latency and increasing system efficiency
    - Some algorithms have better parallel formulations, or with a smaller number of execution steps
    - Amdahl assumes that the dimension of the problems remains constant, while in general the dimension increases with the increase of the available resources, and what remains constant is the execution time

Gustafson law

- In the 1980s, linear speed-ups with 1024 processors were obtained at Sandia National Labs on applications on which Amhdal would have expected non-linear behavior
- The Gustafson law (or Barsis equation) expects a linear speed-up
  - $speedup = n + (1 - n) \cdot S$

n = number of processors or cores
S = fraction of time in the sequential part

Parallelization

- The parallelization of an algorithm can be performed only with decomposition
  - Task decomposition
    - Program decomposition into functions, and analysis of which functions can execute in parallel
  - Data decomposition
    - Decomposition of the problem depending on the parallelism applicable to the data, rather than its functional/logical nature
  - Data flow decomposition
    - Decomposition of the problem according to the flow of data between the various functions or tasks to be completed

Parallelization

- The decomposition of a problem can only be done by knowing its dependencies
- The precedence constraints can be represented with a **precedence graph**

Relationship with Control Flow Graph (CFG) and Process Generation Trees

Precedence graph

- A precedence graph is a direct acyclic graph, in which
  - The vertices correspond to single instructions, blocks of instructions, processes
  - The arcs correspond to precedence conditions
    - An arc from vertex A to vertex B means that task B can only be executed after task A has been completed
  - Precedence can be imposed through synchronization techniques
    - Synchronization = mechanism used to impose constraints on the order of execution of processing units (processes or threads)

Precedence graph