PaPro : Parallel Programming

Houssam-Eddine Zahaf
houssameddine.zahaf@univ-nantes.fr

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems
	4h	Practice Pthreads and OpenMP

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems
	4h	Practice Pthreads and OpenMP
4h		Parallel programming : distributed memory systems

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems
	4h	Practice Pthreads and OpenMP
4h		Parallel programming : distributed memory systems
	4h	Practice OpenMPI

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems
	4h	Practice Pthreads and OpenMP
4h		Parallel programming : distributed memory systems
	4h	Practice OpenMPI
6h		Parallel programming : massively parallel hardware (GPUS)

This lecture

Lecture in hours

- Theoretical : 18h
- Practical : 12h
- Exam : 1h20

Lecture content :

Theo.	Prac.	Title
4h		Introduction to parallel programming
4h		Parallel Programming : shared memory systems
	4h	Practice Pthreads and OpenMP
4h		Parallel programming : distributed memory systems
	4h	Practice OpenMPI
6h		Parallel programming : massively parallel hardware (GPUS)
	4h	Practice CUDA and OpenCL

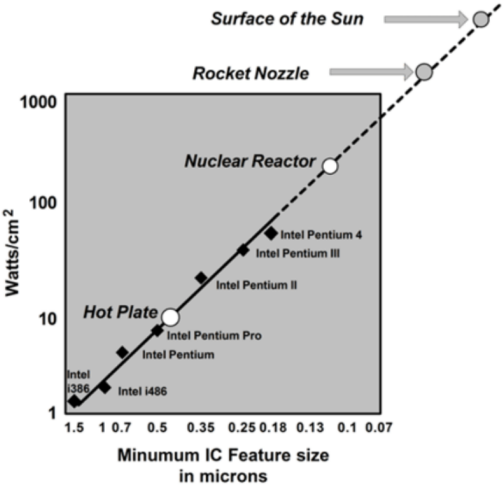
Plan

1. Sequential, Concurrent and PaPro
2. Hardware architecture for parallel programming
3. Sources of parallelisms
4. Parallelism efficiency
5. Practice preparation

Plan

1. **Sequential, Concurrent and PaPro**
2. Hardware architecture for parallel programming
3. Sources of parallelisms
4. Parallelism efficiency
5. Practice preparation

Single core limitations



Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

- Parallel programming: multiple parts of program execute at once:

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

- Parallel programming: multiple parts of program execute at once:

Rough definitions

- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

- Parallel programming: multiple parts of program execute at once:

- In last both cases, we achieve the same functionality (task)

Rough definitions

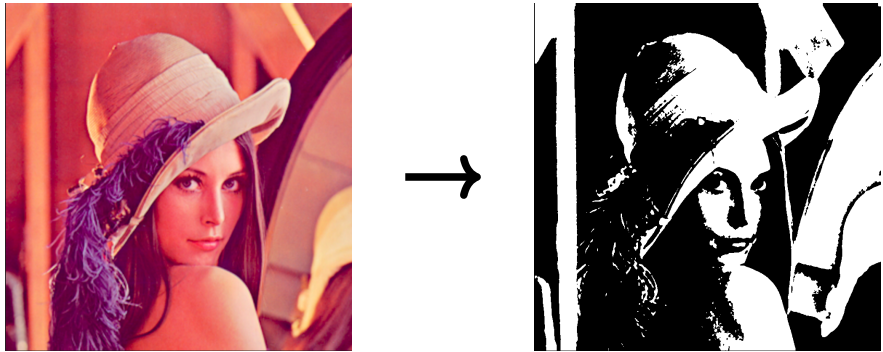
- Concurrent: Multiple tasks compete for the same resources
  - → Allows responsiveness

- Sequential programming: at every point in time, one part of program executing:

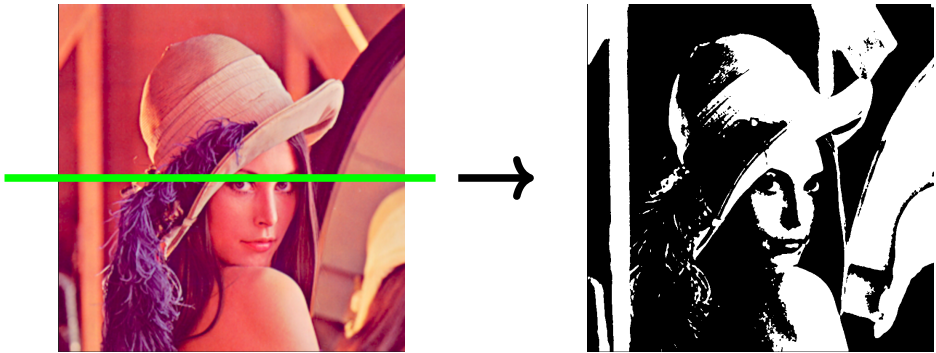
- Parallel programming: multiple parts of program execute at once:

- In last both cases, we achieve the same functionality (task)
- Objective : accelerate task execution, *i.e.* shorten task response time

Example (1) of a parallel program : binarization



Example (1) of a parallel program : binarization



Example (1) of a parallel program : binarization

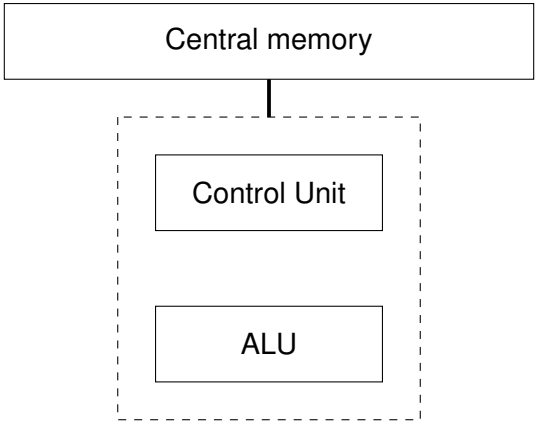
Example (1) of a parallel program : binarization

→ What is the best configuration to execute a task on a parallel architecture?

Plan

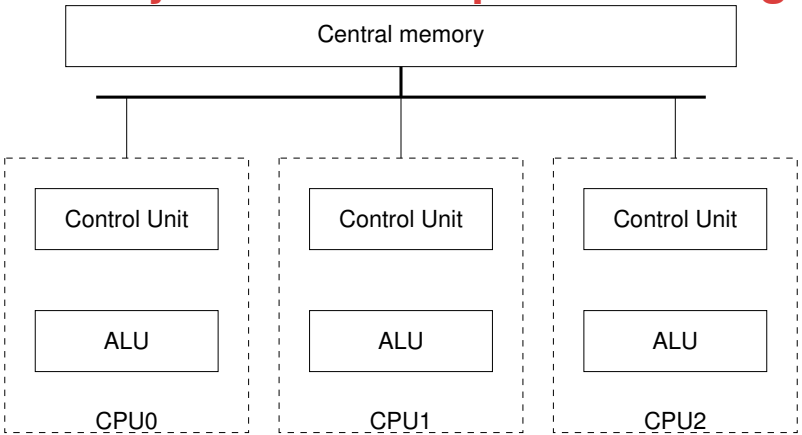
1. Sequential, Concurrent and PaPro
2. **Hardware architecture for parallel programming**
3. Sources of parallelisms
4. Parallelism efficiency
5. Practice preparation

Single CPU - Single DRAM



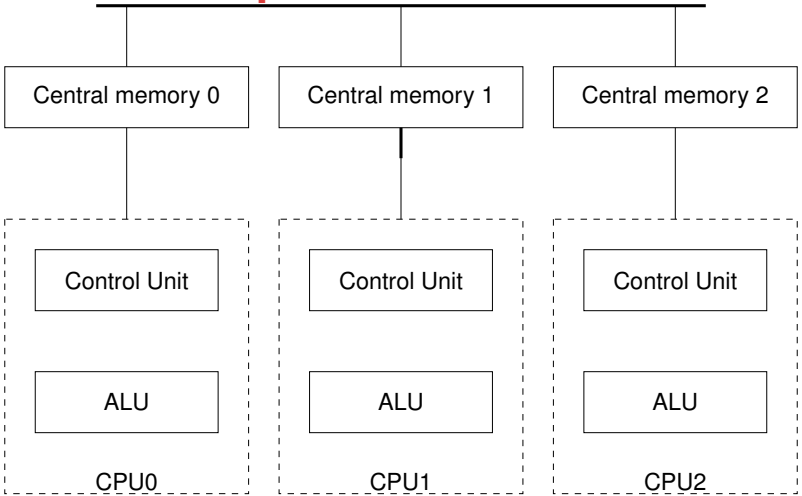
- Single central memory + single compute unit

Shared memory model : Multiple CPU - Single DRAM



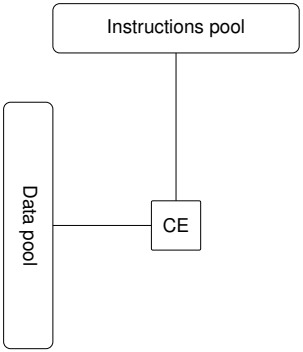
- Global memory which can be accessed by all processors of a parallel computer.
- Data in the global memory can be read/write by any of the processors.

Multiple CPU - Multiple DRAM



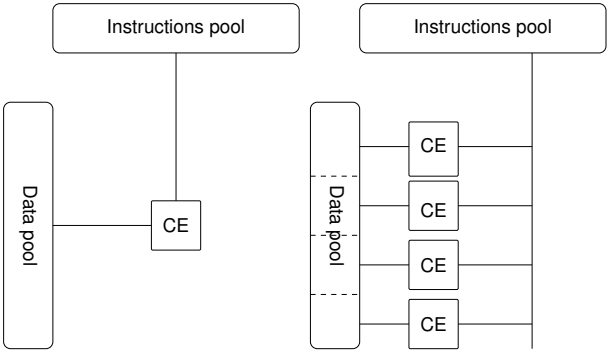
- Every core has its own memory and no other processor can access *directly* this memory

Classification : Flynn



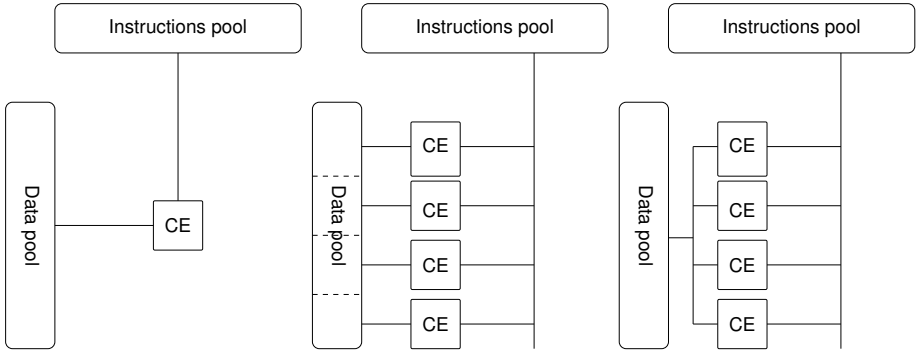
- Single instruction
  Single DATA SISD

Classification : Flynn



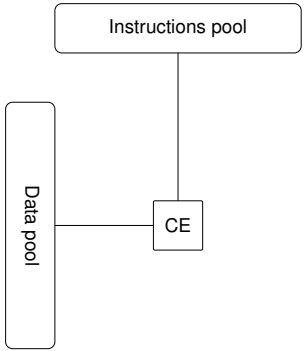
- Single instruction Single DATA SISD
- Single instruction multiple DATA SIMD

Classification : Flynn

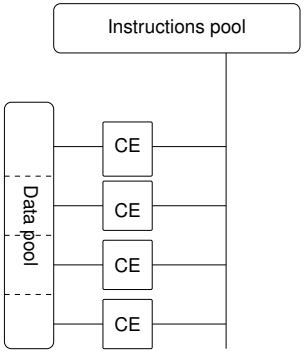


- Single instruction Single DATA SISD
- Single instruction multiple DATA SIMD
- Multiple instruction Single DATA MISD

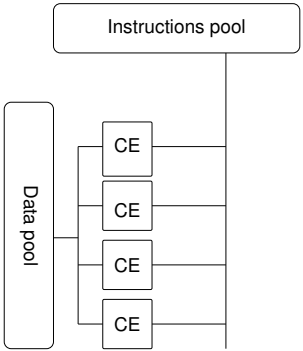
Classification : Flynn



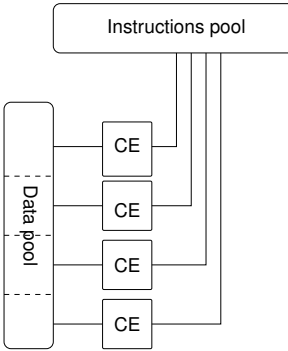
- Single instruction Single DATA SISD



- Single instruction multiple DATA SIMD



- Multiple instruction Single DATA MISD



- Multiple instruction Multiple DATA MIMD

Classification : Heterogeneity

Identical

Identical: The processors are identical; hence the execution time of a processing is the same on all processors.

Classification : Heterogeneity

Identical

Identical: The processors are identical; hence the execution time of a processing is the same on all processors.

Uniform

The rate of execution of a processing depends only on the speed of the processor. Thus, a processor of speed ×2, will execute a processing at twice of the rate of a processor of speed 1.

Classification : Heterogeneity

Identical

Identical: The processors are identical; hence the execution time of a processing is the same on all processors.

Uniform

The rate of execution of a processing depends only on the speed of the processor. Thus, a processor of speed ×2, will execute a processing at twice of the rate of a processor of speed 1.

Heterogeneous

The processors are different. The rate of execution of a processing depends on both the processor and the task. Indeed, not all tasks may be able to execute on all processors and a processing may have different execution rates on two different processors operating at the same speed.

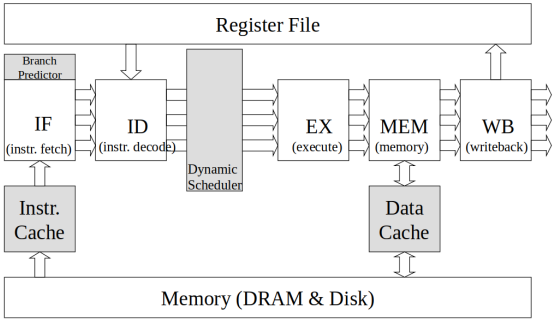
Plan

Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.

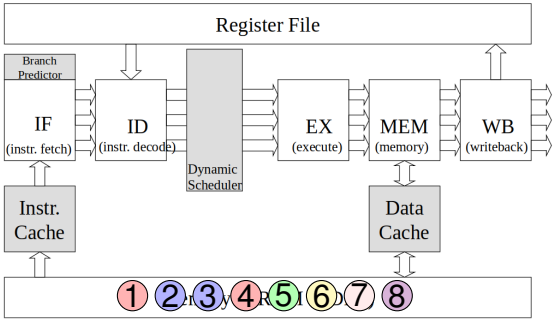
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



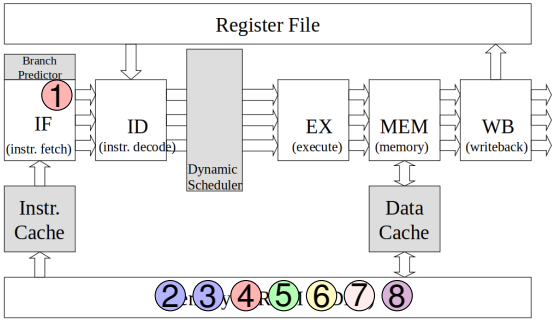
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



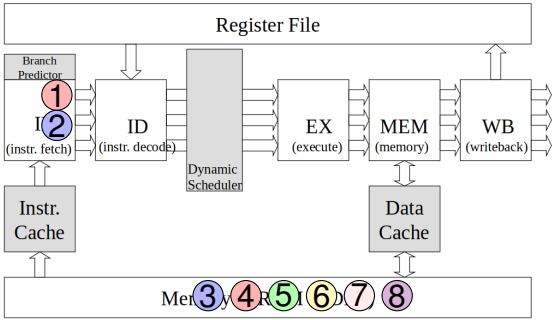
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



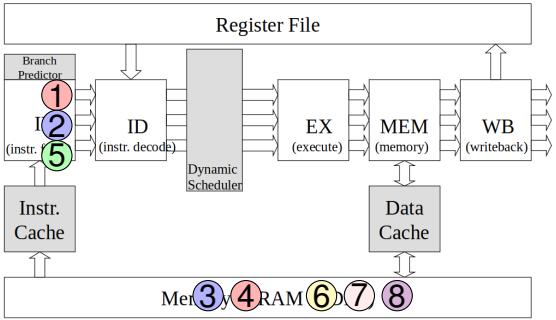
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



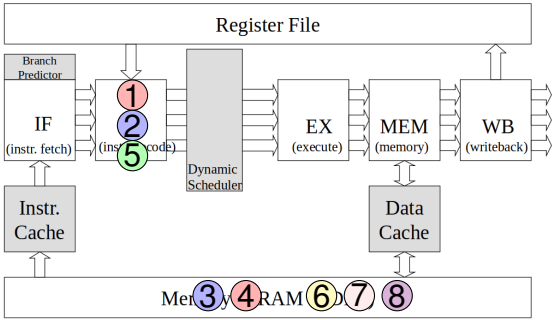
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



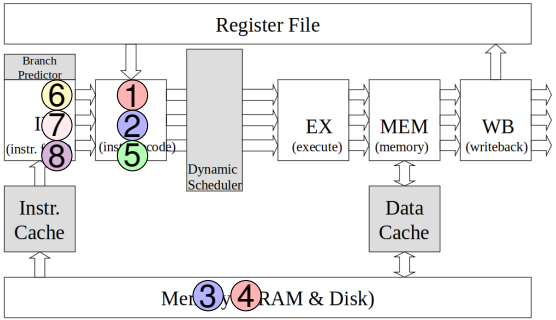
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



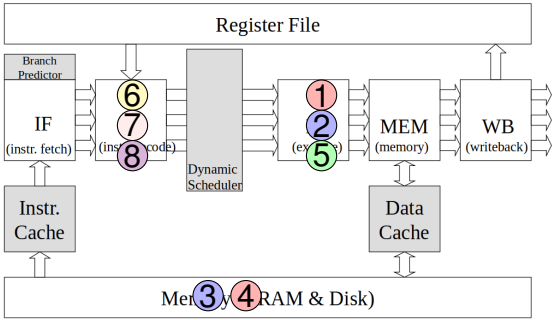
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



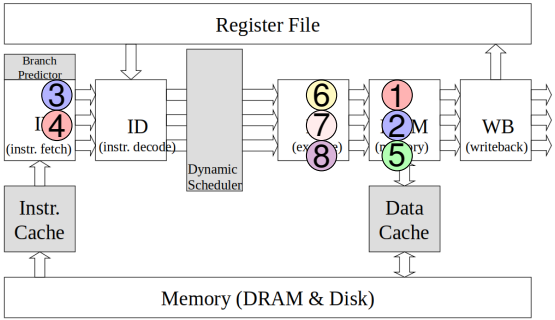
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



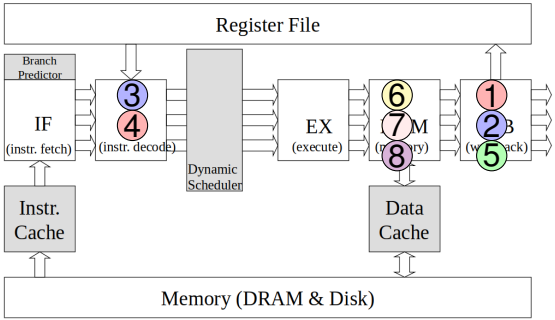
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



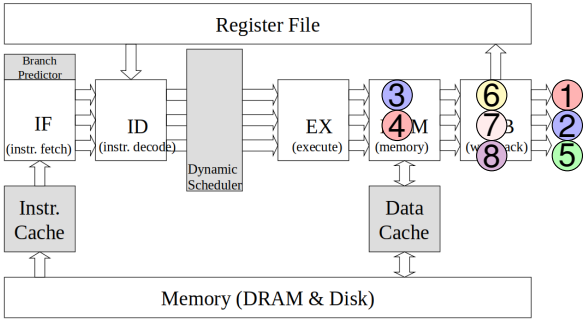
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



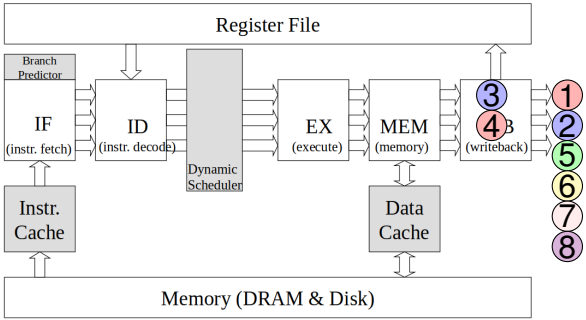
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



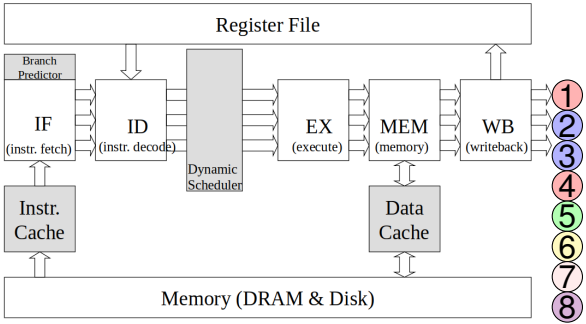
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



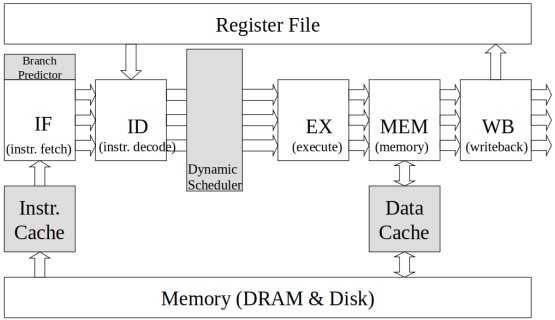
Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



Instruction level

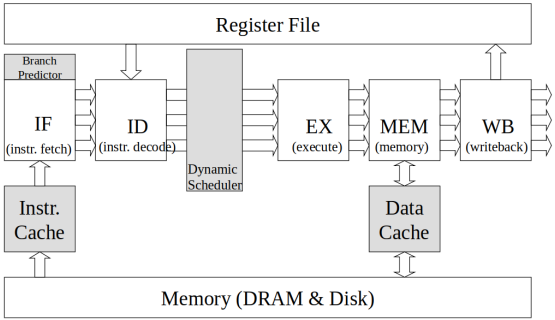
- is the ability to initiate multiple instructions during the same clock cycle.



- Automatic parallelization of sequential programs
  - Compiler performs dependence analysis on a sequential program's source data

Instruction level

- is the ability to initiate multiple instructions during the same clock cycle.



- Automatic parallelization of sequential programs
  - Compiler performs dependence analysis on a sequential program's source data
- Do not exploit functional parallelism (limited parallelism)

Data parallelism

- Execute the same instruction or program segment over different data sets simultaneously on multiple computing nodes.
  - Parallelism is exploited at data set level

Data parallelism

- Execute the same instruction or program segment over different data sets simultaneously on multiple computing nodes.
  - Parallelism is exploited at data set level

Example

```
void MatrixMul()
{
for (i = 0; i < rlen_A; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

Data parallelism

- Execute the same instruction or program segment over different data sets simultaneously on multiple computing nodes.
  - Parallelism is exploited at data set level

Example

```
void MatrixMul()
{
for (i = 0; i < rlen_A; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

```
void MatrixMul1()
{
for (i = 0; i < rlen_A/2; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

```
void MatrixMul2()
{
for(i=rlen_A/2; i < rlen_A; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

Data parallelism

- Execute the same instruction or program segment over different data sets simultaneously on multiple computing nodes.
  - Parallelism is exploited at data set level

Example

```
void MatrixMul()
{
for (i = 0; i < rlen_A; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

```
void MatrixMul1()
{
for (i = 0; i < rlen_A/2; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

```
void MatrixMul2()
{
for(i=rlen_A/2; i < rlen_A; i++)
{
  for (k = 0; k < clen_B; k++)
  {
    sum = 0;
    for (j = 0; j < clen_A; j++)
    {
      sum += A[i][j] * B[j][k];
    }
    C[i][k] = sum;
  }
}
```

- Data mapping is critical : static and dynamic?

Task parallelism

- Each processor performs a different task

Example

```
int main(int argc, char ** argv)
{
  color_correction();
  detect_lane();
  segmentation();
}
```

Task parallelism

- Each processor performs a different task

Example

```
int main(int argc, char ** argv)
{
  color_correction();
  detect_lane();
  segmentation();
}
```

```
int main(int argc, char ** argv)
{
  if (CPU==CPU1)
      color_correction();
  if (CPU==CPU2)
      detect_lane();
  if (CPU==CPU1){
    wait_for(detect_lane);
    segmentation();
  }
}
```

Task parallelism

- Each processor performs a different task

Example

```
int main(int argc, char ** argv)
{
  color_correction();
  detect_lane();
  segmentation();
}
```

```
int main(int argc, char ** argv)
{
  if (CPU==CPU1)
      color_correction();
  if (CPU==CPU2)
      detect_lane();
  if (CPU==CPU1){
    wait_for(detect_lane);
    segmentation();
  }
}
```

- More difficult to balance load

Domain decomposition and load balancing

Domain decomposition

- The computation domain is partitioned into several subdomains and then mapped onto processors of a parallel system.
- **In general**, the number of subdomains equals to the number of processors

Domain decomposition and load balancing

Domain decomposition

- The computation domain is partitioned into several subdomains and then mapped onto processors of a parallel system.
- **In general**, the number of subdomains equals to the number of processors

Load balancing

- The goal of partitioning is to distribute the computation load such that all processors can finish their computation at about the same time
- For identical cores parallel systems, the computation load is distributed as evenly as possible in a parallel computer.
- For heterogeneous parallel system, the computation load is distributed according to the computing power of each processor.

Granularity

- The size of load processed by a single computing component (software/hardware).

Granularity

- The size of load processed by a single computing component (software/hardware).
- Can be classified to :
  1. Fine-grain : In fine granularity, a process might consist of a few instructions (usually a lot of workers).
  2. Course-grain : each process contains a large number of sequential instructions and takes a substantial time to execute (usually few workers)
  3. Medium-grain : Medium granularity describes the middle ground between fine-grain and course grain.

Granularity

- The size of load processed by a single computing component (software/hardware).
- Can be classified to :
  1. Fine-grain : In fine granularity, a process might consist of a few instructions (usually a lot of workers).
  2. Course-grain : each process contains a large number of sequential instructions and takes a substantial time to execute (usually few workers)
  3. Medium-grain : Medium granularity describes the middle ground between fine-grain and course grain.

Alternative definition

granularity is defined as the time size of the computation between communication or synchronization points

granularity

- Increase the granularity allow reduce the cost of process creation and inter-process communication
- BUT will likely reduce the number of concurrent processes and the amount of parallelism

→ suitable compromise has to be made

- It is better to design a parallel program in which it is easy to vary granularity: i.e. a scalable program design.

Exercice

- Propose a granularity compute function to distribute the workload among m workers to achieve square matrix multiplications of n rows. Each worker is identified by its id ranging from 0 to $m - 1$.

Plan

Key Performance Indicators

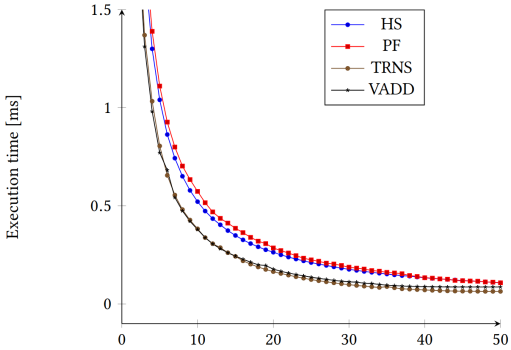
- Once a parallelization has been defined, how to measure the efficiency of my implementation?
- There are a number of metrics that can be used to evaluate its effectiveness of a parallelization
  - timing performance : metrics related to completion
  - clarity : *metric* related readability of domain decomposition
  - portability: *ability* to be auto-tune decomposition
  - generality : how much it is replicable
  - Embarrassingly parallel and inherently sequential

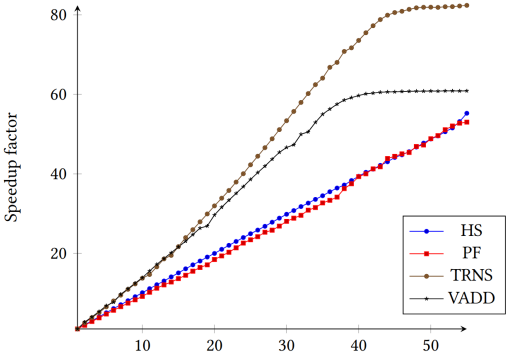
Speedup factor

- Refers to the ratio between sequential time and parallel-time $\mathcal{S} = \frac{T_{par}}{T_{seq}}$
- Bounded by the number of processors

Speedup factor

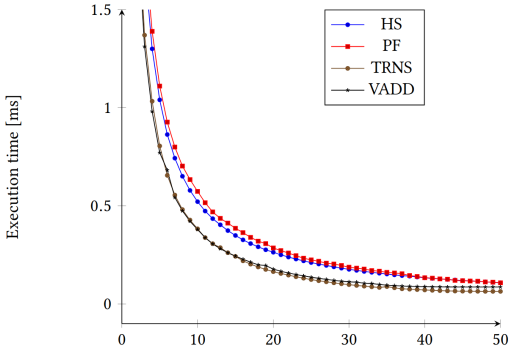
- Refers to the ratio between sequential time and parallel-time $\mathcal{S} = \frac{T_{par}}{T_{seq}}$
- Bounded by the number of processors

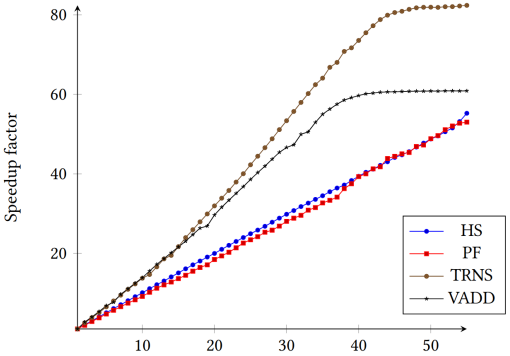




Speedup factor

- Refers to the ratio between sequential time and parallel-time $\mathcal{S} = \frac{T_{par}}{T_{seq}}$
- Bounded by the number of processors

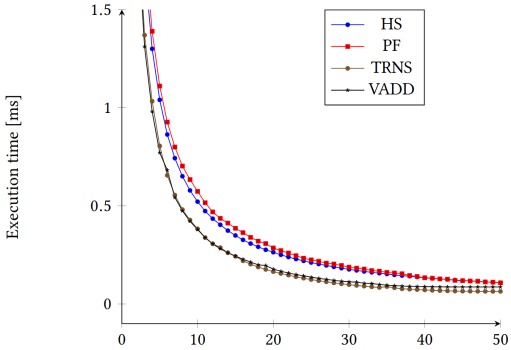


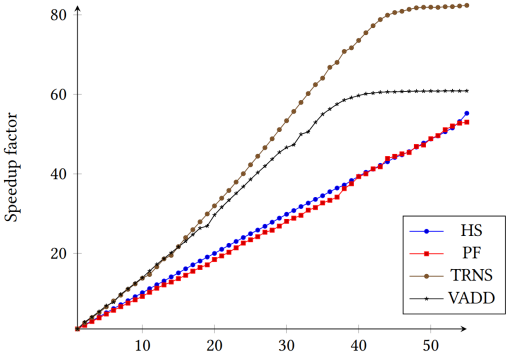


- Obtaining linear speedup factor is challenging, (*likely difficult* to develop)

Speedup factor

- Refers to the ratio between sequential time and parallel-time $\mathcal{S} = \frac{T_{par}}{T_{seq}}$
- Bounded by the number of processors





- Obtaining linear speedup factor is challenging, (*likely difficult* to develop)
- Bottleneck : communication and synchronization overheads

Amdahl's law

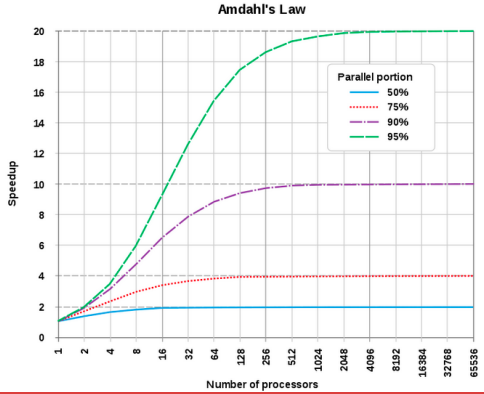
$$\mathcal{S} = \frac{1}{(1-p) + \frac{p}{s}}$$

- $\mathcal{S}$ is the theoretical speedup of the task;
- s is the speedup of the parallelizable parts of the task
- p is the proportion of execution time of the parallelizable part

Amdahl's law

$$\mathcal{S} = \frac{1}{(1-p) + \frac{p}{s}}$$

- $\mathcal{S}$ is the theoretical speedup of the task;
- s is the speedup of the parallelizable parts of the task
- p is the proportion of execution time of the parallelizable part



Other metrics

Clarity

- Is the ability to write clear, yet efficient parallel algorithms.
- Task and data allocation must be **clearly** identified
- Usually sacrificed to improve the speedup

Other metrics

Clarity

- Is the ability to write clear, yet efficient parallel algorithms.
- Task and data allocation must be **clearly** identified
- Usually sacrificed to improve the speedup

Generality

- Same approach for various types of problems
- Especially for parallel programming libraries : OpenMP, CUDA, OpenMPI, etc.

Other metrics : Portability

- A program's portability is practically assured in the sequential computing thanks to compilers.
- More complex for parallel application because of the extreme differences that exist between platforms
  - Number of cores, nature of cores, memory hierarchy, etc.
- Portability implies that a given program will behave consistently on all machines, regardless of their architectural features.
- Performance portability implies that the given program will have *good timing* performances even in the presence of hardware differences

embarrassingly parallel and non parallel problems

embarrassingly parallel and non parallel problems

- An embarrassingly parallel workload or problem (also called embarrassingly parallelizable, perfectly parallel) is the one where little or no effort is needed to separate the problem into a number of parallel tasks

embarrassingly parallel and non parallel problems

- An embarrassingly parallel workload or problem (also called embarrassingly parallelizable, perfectly parallel) is the one where little or no effort is needed to separate the problem into a number of parallel tasks

- Some classes of problems are not parallelizable by nature, ex. Fibonacci sequence (decision at iteration n depends on $n-1$)

- Some sequential by nature problems can be parallelized if the dependency can be spatially or temporally broken

Example (2) minimum tableau

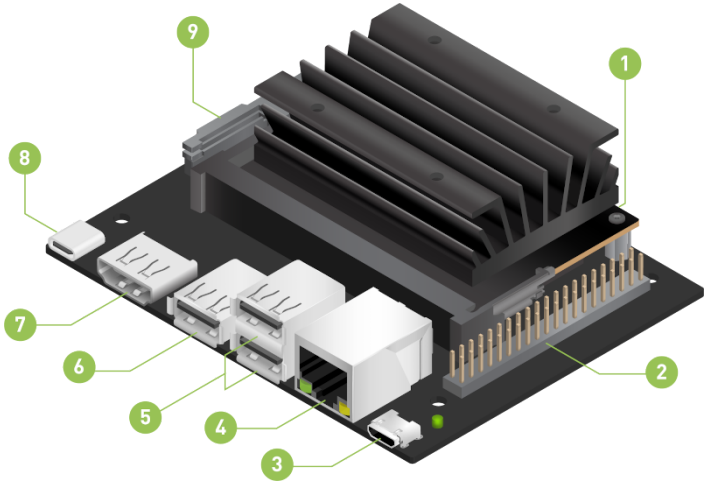
Exercise

Propose techniques to parallelize the problem of computing the minimum values of an array of size n

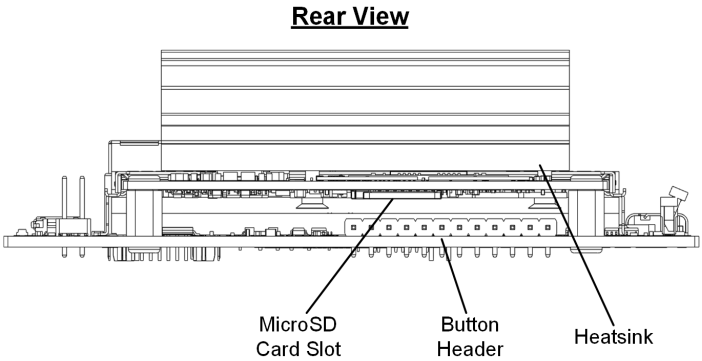
Plan

1. Sequential, Concurrent and PaPro
2. Hardware architecture for parallel programming
3. Sources of parallelisms
4. Parallelism efficiency
5. **Practice preparation**

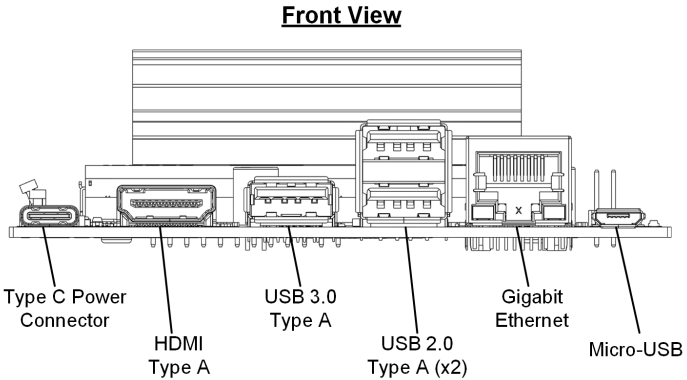
Parallelization requirements : NVIDIA Jetson Nano



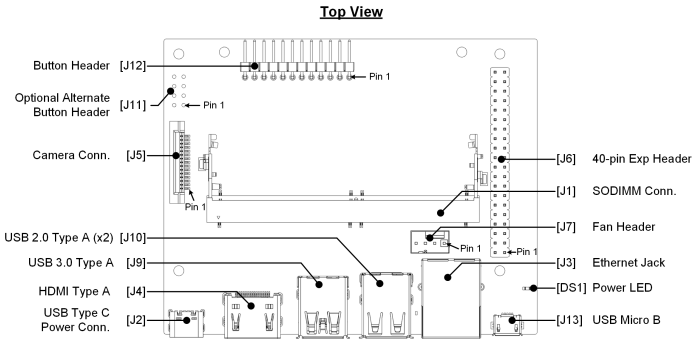
Parallelization requirements : NVIDIA Jetson Nano



Parallelization requirements : NVIDIA Jetson Nano



Parallelization requirements : NVIDIA Jetson Nano

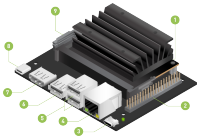


Secured remote connection with SSH (Practice)

What is SSH

- SSH is a remote connection protocol
- Connections are secured : asymmetric encryption, decryption
- Functions : secure command shell, port redirecting, file transfer

Client



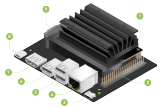
Server

- Several implementations exists : for the server it is Open SSH server

Remote secured connection by SSH

- each card is labeled by an identifier X.

Client
Adresse IP : 192.168.1.X



Server
Adresse IP : 192.168.1.(X+100)

Secure Command Shell

From the client:

1. ssh [utilisateur]@[machine]

Example :

- ssh geii@192.168.1.X

PS : do not forget to configure your IP address in the same network as the board.

File copy through SSH

- Achieved thanks to **scp**

Syntax

From client terminal :

```
1 scp [source] [destination]
```

Example

copy from client to server

- scp /home/geii/f1.txt geii@192.168.1.X:/home/zahaf/f1.txt

Copy from server to client

- scp geii@192.168.1.X:/home/zahaf/f1.txt /home/geii/f1.txt

Extra

Define a dynamic domain decomposition for the image binarization problem.