POLITECNICO DI MILANO

Parallel Algorithms and Parallel Programming – Introduction

095946 - Advanced Algorithms and Parallel Programming

Fabrizio Ferrandi

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
fabrizio.ferrandi@polimi.it

Outline of this Lecture

- Motivation
- Automatic Parallelization vs. Parallelization by hand
- Types of parallelism
- Examples of parallel programming infrastructures

Introduction

- Traditionally, people (are used to) think sequential
  - Developers (are used to) think sequential
  - Most of the existing algorithms are sequential

BUT

- Modern architectures offer an high degree of parallelism: they can execute different instructions/tasks at the same time

Parallel is better

- **Time saving**: parallel algorithms can be more performant than sequential ones (i.e., take less time)
- **Money saving**: a parallel architecture composed of cheap components can be less expensive than a single processor architecture composed of a costly processor
- **Solve «Grand Challenge» problems**, i.e., problems that in practice can be solved only exploiting parallelism because of their complexity

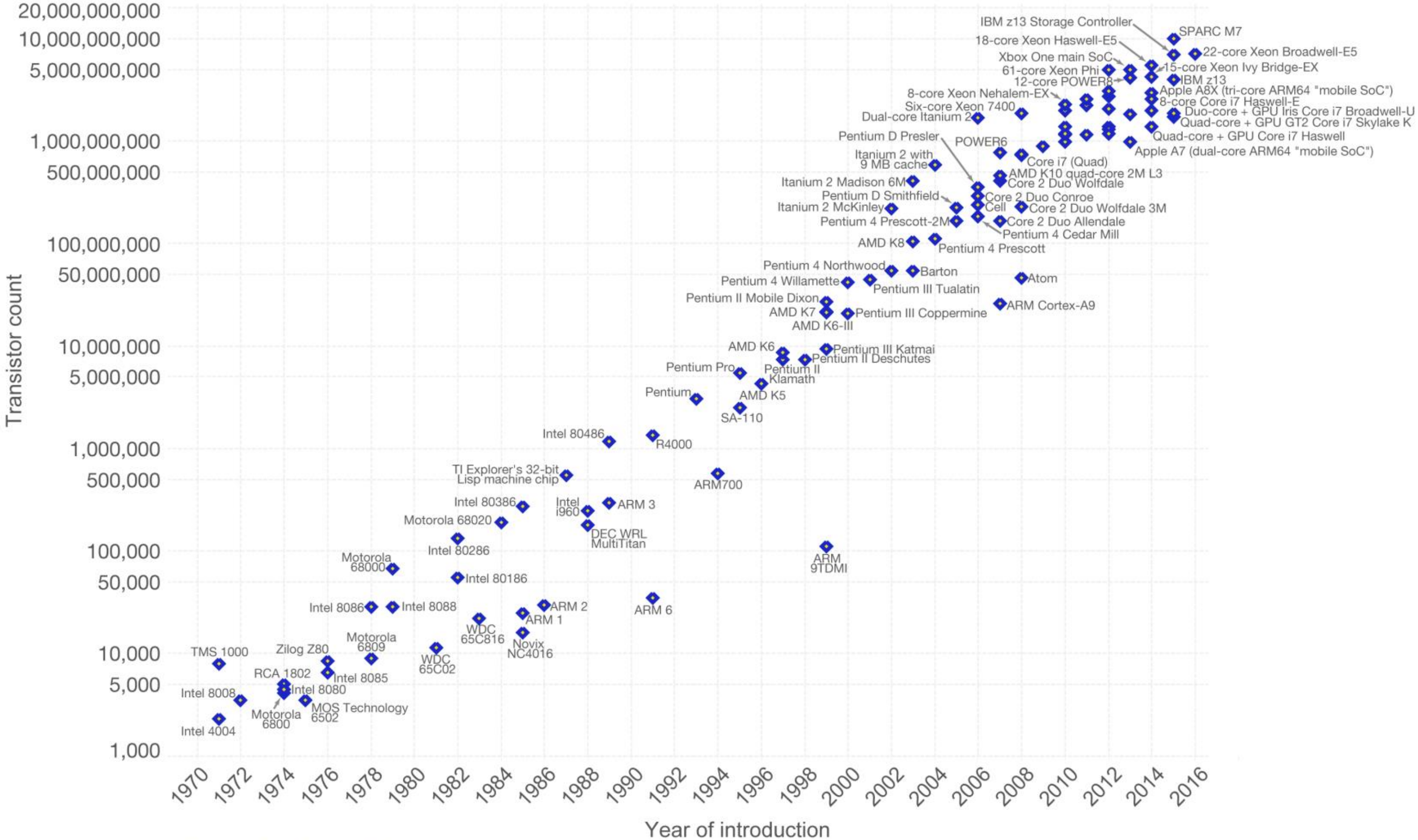
Why nowadays has it become so important?

Moore’s law

Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

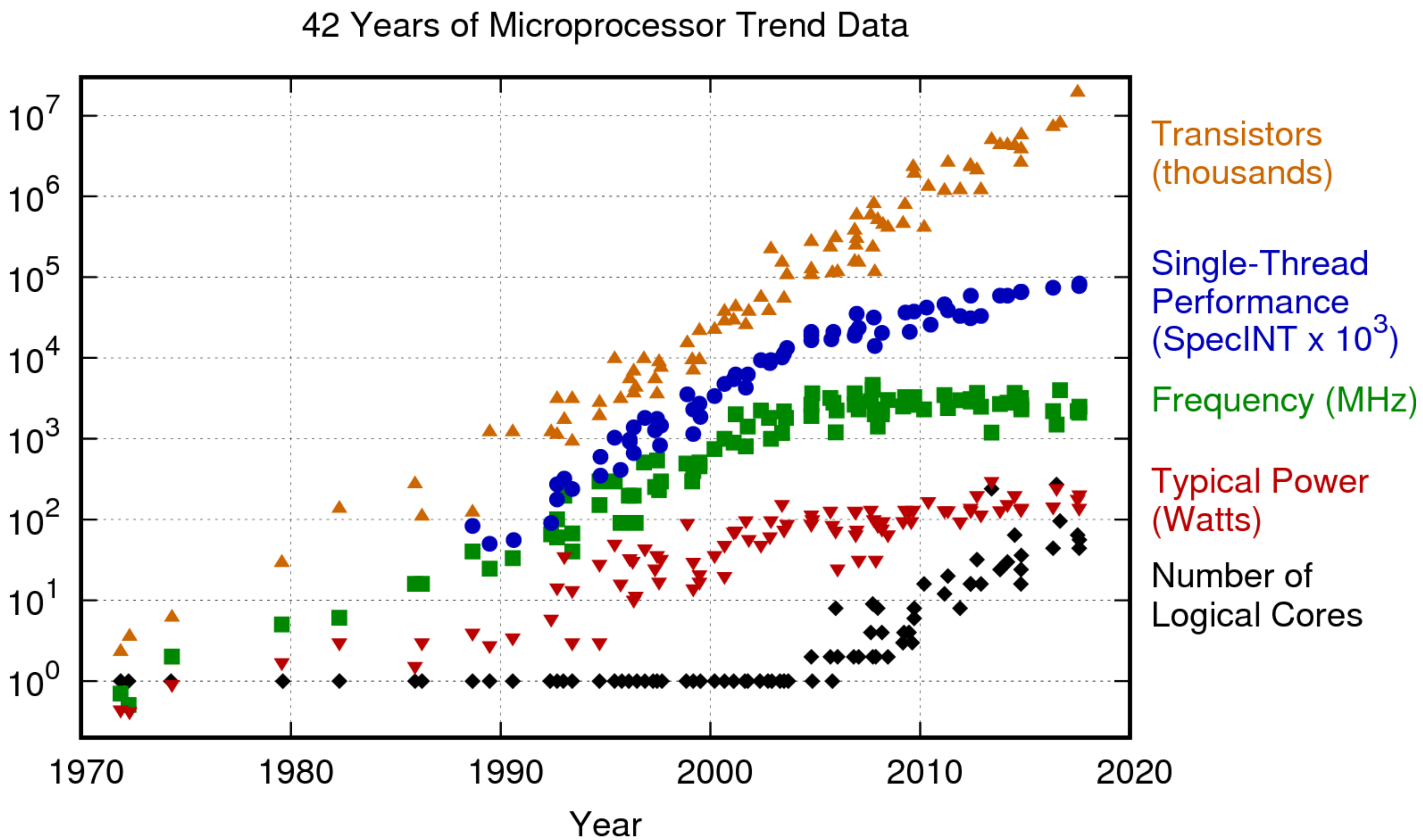
Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Moore’s law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Limits in improvement of single cores

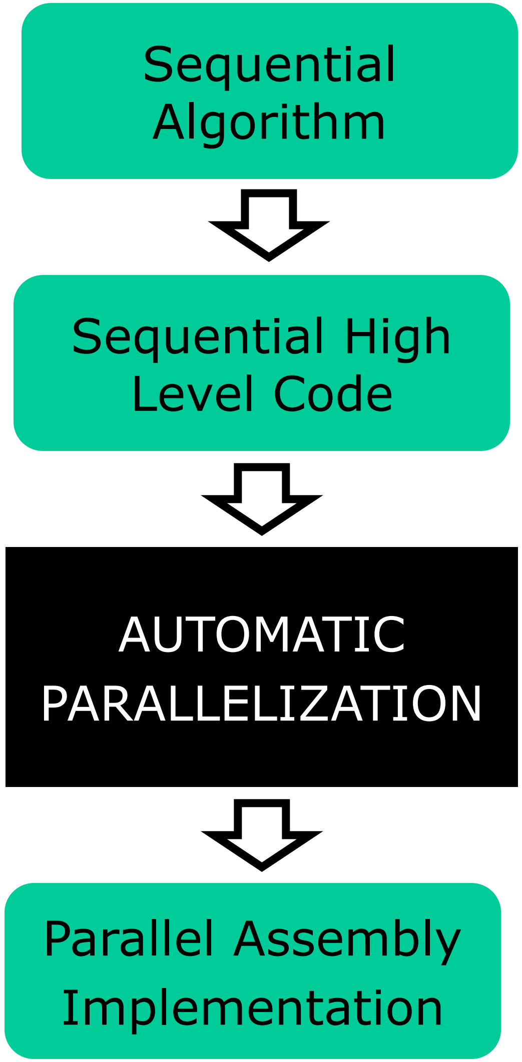
- Performance increasing of single cores is **slowing**
- Moore's law:

  The number of transistors incorporated in a chip will approximately double every 24 months
- But a single core **can not** exploit anymore all these transistors
- Continue **increasing** the frequency of processors is not anymore possible because of **power consumption**

Multiple processing units

- Multi cores in the same CPU chip
  - Intel i9 10980XE – 18 cores - ~5-10 TFlop/s
  - Intel Xeon PHI 7290 – 72 cores – 3 TFlop/s
- Multi cores in the same GPU chip
  - Nvidia Hopper H100 – 16896 cores – 30-2000 TFlop/s
- Multi computer (clusters)
  - **Frontier** – HP
    9,472 AMD Epyc 7A53s "Trento" 64 core 2 GHz CPUs (606,208 cores) 37,888 Radeon Instinct MI250X GPUs (8,335,360 cores). 1.102 exaFLOPS (Rmax) / 1.685 exaFLOPS (Rpeak)

Automatic Parallelization



- Write sequential algorithm
- Implement with sequential code
- Leave all the parallelization work to automatic tools

Automatic Parallelization fails


```
void and_or(int SIZE, int * and, int * or, int * b,
int * c){
    int i = 0;
    for(i=0; i<SIZE; i++)
        and[i] = b[i] & c[i];
    for(i=0; i<SIZE; i++)
        or[i] = b[i] | c[i];
}
```

Automatic Parallelization fails

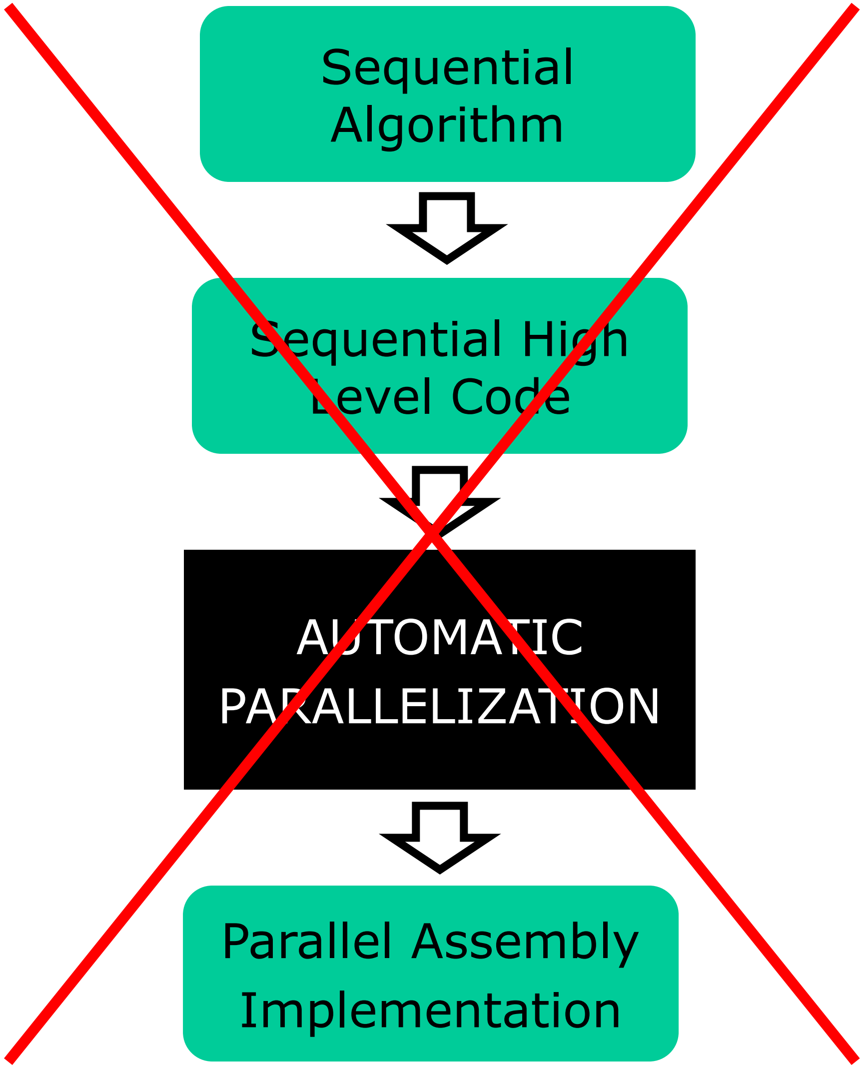

```
void and_or(int SIZE, int * and, int * or, int * b,
int * c){
    int i = 0;
    for(i=0; i<SIZE; i++)
        and[i] = b[i] & c[i];
    for(i=0; i<SIZE; i++)
        or[i] = b[i] | c[i];
}
```

- This code fragment can not be automatically parallelized as is:
- `and, or, b, c` can overlap:

```
int array_a[10], array_b[10];
...
and_or(9, &array_a[1], &array_a[1], array_b, array_a);
```

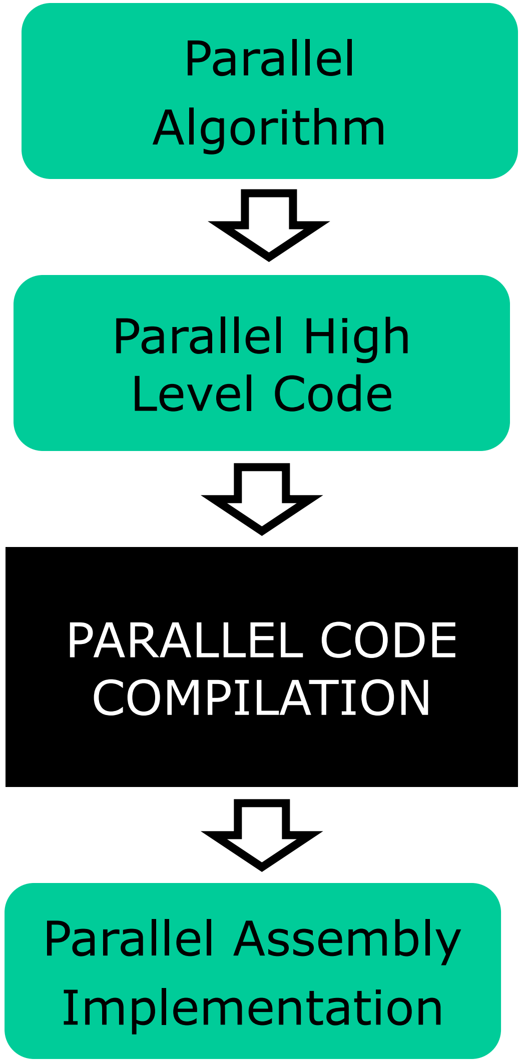
- The designer maybe will never use the function in this way, but the compiler does not know...

Automatic Parallelization



- Complete automatic parallelization is (at the moment?) **not feasible**
- Tools are not able to extract all the available parallelism from a specification designed to be executed in **sequential way**

Parallelization by hand



- The programmer needs to give **hints** to the tools
- Write **parallel algorithms**
- Implement with high level parallel code
- Leave only code compilation to the tools

Parallelization by hand

- There are three critical aspects:
  - Which **type** of parallelism has to be considered
  - How to **design** the parallel algorithm
    - Trying to parallelize existing sequential algorithms
    - From scratch
  - How to **provide information** about the parallelism to the tools

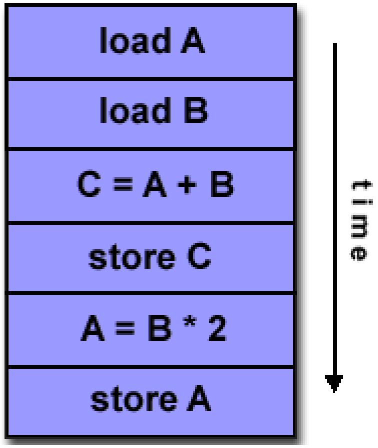
Types of Parallelism – 1
Flynn's Classical Taxonomy

- There is not a single kind of parallelism:
  - Instruction Parallelism
  - Data Parallelism
- They can be combined: Flynn's Taxonomy
- Proposed in 1966 to classify computer architectures, but it can describe types of parallelism in general

S.I.S.D. (SINGLE INSTRUCTION SINGLE DATA)	S.I.M.D. (SINGLE INSTRUCTION MULTIPLE DATA)
M.I.S.D. (MULTIPLE INSTRUCTION SINGLE DATA)	M.I.M.D (MULTIPLE INSTRUCTION MULTIPLE DATA)

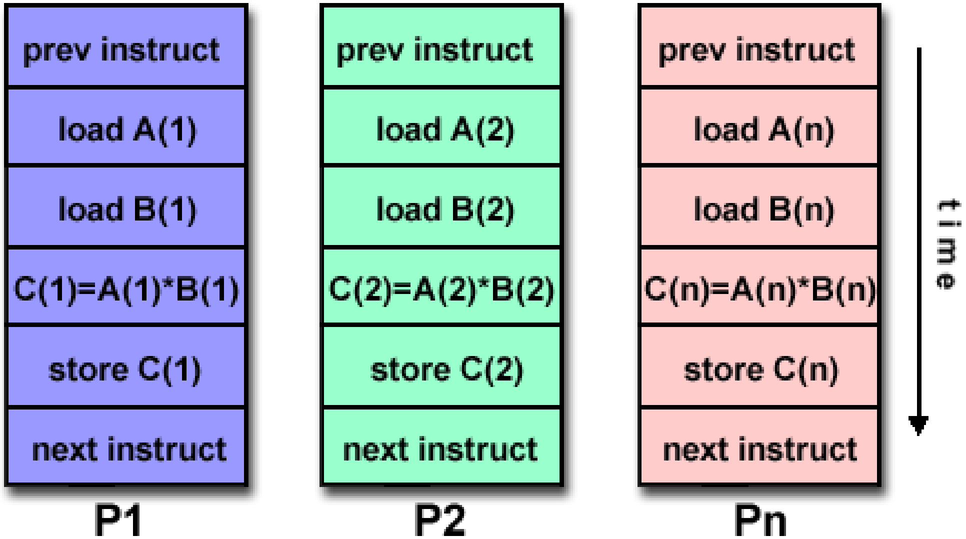
Single Instruction Single Data

- This is the sequential case:
- Single instruction:
  - CPU processes single instruction stream
- Single data:
  - A single data input stream
- Deterministic execution
- Examples:
  - All the single core architectures



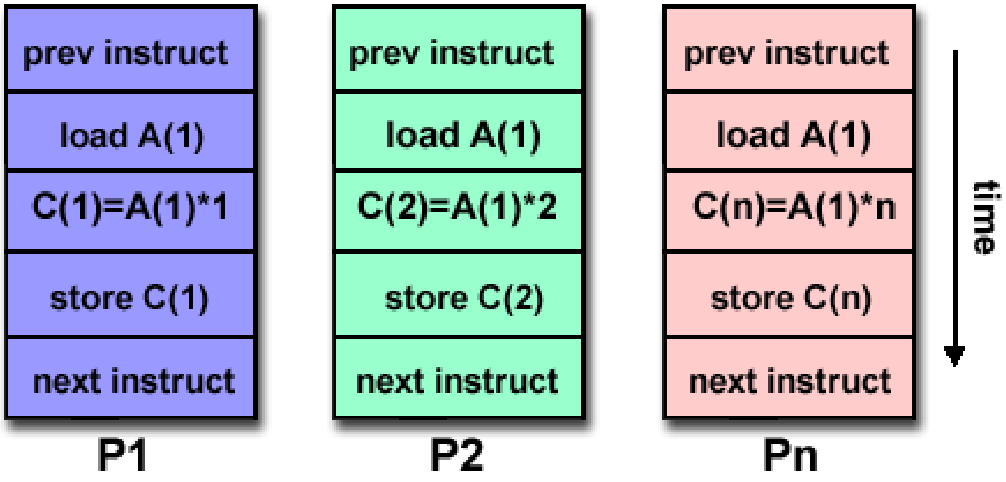
Single Instruction Multiple Data

- Single instruction:
  - All the cores execute the same instruction in the same clock cycles
- Multiple data:
  - Each core elaborate different data
- Synchronous and deterministic execution
- Examples:
  - most of the modern GPUs



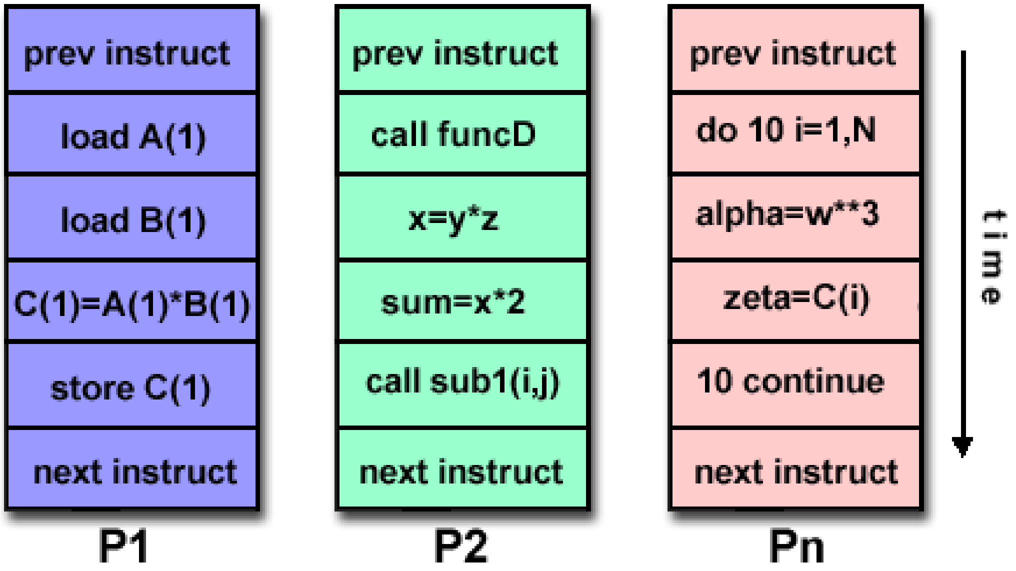
Multiple Instruction Single Data

- Multiple Instruction
  - Each core process the data with different instructions
- Single Data
  - A single data stream is fed into multiple processing units
- Examples:
  - Experimental architectures
  - Any multicore architecture if we relax synchronization



Multiple Instruction Multiple Data

- Multiple Instruction
  - Each core executes different instructions
- Multiple data
  - Each core processes different data
- Execution can be synchronous or asynchronous, deterministic or not
- Examples:
  - multicores



Types of Parallelism – 2
Level of Parallelism

- Bits
- Instructions
- Tasks

Bit Level Parallelism

- Bits composing words represent different data
- A single instruction can manipulate different data at a time
- It is very relevant in Hardware Implementation of algorithm
- It can become significant also in software implementation (e.g., representing set of elements as strings of bits)

Instruction Level Parallelism

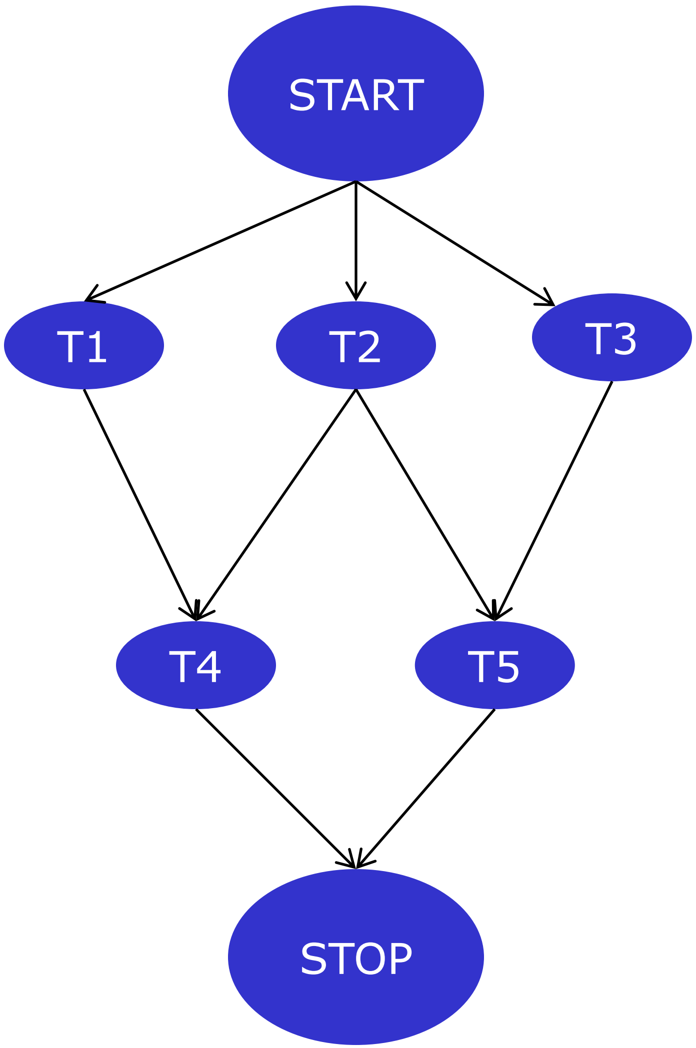
- Different instructions executed at the same time on the same core
- Supported by multiple execution units, pipeline, vector, SIMD units etc.
- This type of parallelism can be easily extracted by compilers

Task Level Parallelism

- Task: a logically discrete section of computational work
  - Typically a program or program-like set of instructions that is executed by a processor
- Parallel program
  - Multiple tasks running on multiple processors
- Supported by shared memory, cache coherence mechanisms
- Usually difficult to be automatically extracted

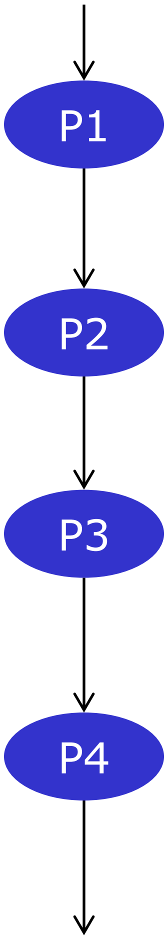
Task Level Parallelism: Parallel Task Graph

T1,T2,T3 are running in parallel



- Vertices correspond to tasks
- Edges represent precedencies or data communications
- Each task is executed once (in principle)

Task Level Parallelism: Pipeline for parallelizing



- Like processor pipeline
- Edges represent data passing
- Suitable to parallelize streaming elaboration such as audio and video encoding
- Each task is executed at each stage

Task Level Parallelism: Communication

- Two main models for communication
- **Shared memory**:
  - All the processors, so all the tasks, share a global memory with the same address space
  - Modifications in a memory location performed by a processor are seen by all the other processors
- **Message Passing**:
  - Each task has its private memory
  - Tasks communicate by explicitly sending and receiving messages

Example of different levels of parallelism


```
void and_or(int SIZE, int * restrict and, int * restrict
or, int * restrict b, int * restrict c){
   int i = 0;
   for(i=0; i<SIZE; i++)
      and[i] = b[i] & c[i];
   for(i=0; i<SIZE; i++)
      or[i] = b[i] | c[i];
}
```

- If memory locations do not overlap (`restrict`) we can extract parallelism at different levels:
  - **Task** (inter-loops): the two loops can be run in parallel

    because they're going to write in different memory portion -> the 2 loops are indipendent so they can be parallelized.
  - **Instruction** (intra-loops): each iteration of the loop can run in parallel
  - **Bit** (intra-operation): each bit can be computed in parallel

    bit operation in 2 different loops.

Design parallel algorithms

- Design a «good» parallel algorithm by extracting all the available parallelism is **not enough**
  - not all the extracted parallelism is exploitable on a real architecture
- We need to consider which parallelism is available on the considered architecture
  - Non suitable parallelism can introduce **overhead**
- We need to «describe» the parallelism to the compilation tools to make it exploitable

Sequential Programming

- Translation from pseudo-code to high level source code (e.g., C/C++) is usually quite trivial
- All the sequential machine can be modeled as a von Neumann Architecture
- Real processors can differ a lot, but compilers are usually able to manage this gap and optimize the application for a given architecture
- Example:
  - Intel 80386 and Intel i9 processors are quite different, but you can use the same source code to create an optimized application for them

Parallel Programming

- New programming languages (mainly developed for research activity):
  - + Introduced since parallel programming is a different paradigm
  - - did not have big success:
    - too immature compilers
    - you need to learn a new language
- Extensions to existing programming language
  - + can be easily adopted by designer
  - + can be easily integrated in existing compilers
  - - can describe only some types of parallelism (e.g., pipeline parallelism is difficult to be described)

List of Parallel Programming Languages (from Wikipedia)

- Actor model: Axum, Elixir, Erlang, Janus, Red, SALSA, Scala/Akka, Smalltalk, Akka.NET
- Coordination languages: CnC, Glenda, Linda, coordination language, Millipede
- Dataflow programming: CAL, E, Joule, LabView, Lustre, Preesm, Signal, SISAL, BMDFM
- Distributed computing: Bloom, Hermes, Julia, Limbo, MPD, Oz, Sequoia, SR
- Event-driven and hardware description: Esterel, SystemC, SystemVerilog, Verilog, Verilog-AMS, VHDL
- Functional programming: Clojure, Concurrent ML, Elixir, Erlang, Futhark, Haskell, Id, MultiLisp, SequenceL, Elm
- Logic programming: Parlog, Prolog
- Monitor-based: Concurrent Pascal
- Multi-threaded: C=, Cilk, Cilk Plus, C#, Clojure, Fork, Java, ParaSail, Rust, SequenceL

List of Parallel Programming Languages (from Wikipedia)

- Object-oriented programming: μC++, Ada, C*, C#, C++ AMP, Charm++, D Programming Language, Eiffel SCOOP, Emerald, Java, Join Java, ParaSail, Smalltalk
- Partitioned Global Address Space (PGAS): Chapel, Coarray Fortran, Fortress, High Performance Fortran, Titanium, Unified Parallel C, X10, ZPL
- Message passing: Ateji, Rust
- Communicating Sequential Processing: JCSP, Alef, Ease, FortranM, Go, JoCaml, Joyce, Limbo, Newsqueak, Occam, Occam-π, PyCSP, SuperPascal, XC
- APIs/Frameworks: Apache Hadoop, Apache Spark, Apache Flink, Apache Beam, CUDA, OpenCL, OpenHMPP, OpenMP

Why so many Parallel Programming Languages?

- Translation from pseudo-code to high level source code (e.g., C/C++) can be an hard task
- Parallel architecture are composed of equivalent von Neumann Machines, but
- Real architectures differ so much that now compilers are not able to fill the gap between an abstract model and real implementation
  - Optimized applications can not be generated starting from generic parallel code
  - Code extensions have been specialized for particular types of applications/architectures

Evolution of Parallel Programming

TECHNOLOGY	TYPE	YEAR
Verilog/VHDL	Languages	1984/1987
MPI	Library	1994
PThread	Library	1995
OpenMP	C/Fortran Extensions	1997
CUDA	C Extensions	2007
OpenCL	C/C++ Extensions + API	2008
Apache Spark	API	2014

- Message passing and threads are technologies older than standards that defined them
- New technologies directly introduced as standard

Authors and Developers

TECHNOLOGY	AUTHORS	DEVELOPERS
Verilog/VHDL	Phil Moorby – Prabhu Goel / United States Department of Defence	Accellera Systems Initiative / IEEE VHDL Analysis and Standardization Group
MPI	MPI Forum	MPI Forum
PThread	IEEE Technical Committee on Operating System	Austin Group
OpenMP	OpenMP Architecture Review Board	OpenMP Architecture Review Board
CUDA	NVIDIA	NVIDIA
OpenCL	Apple	Khronos Group
Apache Spark	Berkeley	Apache Foundation Databricks

All technologies except CUDA are developed by consortiums/Foundation

Level of Parallelism

TECHNOLOGY	Bit	Instruction	Task
Verilog/VHDL	Yes	Yes	No
MPI	(Yes)	(Yes)	Yes
PThread	(Yes)	(Yes)	Yes
OpenMP	(Yes)	(Yes)	Yes
CUDA	(Yes)	No	(Yes)
OpenCL	(Yes)	No	Yes
Apache Spark	(Yes)	No	(Yes)

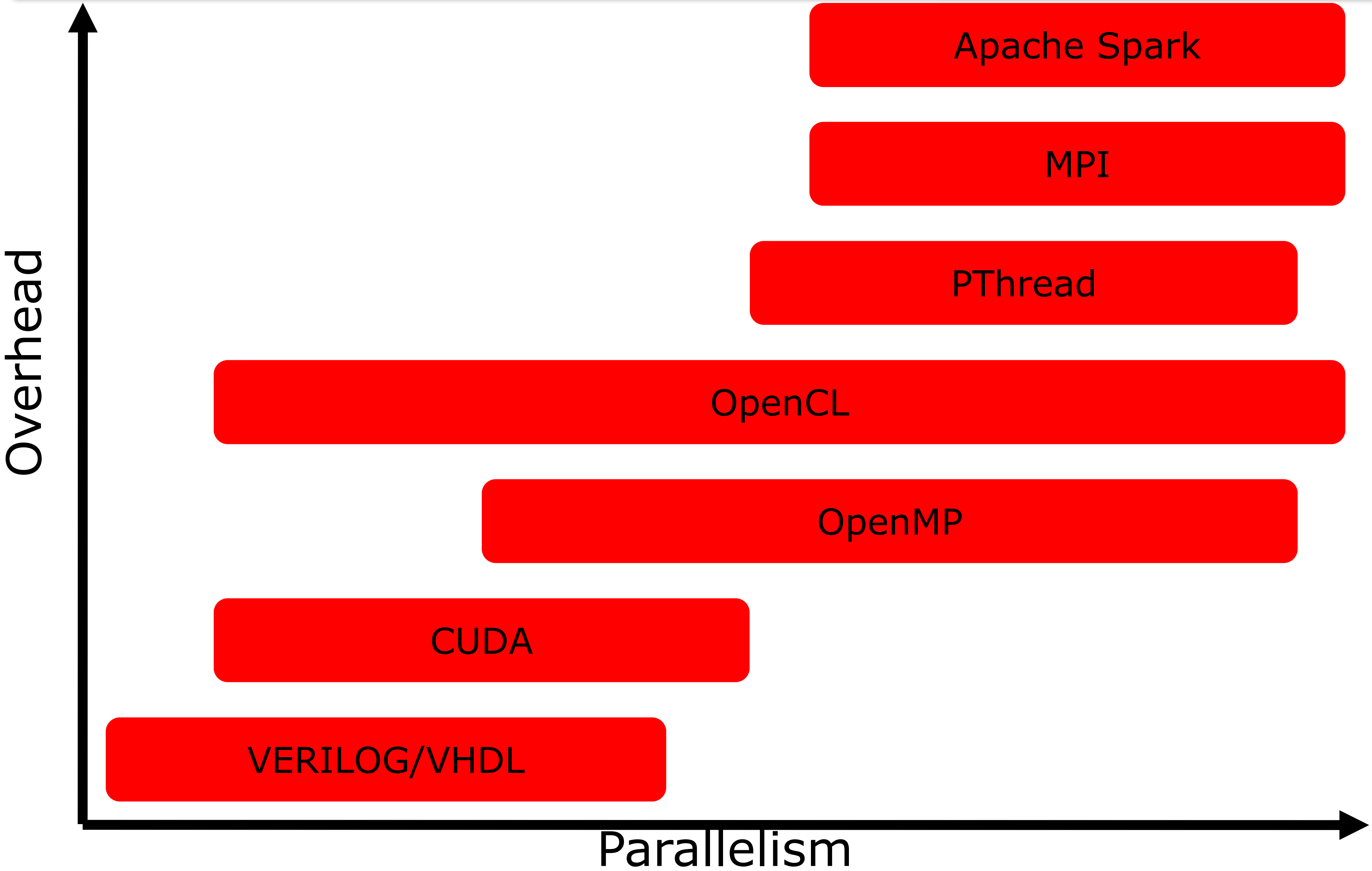
- Bit level parallelism can be exploited on a single core by means of bit-level operators
- Instruction level parallelism can be exploited through compilers on superscalar cores

Type of Parallelism

TECHNOLOGY	SIMD	MISD	MIMD
Verilog/VHDL	Yes	Yes	Yes
MPI	Yes	Yes	Yes
PThread	Yes	(Yes)	Yes
OpenMP	Yes	Yes	Yes
CUDA	Yes	No	(Yes)
OpenCL	Yes	(Yes)	Yes
Apache Spark	Yes	No	No

- MISD in PThread and OpenCL can be obtained exploiting MIMD constructs
- MIMD parallelism in CUDA can be exploited as CPU-GPU parallelism and multiple kernel running simultaneously in streams

Parallelism granularity



Target architectures

TECHNOLOGY	Processors	Memory
Verilog/VHDL	ASIC – FPGA	
MPI	Multi CPUs	(Mainly) Distributed Memory
PThread	Multi-core CPU	(Mainly) Shared Memory
OpenMP	Multi-core CPU	(Mainly) Shared Memory
CUDA	CPU + GPU(s)	(Distributed) Shared Memory
OpenCL	Heterogeneous Architecture	Both distributed and shared memory
Apache Spark	Multi CPUs	Distributed Memory

CUDA memory:

- Distributed between CPU and GPU
- Shared on the GPU

Description of Parallelization and Communication

TECHNOLOGY	Parallelism	Communication
Verilog/VHDL	Explicit	Explicit
MPI	Implicit	Explicit
PThread	Explicit	Implicit
OpenMP	Explicit	Implicit
CUDA	Implicit(Explicit)	Implicit(Explicit)
OpenCL	Explicit/Implicit	Explicit/Implicit
Apache Spark	Implicit	Implicit

- CUDA and OpenCL:
  - Different kernels-CPU code: explicit parallelism and communication
  - Threads of the same kernel: implicit parallelism and communication
- In OpenCL explicit parallelism can be used also in a single kernel

Programming support

TECHNOLOGY	Target Independent Code?	Development Platforms
Verilog/VHDL	Yes (behavioral) No (structural)	Mainly Linux
MPI	Yes	All
PThread	Yes	All – Windows through a wrapper
OpenMP	Yes	All – Different compilers
CUDA	Depend on CUDA capabilities	All
OpenCL	Yes	All – Different compilers
Apache Spark	Yes	Mainly Linux

- Performance optimization usually requires knowledge of the target architecture

Programming support

TECHNOLOGY	Compilation tool chain	Runtime environment
Verilog/VHDL	Provided by HW vendor	
MPI	Generic compilers / ad-hoc compilers	Support for multi-processes applications
PThread	Generic compilers	Support for multi-threading applications
OpenMP	Compiler with OpenMP support	Depends on OpenMP implementation
CUDA	NVIDIA Toolkit	Based on NVIDIA drivers
OpenCL	Ad-hoc compilers	Depends on the target architecture
Apache Spark	Interpreter/Runtime compilation	Complex runtime system

Pros and Cons: Verilog/VHDL

- **Pros:**
  - Complete control on computation and memory
  - No overhead introduced in the computation
  - Provides access to potentially large computational power
- **Cons:**
  - Requires specific Hardware (e.g., ASIC or FPGA) to implement functionality
  - Difficult to learn: completely different programming language and programming paradigm
  - Depends on the chosen target architecture

Pros and Cons: MPI

- **Pros:**
  - Can be adopted on different types of architectures
  - Scalable solutions
  - Synchronization and data communication are explicitly managed
- **Cons:**
  - Communication can introduce significant overhead
  - Programming paradigm more difficult than shared memory based ones
  - Standard does not reflect immediately advances in architecture characteristics

Pros and Cons: PThread

- **Pros:**
  - Can be adopted on different architectures
  - Explicit parallelism and full control over application
- **Cons:**
  - Task management overhead can be significant
  - Not easily scalable solutions
  - Low level API

Pros and Cons: OpenMP

- **Pros:**
  - Easy to learn
  - Scalable solution
  - Parallel applications can also be executed sequentially
- **Cons:**
  - Mainly focused on shared memory homogeneous systems
  - Require small interaction between tasks

Pros and Cons: CUDA

- Pros:
  - Provides access to the computational power of GPUs
  - Writing a CUDA kernel is quite easy
  - Already optimized libraries
- Cons:
  - Targets only NVIDIA GPUs
  - Difficult to extract massive parallelism from application
  - Difficult to optimize CUDA kernel

Pros and Cons: OpenCL

- **Pros:**
  - Target-independent standard
  - Hides architecture details
  - Same programming infrastructure for very heterogeneous architecture: CPU + GPU (+FPGA)
- **Cons:**
  - Difficult programming paradigm for its heterogeneity
  - Hiding of architecture details makes difficult to obtain best performances

Pros and Cons: Apache Spark

- **Pros:**
  - API for different languages
  - Explicit parallelization and communication are not required
  - Preinstalled on cloud provider VMs
- **Cons:**
  - Suitable only for big data applications
  - Does not (yet) fully support GPUs

Mix of parallelism technologies: OpenMP + CUDA

- Allows to exploit Multi-core CPU and GPU
- CUDA is used to parallelize GPU code
- OpenMP is used to parallelize CPU code

Mix of parallelism technologies: MPI + OpenMP

- First scenario:
  - MPI used to express coarser parallelism (Multi CPU)
  - OpenMP used to express finer parallelism (Multi core)
- Second scenario:
  - MPI used to implement communications
  - OpenMP used to parallelize computation

Mix of parallelism technologies: OpenCL + Verilog/VHDL

- In principle, hardware kernels (implemented for example on FPGA) can be used as accelerators
- OpenCL used to describe parallelism among different processing elements
- Verilog/VHDL used to describe hardware kernel
- Example of target: Intel Xeon Scalable (Skylake + Arria 10 FPGA)

Domain specific languages: Halide

- Separation of the algorithm being implemented from its execution schedule
  - Execution schedule:
    - loop nesting, parallelization, loop unrolling and vector instruction.
    - allows the programmer to experiment with scheduling and finding the most efficient one.
- Proposed for image processing first by MIT/Stanford/Google
- Now it is used in deep learning applications

References

- Material taken from different sources:
  - Marco Lattuada material
  - two tutorials by Blaise Barney from the Lawrence Livermore National Laboratory and from slides of prof. Lanzi and Ing. Loiacono (Algoritmi e Calcolo Parallelo)
  - Introduction to Parallel Computing
    Blaise Barney, Lawrence Livermore National Laboratory
    https://computing.llnl.gov/tutorials/parallel_comp/
  - Also available as Dr.Dobb's "Go Parallel"
    Introduction to Parallel Computing: Part 2
    Blaise Barney, Lawrence Livermore National Laboratory
  - "Structured Parallel Programming: Patterns for Efficient Computation," Michael McCool,
    Arch Robinson, James Reinders,
    1st edition, Morgan Kaufmann,
    ISBN: 978-0-12-415993-8, 2012