Introduction to Parallel and High Performance Computing

Oguz Kaya

Maˆıtre de Conf'erences
Universit'e Paris-Saclay and the LRI ParSys team, Orsay, France

Outline

7 Around the development of parallel programs

Outline

Around the development of parallel programs

Objectives

Objectives

- Getting to know parallel computing

Objectives

- Getting to know parallel computing
- Discover the applications that need computing power

Objectives

- Getting to know parallel computing
- Discover the applications that need computing power Explore the
- modern architecture of a parallel computer

Objectives

- Getting to know parallel computing
- Discover the applications that need computing power Explore the
- modern architecture of a parallel computer Introduce the main
- models of parallel programming

Objectives

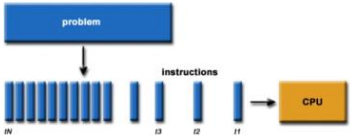
- Getting to know parallel computing
- Discover the applications that need computing power Explore the
- modern architecture of a parallel computer Introduce the main
- parallel programming models Introduce the basic concepts and
- examples

Objectives

- Getting to know parallel computing
- Discover the applications that need computing power Explore the
- modern architecture of a parallel computer Introduce the main
- parallel programming models Introduce the basic concepts and
- examples
- Provide some tips for developing effective parallel programs

Equential programming

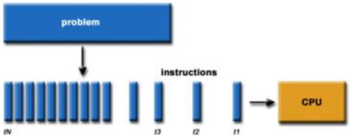
Traditionally,software is based on **sequential** calculation:



Equential programming

Traditionallysoftware is based on **sequential** calculation:

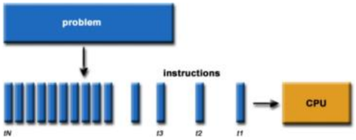
- A problem is often encountered in instructions.



Equential programming

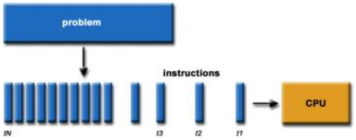
Traditionally,software is based on **sequential** calculation:

- A problem is often encountered in instructions.
- These instructions are executed **sequentially** one after the other.



Equential programming

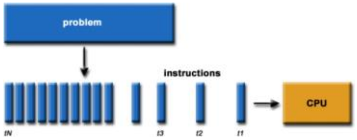
Traditionally,software is based on **sequential** calculation:



- A problem is often encountered in instructions.
- These instructions are executed **sequentially** one after the other.
- They are executed by **a single processor**.

Equential programming

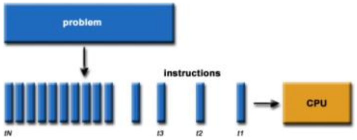
Traditionally,software is based on **sequential** calculation:



- A problem is often encountered in instructions.
- These instructions are executed **sequentially** one after the other.
- They are executed by **a single processor.** At a given
- moment, only one instruction is executed

Equential programming

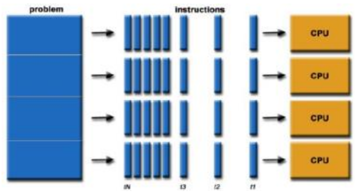
Traditionally, software is based on **sequential** calculation:



- A problem is often encountered in instructions.
- These instructions are executed **sequentially** one after the other.
- They are executed by **a single processor**. [illegible]
- moment, only one instruction is [illegible]
- The performance is mainly determined by **the frequency** (Hz) of the processor.

Parallel programming

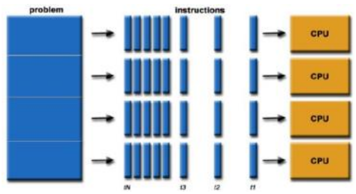
Parallel programming allows the use of several computing resources to solve a problem:



Parallel programming

Parallel programming allows the use of several computing resources to solve a problem:

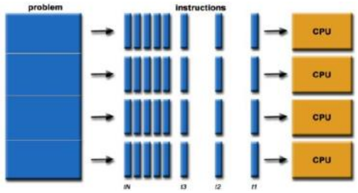
- A problem is divided into parts that can ˆetre launched in [illegible]



Parallel programming

Parallel programming allows the use of several computing resources to solve a [illegible]problem:

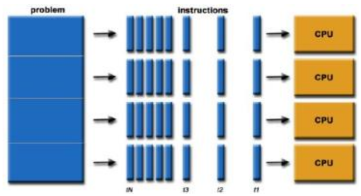
- A problem is divided into parts that can ˆetre launched i[illegible]
- Each part is still under instruction.



Parallel programming

Parallel programming allows the use of several computing resources to solve a problem:

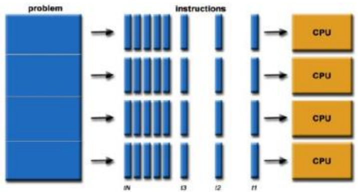
- A problem is divided into parts that can ˆetre launched in
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.



Parallel programming

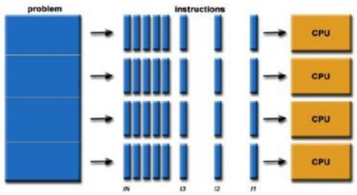
Parallel programming allows the use of several computing resources to solve a problem:

- A problem is divided into parts that can ˆetre launched in
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.
- The performance is determined by:



Parallel programming

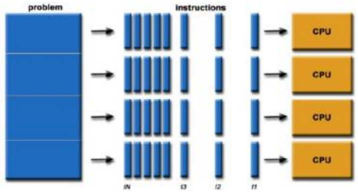
Parallel programming allows the use of several computing resources to solve a problem:



- A problem is divided into parts that can ˆetre launched [illegible]
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.
- The performance is determined by:
  - The frequency of the processor

Parallel programming

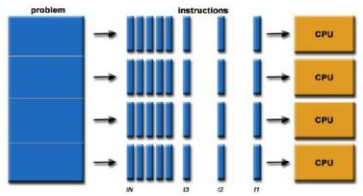
Parallel programming allows the use of several computing resources to solve a problem:



- A problem is divided into parts that can ˆetre launched [illegible]
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.
- The performance is determined by:
  - The frequency of the
  - processor The number of processors

Parallel programming

Parallel programming allows the use of several computing resources to solve a [illegible]problem:



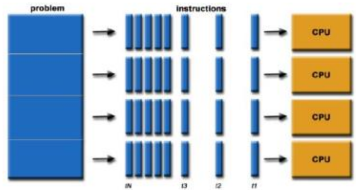
- A problem is divided into parts that can ˆetre launched [illegible]
- Each part is still under instruction.
- The instructions of each part are executed **in parallel** using several processors.
- The performance is determined by:
  - The frequency of the
  - processor The number of processors
  - The degree of parallelism of the problem

Outline

Parallel programming

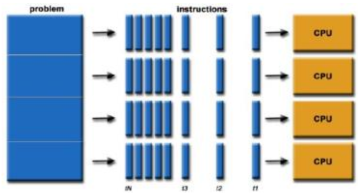
Applications of parallel computing

Many time-consuming applications in various fields:



Applications of parallel computing

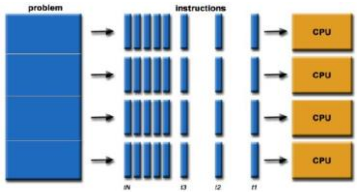
Many time-consuming applications in various fields:



- Scientific computing: Simulations in physics, chemistry, biology, ...

Applications of parallel computing

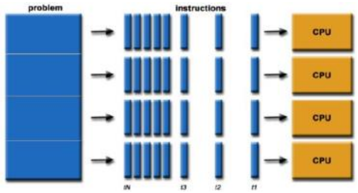
Many time-consuming applications in various fields:



- Scientific computing: Simulations in physics, chemistry, biology, ...
- Neural network processing

Applications of parallel computing

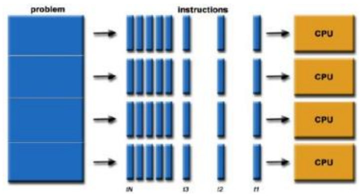
Many time-consuming applications in various fields:



- Scientific computing: Simulations in physics, chemistry, biology, ...
- Neural network processing Graphics
- (rendering, video games, etc.)

Applications of parallel computing

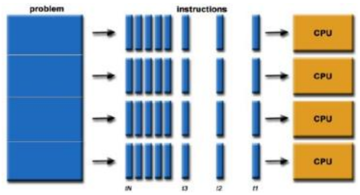
Numerous time-consuming applications in various fields:



- Scientific computing: Simulations in physics, chemistry, biology, ...
- Neural network processing Graphics
- (rendering, video games, etc.)
- Operating systems (Linux, Android, etc.)

Applications of parallel computing

Many time-consuming applications in various fields:



- Scientific computing: Simulations in physics, chemistry, biology, ...
- Neural network processing Graphics
- (rendering, video games, etc.)
- Operating systems (Linux, Android, etc.) and
- many others...

Maximum frequency of a CPU in 2002

What is the maximum frequency of this CPU in 2002?

Intel Pentium 4, Northwood

Maximum frequency of a CPU in 2002

What is the maximum frequency of this CPU in 2002?

Intel Pentium 4, Northwood

- 3.06 GHz

Maximum frequency of a CPU

What is the maximum frequency of a CPU in 2020?

Intel Core i9-10900K, Comet Lake

Maximum frequency of a CPU

What is the maximum frequency of a CPU in 2020?

Intel Core i9-10900K, Comet Lake

- 5.3 GHz

Maximum frequency of a CPU

What is the maximum frequency of a CPU in 2020?

Intel Core i9-10900K, Comet Lake

- 5.3 GHz
- The performance crˆete? 5.3 Gflops/s?

Maximum frequency of a CPU

What is the maximum frequency of a CPU in 2020?

Intel Core i9-10900K, Comet Lake

- 5.3 GHz
- The performance crˆete? 5.3 Gflops/s?
- No! ≈ 1.7Tflops/s (1700Gflops/s) How?

Maximum frequency of a CPU

What is the maximum frequency of a CPU in 2020?

Intel Core i9-10900K, Comet Lake

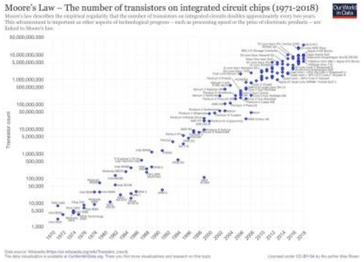
- 5.3 GHz
- The performance cr^ete? 5.3 Gflops/s?
- No! ≈ 1.7Tflops/s (1700Gflops/s) How?
- Thanks to deadans parallelism (10 cores, each with 2 AVX256 virtual units)

Outline

Maximum frequency of a CPU

Moore's Law

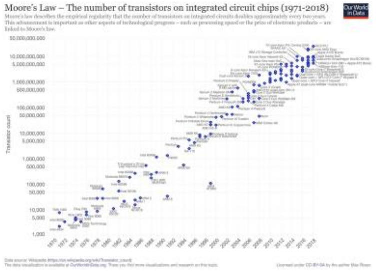
The number of transistors in the integrated circuits doubles every 2 years.



Moore's Law

Moore's Law

The number of transistors in the `integrated` circuits doubles every 2 years.

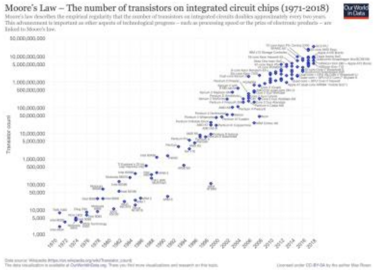


Moore's Law

- What is it ?

Moore's Law

The number of transistors in the `integrated` circuits doubles every 2 years.

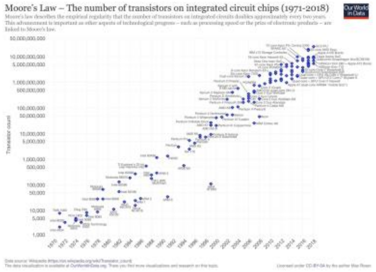


Moore's Law

- What is it ?
- More transistors → more performance potential

Moore's Law

The number of transistors in the `integrated` circuits doubles every 2 years.

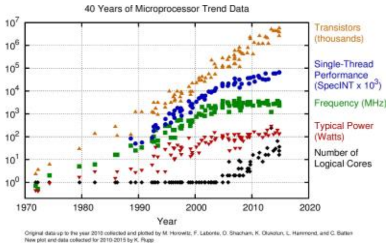


Moore's Law

- What is it ?
- More transistors → more performance potential
- Moore's law is still valid today (although a bit slowed down).

Dennard Scaling

If the engraving size is reduced by -30% (0.7x) in each generation, this gives

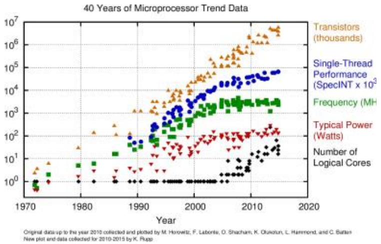


Dennard Scaling

Dennard Scaling

If the engraving size is reduced by -30% (0.7x) in each generation, this gives

- A decrease in the circuit area of 50%.

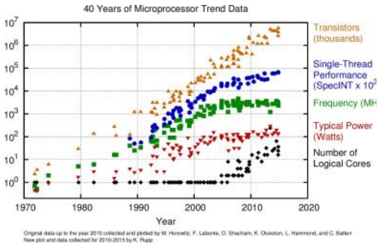


Dennard Scaling

Dennard Scaling

If the engraving size is reduced by -30% (0.7x) in each generation, this gives

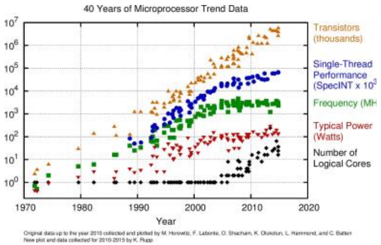
- 50% decrease in circuit area 30%
- decrease in latency



Dennard Scaling

Dennard Scaling

If the engraving size is reduced by -30% (0.7x) in each generation, this gives

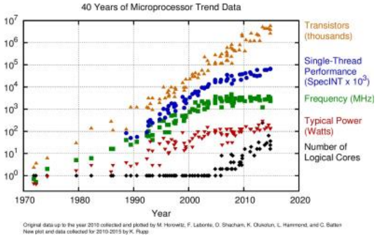


Dennard Scaling

- 50% decrease in circuit area 30%
- decrease in latency
- Consequently, a 40% increase in frequence (10/7 ≈ 1.4).

Dennard Scaling

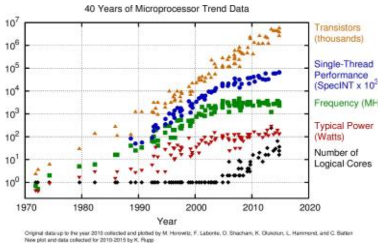
If the engraving size is reduced by -30% (0.7x) in each generation, this gives



- 50% decrease in circuit area 30%
- decrease in latency
- Consequently, a 40% increase in frequence (10/7 ≈ 1.4).
  - Not possible anymore because of quantum effects!

Dennard Scaling

If the engraving size is reduced by -30% (0.7x) in each generation, this gives



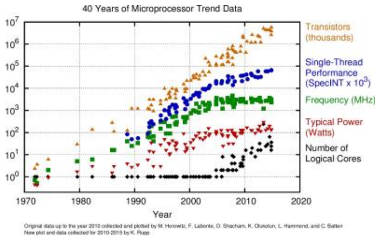
Dennard Scaling

Dennard Scaling

- 50% decrease in circuit area 30%
- decrease in latency
- Consequently, a 40% increase in frequence (10/7 ≈ 1.4).
  - Not possible anymore because of quantum effects!
  - Use either multi-core or large-core.

Dennard Scaling

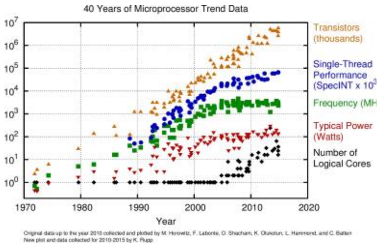
If the engraving size is reduced by -30% (0.7x) in each generation, this gives



- 50% decrease in circuit area 30%
- decrease in latency
- Consequently, a 40% increase in frequence (10/7 ≈ 1.4).
  - Not possible anymore because of quantum effects!
  - Use either multi-core or large-core.
- *energy ∼ frequency2*

Acceleration and efficiency

If the engraving size is reduced by -30% (0.7x) in each generation, this gives



Dennard Scaling

Cray-1 Supercomputer (1975) with peak performance of ≈ 160Mflops/s

- 50% decrease in circuit area 30%
- decrease in latency
- Consequently, a 40% increase in frequence (10/7 ≈ 1.4).
  - Not possible anymore because of quantum effects!
  - Use either multi-core or large-core.
- *energy ∼ frequency2*
  - Use more cores at a lower frequency

    → more performance in the mˆeme fenˆetre energetics

Dennard Scaling

How much faster does a parallel program run? How well are computing resources [illegible]

Acceleration and efficiency

How much faster does a parallel program run? How well are computing resources used?

Cray-1 Supercomputer (1975) with peak performance of ≈ 160Mflops/s

- Sequential execution time: $T(1)$

Acceleration and efficiency

How much faster does a parallel program run? How well are computing resources [illegible]

Cray-1 Supercomputer (1975) with peak performance of ≈ 160Mflops/s

- Sequential execution time: $T(1)$
- The execution time on N processors: $T(N)$

Acceleration and efficiency

How much faster does a parallel program run? How well are computing resources used?

Cray-1 Supercomputer (1975) with peak performance of ≈ 160Mflops/s

- Sequential execution time: $T(1)$
- The execution time on N processors: $T(N)$
- **Acceleration**: $S(N) = T(1)/T(N)$

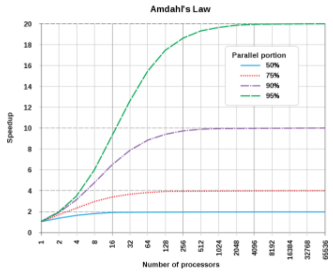
Acceleration and efficiency

How much faster does a parallel program run? How well are computing resources used?

Cray-1 Supercomputer (1975) with peak performance of ≈ 160Mflops/s

- Sequential execution time: $T(1)$
- The execution time on N processors: $T(N)$
- **Acceleration**: $S(N) = T(1)/T(N)$
- **Efficiency**: $E(N) = S(N)/N$

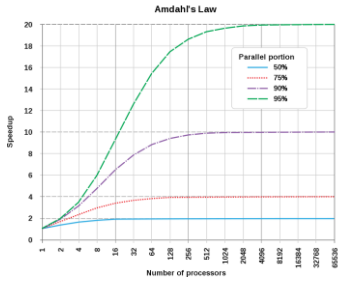
Amdahl's Law



Amdahl's Law

Amdahl's Law

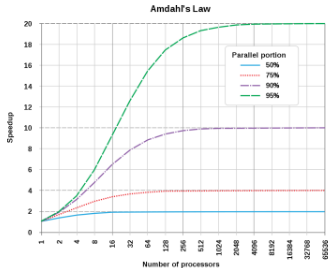
How much faster could a program run? What is the limit?



Amdahl's Law

- Introduced in 1967 by Gene Amdahl

Amdahl's Law



Amdahl's Law

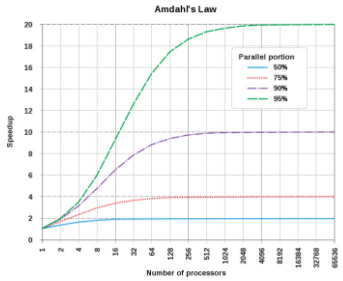
Introduced in 1967 by Gene Amdahl

- Let *s* and *p* be the **equential** fraction **s**
- and

  parallelizable of a program ($s + p = 1.0$).

Amdahl's Law

How much faster could a program run? What is the limit?

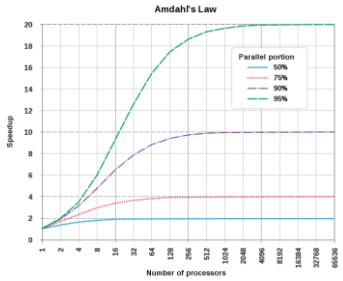


- Introduced in 1967 by Gene Amdahl
- Let *s* and *p* be the **equential** fraction **s** and **parallelizable** of a program ($s + p = 1.0$).
- The achievable acceleration is defined by

$$A(N) = \frac{T(1)}{T(N)} \leq \frac{T(1)}{pT(1)/N + sT(1)} = \frac{1}{p/N + s}$$

Amdahl's Law

How much faster could a program run? What is the limit?



Amdahl's Law

- Introduced in 1967 by Gene Amdahl
- Let s and p be the **equential** fraction **s** and **parallelizable** of a program ($s + p = 1.0$).
- The achievable acceleration is defined by

$$A(N) = \frac{T(1)}{T(N)} \leq \frac{T(1)}{pT(1)/N+sT(1)} = \frac{1}{p/N+s}$$

-

Amdahl's Law

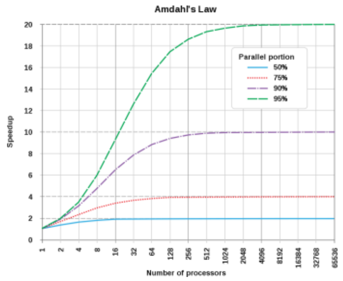
Amdahl's Law

Strong scaling:

Parallel scalability for a fixed size problem

Amdahl's Law

How much faster could a program run? What is the limit?



Amdahl's Law

- Introduced in 1967 by Gene Amdahl
- Let *s* and *p* be the **equential** fraction **s** and **parallelizable** of a program ($s + p = 1.0$).
- The achievable acceleration is defined by

$$A(N) = \frac{T(1)}{T(N)} \leq \frac{T(1)}{pT(1)/N + sT(1)} = \frac{1}{p/N + s}$$

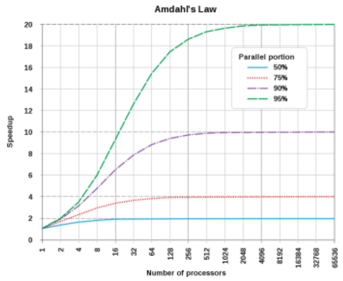
- **Strong scaling:** Parallel scalability for a fixed size problem

Amdahl's Law

- Hope for parallel computing?

Amdahl's Law

How much faster could a program run? What is the limit?



Amdahl's Law

- Introduced in 1967 by Gene Amdahl
- Let *s* and *p* be the **equential** fraction **s** and **parallelizable** of a program ($s + p = 1.0$).
- The achievable acceleration is defined by
$$A(N) = \frac{T(1)}{T(N)} \leq \frac{T(1)}{pT(1)/N + sT(1)} = \frac{1}{p/N+s}$$
-

Amdahl's Law

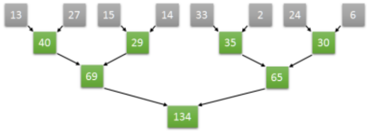
Amdahl's Law

Strong scaling:

- Parallel scalability for a fixed size problem
- Hope for parallel computing?
  - Typically, *S* decreases with the size of the problem.

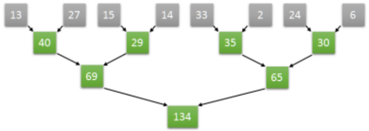
Example: Sum of an array

$$\text{L-N-1}_{i=0}\; A[i\,]$$



Sum in parallel

Example: Sum of an array



Sum in parallel

$$\sum_{i=0}^{N-1} A[i]$$

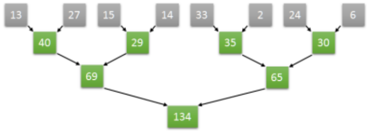
- N - 1 additions

Example: Sum of an array

L- N-1 $A[i$
i=0
]

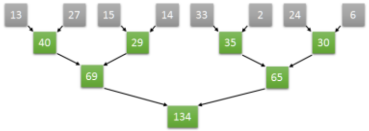
- N - 1 additions
- For N = 8, 7 additions 6which 3 are equal,

$S \leq 7/3 = 2.34$



Sum in parallel

Example: Sum of an array

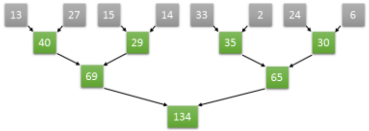


Sum in parallel

$$\sum_{i=0}^{N-1} A[i]$$

- N - 1 additions
- For N = 8, 7 additions of which 3 are equal, $S \leq 7/3 = 2.34$
- For N = 16, 15 additions of which 4 are equential, $S \leq 15/4 = 3.75$

Example: Sum of an array



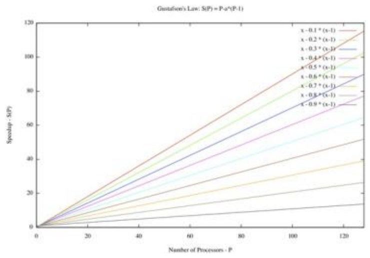
Sum in parallel

$$\sum_{i=0}^{N-1} A[i]$$

- N - 1 additions
- For N = 8, 7 additions of which 3 are equal, $S \leq 7/3 = 2.34$
- For N = 16, 15 additions of which 4 are equential, $S \leq 15/4 = 3.75$
- In the general case, N - 1 additions of which log N equals, $S \leq (N - 1)/\log N$

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

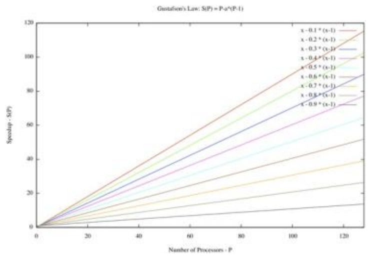


Gustafson's law

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

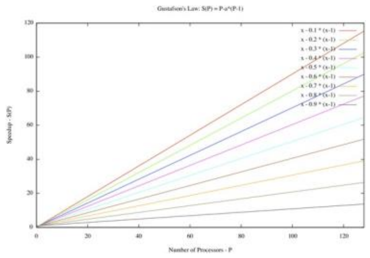
- $S(N) = N - (1 - N)s$



Gustafson's law

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

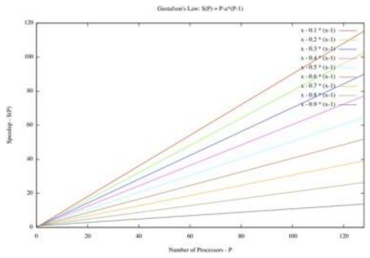


Gustafson's law

- $S(N) = N - (1 - N)s$
- Amdalh: It could ˆetre difficult accèler un calculation with more processors

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

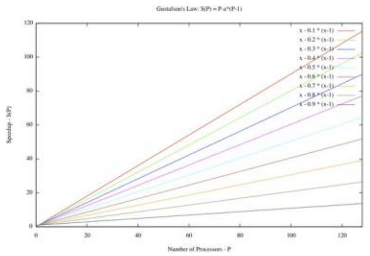


Gustafson's law

- $S(N) = N - (1 - N)s$
- Amdalh: It could ˆetre difficult accèler un calculation with more processors
- Gustafson: It is possible to do more computation as fast with more processors

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

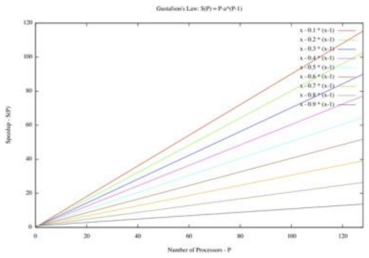


Gustafson's law

- $S(N) = N - (1 - N)s$
- Amdalh: It could ˆetre difficult accèler un calculation with more processors
- Gustafson: It is possible to do more computation as fast with more processors
- Which one is more relevant in practice?

Gustafson's law

How much acceleration could be obtained if the problem size increases with the number of processors?

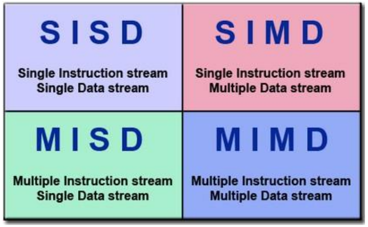


Gustafson's law

- $S(N) = N - (1 - N)s$
- Amdalh: It could ˆetre difficult accèler un calculation with more processors
- Gustafson: It is possible to do more computation as fast with more processors
- Which one is more relevant in practice?
- **Weak scaling**Scalability parallel to the The problem size is increasing.

Flynn's classification

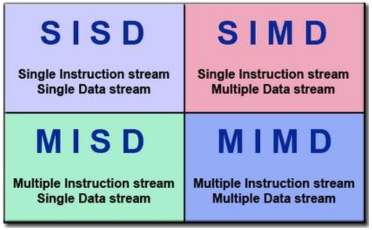
A 2 × 2 matrix to classify parallel machines.


SISD
Single Instruction stream Single Data stream
SIMD
Single Instruction stream Multiple Data stream
MISD
Multiple Instruction stream Single Data stream
MIMD
Multiple Instruction stream Multiple Data stream

Flynn's classification

Flynn's classification

A 2 × 2 matrix to classify parallel machines.

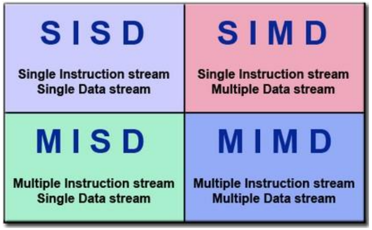

SISD
Single Instruction stream Single Data stream
SIMD
Single Instruction stream Multiple Data stream
MISD
Multiple Instruction stream Single Data stream
MIMD
Multiple Instruction stream Multiple Data stream

Flynn's classification

- Extended classification, used since 1966

Flynn's classification

A 2 × 2 matrix to classify parallel machines.

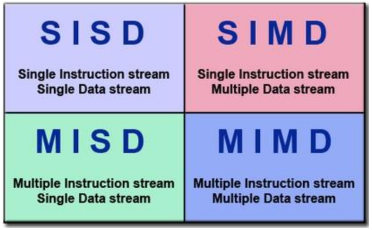

S I S D
Single Instruction stream
Single Data stream
S I M D
Single Instruction stream
Multiple Data stream
M I S D
Multiple Instruction stream
Single Data stream
M I M D
Multiple Instruction stream
Multiple Data stream

Flynn's classification

- Extended classification, used since 1966 The X
- axis: single, multiple instruction

Flynn's classification

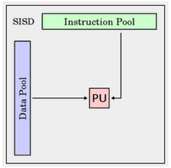
A 2×2 matrix to classify parallel machines.

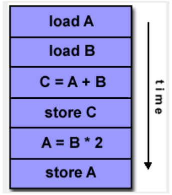

SISD
Single Instruction stream Single Data stream
SIMD
Single Instruction stream Multiple Data stream
MISD
Multiple Instruction stream Single Data stream
MIMD
Multiple Instruction stream Multiple Data stream

Flynn's classification

- Extended classification, used since 1966 X
- axis: single (simple), multiple instruction Y
- axis: single (simple), multiple data

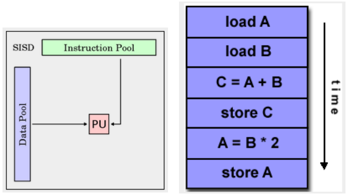
Single Instruction, Single Data (SISD)





SISD Processor

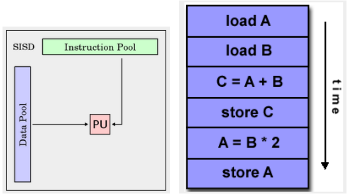
Single Instruction, Single Data (SISD)



SISD Processor

- A completely sequential m

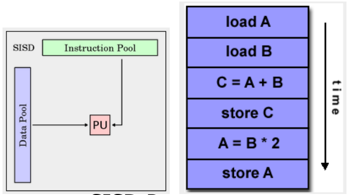
Single Instruction, Single Data (SISD)



SISD Processor

- A completely sequential m̶e
- Single" instructionOnly one instruction is executed eachclock cycle

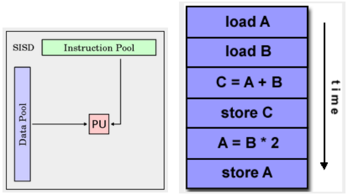
Single Instruction, Single Data (SISD)



SISD Processor

- A completely sequential m[illegible]
- Single" instruction: Only one instruction is executed at each clock cycle
- Single data: Only one data stream is used

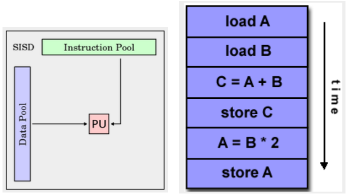
Single Instruction, Single Data (SISD)



SISD Processor

- A completely sequential m
- Single" instruction Only one instruction is executed each clock cycle
- Single data: Only one data stream is used
- Execution of eterminist

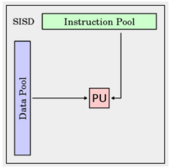
Single Instruction, Single Data (SISD)

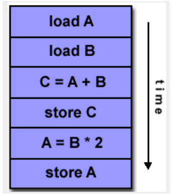


SISD Processor

- A completely sequential me
- Single" instruction:Only one instruction is executed each clock cycle
- Single data: Only one data stream is used
- Execution of eterminist
- The oldest computers (i.e., architecture courses)

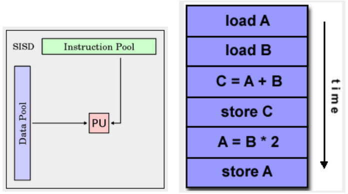
Single Instruction, Multiple Data (SIMD)





SISD Processor

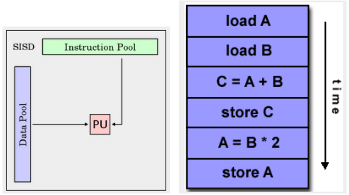
Single Instruction, Multiple Data (SIMD)



SISD Processor

- A completely sequential m[illegible]

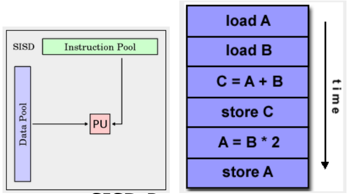
Single Instruction, Multiple Data (SIMD)



SISD Processor

- A completely sequential me
- Single" instructionOnly one instruction is executed each clock cycle

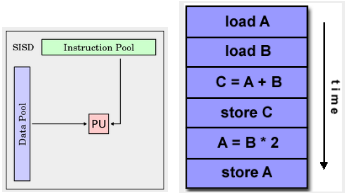
Single Instruction, Multiple Data (SIMD)



SISD Processor

- A completely sequential rte
- Single" instructionOnly one instruction is executed each clock cycle
- Single data: Only one data stream is used

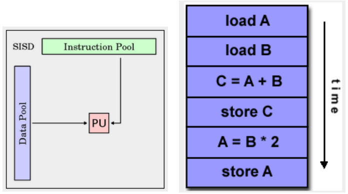
Single Instruction, Multiple Data (SIMD)



SISD Processor

- A completely sequential m[illegible]
- Single" instruction: Only one instruction is executed each clock cycle
- Single data: Only one data stream is used
- Execution of eterminist

Single Instruction, Multiple Data (SIMD)

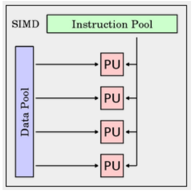


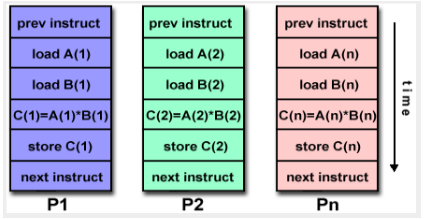
SISD Processor

- A completely sequential me
- Single" instructionOnly one instruction is executed each clock cycle
- Single data: Only one data stream is used
- Execution of eterminist
- The oldest computers (i.e., architecture courses)

Single Instruction, Multiple Data (SIMD)

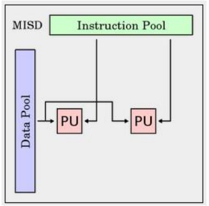
- An example of a parallel machine
- Instruction "Single" : All the computing units execute the same instruction at each clock cycle
- Multiple data: Each computing unit can operate on a different data Compatible for quite
- regular problems like image processing

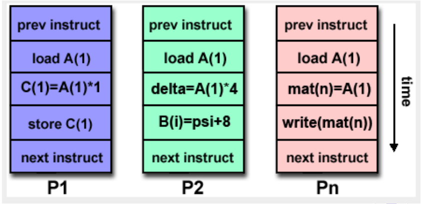




Multiple Instruction, Single Data (MISD)

- An example of a parallel machine
- Multiple Instruction: Each computing unit operates on independent data via different instruction flows
- Single" data: A single data stream feeds several computing units Some
- examples of use are :
  - several frequency filters operating on a single signal
  - several cryptographic algorithms trying to decrypt a single coded message

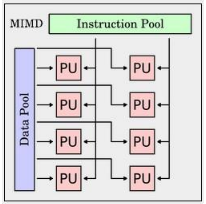


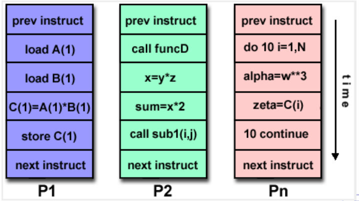


université PARIS-SACLAY

Multiple Instruction, Multiple Data (MIMD)

- An example of a parallel machine
- Multiple Instructioneach computing unit can execute a different instruction flow
- Multiple data: each processor can operate on a different data stream Execution can
- be synchronous or asynchronous, deterministic or non-deterministic The majority
- of modern parallel machines belong to this category Several MIMD architectures
- also include SIMD components

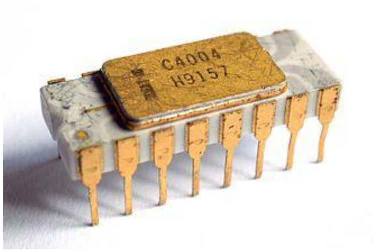




Outline

Parallelism at bit level

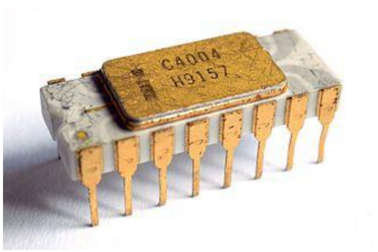
This is the word-size of the processor.



Intel C4004, 4-bit processor (1971)

Parallelism at bit level

This is the word-size of the processor.

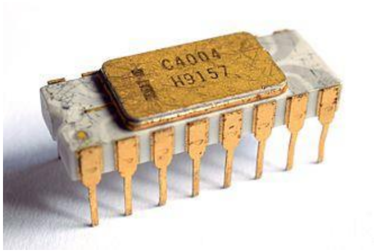


Intel C4004, 4-bit processor (1971)

- Relevant for the years 1970..1986.

Parallelism at bit level

This is the word-size of the processor.

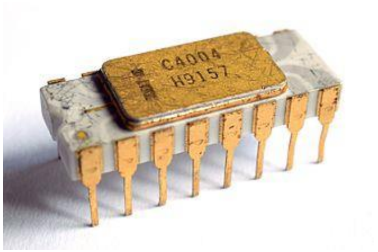


Intel C4004, 4-bit processor (1971)

- Relevant for the years 1970..1986.
- The word size: 4-bit → 8-bit → 16-bit → 32-bit

Parallelism at bit level

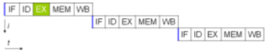
This is the word-size of the processor.



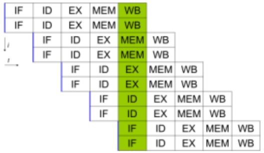
Intel C4004, 4-bit processor (1971)

- Relevant for the years 1970..1986.
- The word size: 4-bit $\rightarrow$ 8-bit $\rightarrow$ 16-bit $\rightarrow$ 32-bit
- Convergée in 64-bit today.

Parallèlism at the instruction level (ILP)



Ex'ecution without pipeline

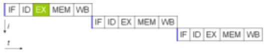


Superscalar execution with pipeline

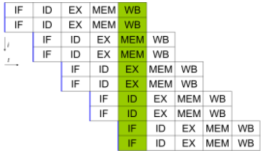
Parallèlism at the instruction level (ILP)

Allows to execute independent instructions simultaneously.

- **a = b + c; c = d + e; f = a + c; g = c - a;**

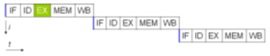


Ex'ecution without pipeline

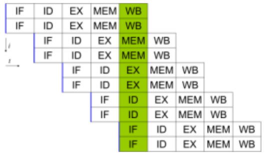


Superscalar execution with pipeline

Parallèlism at the instruction level (ILP)



Ex'ecution without pipeline

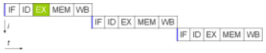


Superscalar execution with pipeline

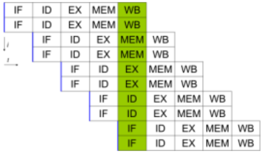
- **a = b + c; c = d + e;**
  f = a + c; g = c - a;
- Possibilitè 1: Pipeline. Differents ètapes de deux instructions ind'ependantes peuvent ˆetre exècutèes dans le pipeline en mˆeme temps.

Parallèlism at the instruction level (ILP)

Allows to execute independent instructions simultaneously.



Ex'ecution without pipeline

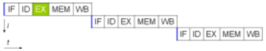


Superscalar execution with pipeline

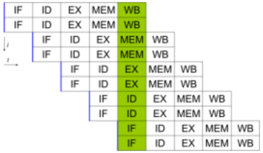
- **a = b + c; c = d + e; f = a + c; g = c - a;**
- Possibility 1: Pipeline. Different stages of two ind'ependant instructions can ˆetre exècutèes in the pipeline en mˆeme temps.
- Possibility 2: Superscalar execution: Independent instructions can use several execution units (i.e. ALU) in a **superscalar** processor.

Data Parallelism

Allows to execute independent instructions simultaneously.



Ex'ecution without pipeline

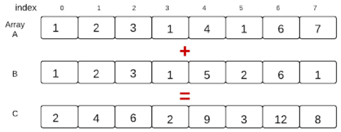


Superscalar execution with pipeline

- **a = b + c; c = d + e; f = a + c; g = c - a;**
- Possibility 1: Pipeline. Different stages of two ind'ependant instructions can ˆetre exécutées in the pipeline en mˆeme temps.
- Possibility 2: Superscalar execution: Independent instructions can use several execution units (i.e. ALU) in a **superscalar** processor.
- Most modern processors are superscalar.

Parallelism at the instruction level (ILP)

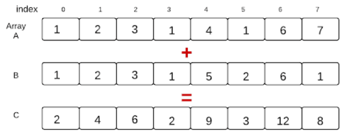
Carry out the m ˆemes independent operations of the data tables.



Addition of the two tables

Data Parallelism

Carry out the m ˆemes ind'ependent operations of the data tables.

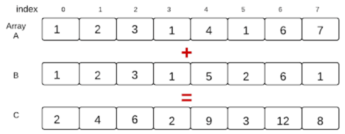


Addition of the two tables

- Suitable for the SIMD paradigm.

Data Parallelism

Carry out the mˆemes ind'ependent operations of the data tables.

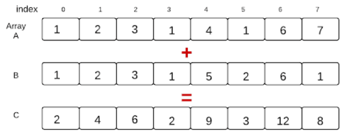


Addition of the two tables

- Suitable for the SIMD paradigm.
- Well adapted hardware for efficient execution (CPU and GPU vector unit)

Data Parallelism

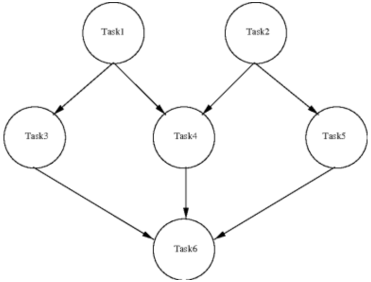
Carry out the m ˆemes ind'ependent operations of the data tables.



Addition of the two tables

- Suitable for the SIMD paradigm.
- Well adapted hardware for efficient execution (CPU and GPU vector unit)
- Extensive use in scientific computing (matrices, vectors), graphics (rendering), image and signal processing (filters).

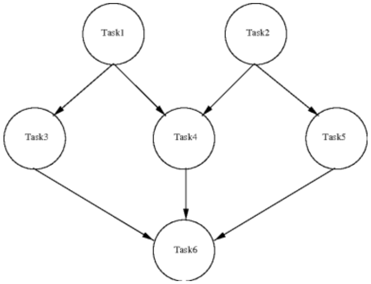
Task Parallelism



Task parallelism

Task Parallelism

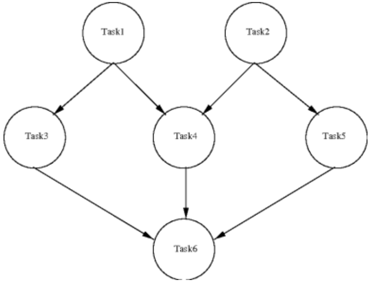
It is the execution of ind'ependent tasks in a program.



Task parallelism

- Examples of ind'ependent tasks?

Task Parallelism

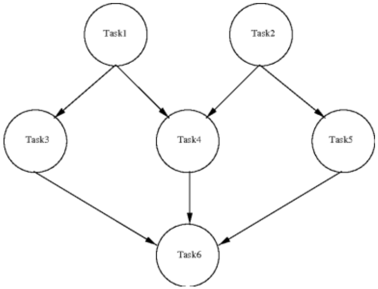


Task parallelism

- Examples of ind'ependent tasks?
    - Calls to functions: **a = f(x); b = g(y);**

Task Parallelism

It is the execution of ind'ependent **tasks** in a program.

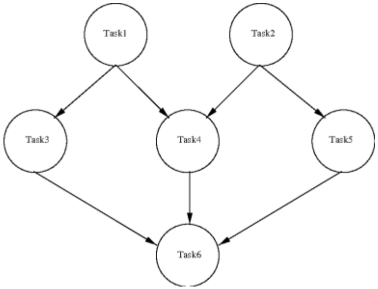


Task parallelism

- Examples of ind'ependent **tasks**?
  - Appels aux fonctions: **a = f(x); b =**
  - **g(y);** Blocs de code indèpendents dans une mˆeme fonction.

Task Parallelism

It is the execution of ind'ependent tasks in a program.

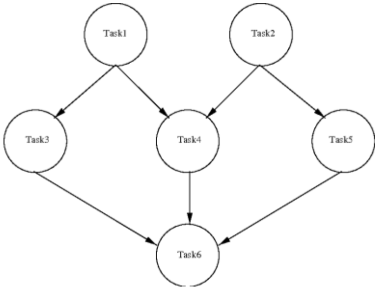


Task parallelism

- Examples of ind'ependent tasks?
    - Appels aux fonctions: **a = f(x); b =**
    - **g(y);** Blocs de code indèpendents dans une m^eme fonction.
    - Multiple execution of a m^eme program with different parameters (i.e. simulation).

Task Parallelism

It is the execution of ind'ependent tasks in a program.

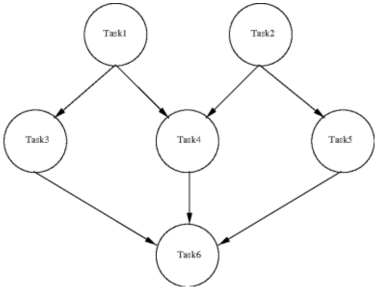


Task parallelism

- Examples of ind'ependent tasks?
  - Appels aux fonctions: **a = f(x); b =**
  - **g(y);** Blocs de code indèpendents dans une mˆeme fonction.
  - Multiple execution of a mˆeme program with different parameters (i.e. simulation).
- Each task can be executed on a separate calculation unit.

Task Parallelism

It is the execution of ind'ependent tasks in a program.

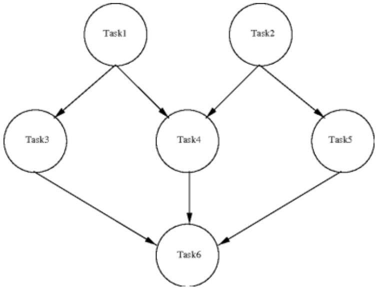


Task parallelism

- Examples of ind'ependent tasks?
  - Appels aux fonctions: **a = f(x); b =**
  - **g(y);** Blocs de code indèpendents dans une mˆeme fonction.
  - Multiple execution of a mˆeme program with different parameters (i.e. simulation).
- Each task can be executed on a separate calculation unit.
- It is necessary to respect the d'ependences if there are any.

Task Parallelism

It is the execution of ind'ependent tasks in a program.



- Examples of ind'ependent tasks?
  - Appels aux fonctions: **a = f(x); b =**
  - **g(y);** Blocs de code indèpendents dans une mˆeme fonction.
  - Multiple execution of a mˆeme program with different parameters (i.e. simulation).
- Each task can be executed on a separate calculation unit.
- It is necessary to respect the d'ependences if there are any.

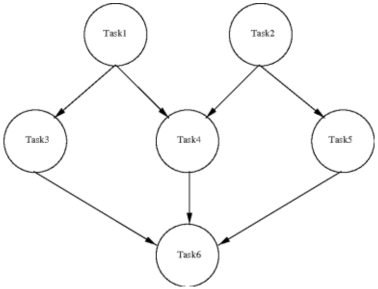
Task Parallelism

- Load balancing?

Task parallelism

Task Parallelism

It is the execution of ind'ependent tasks in a program.



- Exemples des tˆaches ind'ependentes?
  - Appels aux fonctions: **a = f(x); b =**
  - **g(y);** Blocs de code indėpendents dans une mˆeme fonction.
  - Multiple execution of a mˆeme program with different parameters (i.e. simulation).
- Each task can be executed on a separate calculation unit.
- It is necessary to respect the d'ependences if there are any.
  - Load balancing?
  -

Task Parallelism

Task parallelism

A whole field of research (scheduling).

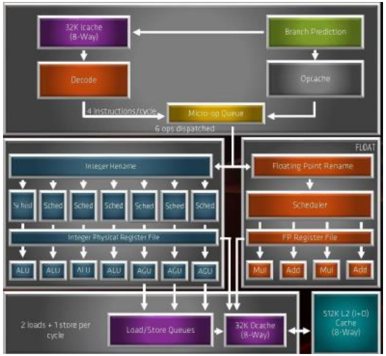
Task parallelism

Outline

Task Parallelism

CPU

"Central Processing Unit, a general computing unit consisting 6

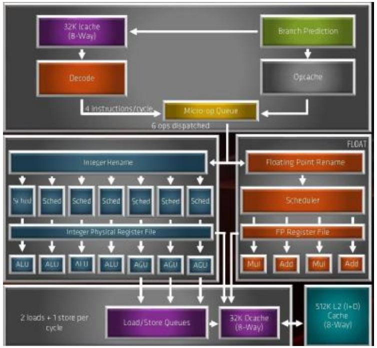


Zen 2 architecture

CPU

"Central Processing Unit, a general computing unit consisting 6

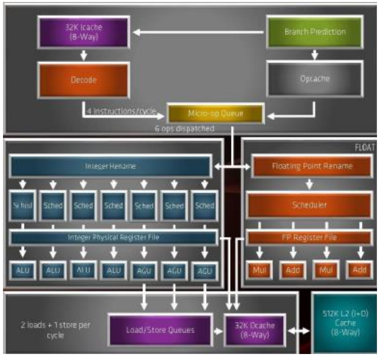
- Several execution units (hearts)



Zen 2 architecture

CPU

"Central Processing Unit, a general computing unit consisting of



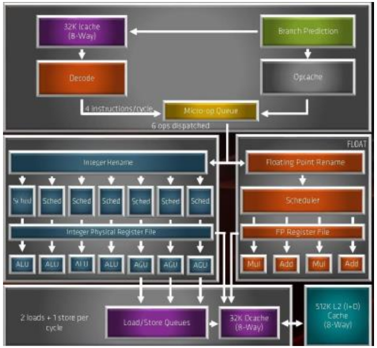
Zen 2 architecture

- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)

CPU

"Central Processing Unit, a general computing unit consisting of

- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)

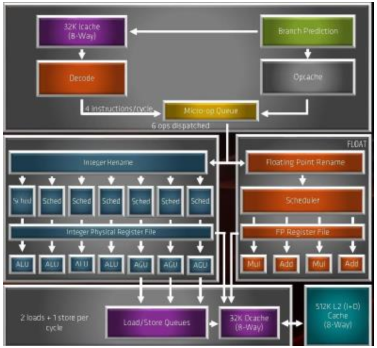


Zen 2 architecture

CPU

"Central Processing Unit, a general computing unit consisting of

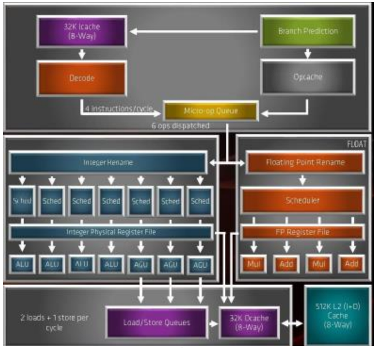
- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)
- Vector units (AVX2, AVX512, Arm Neon, ...)



Zen 2 architecture

CPU

"Central Processing Unit, a general computing unit consisting of

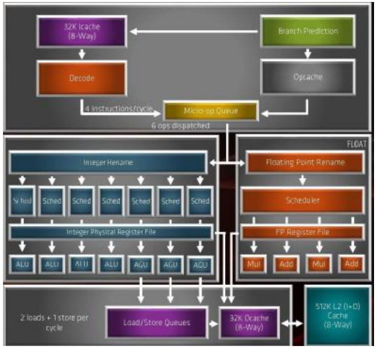


Zen 2 architecture

- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)
- Vector units (AVX2, AVX512, Arm Neon, ...)
- Execution of some threads (1-4) simultaneously

CPU

"Central Processing Unit, a general computing unit consisting of

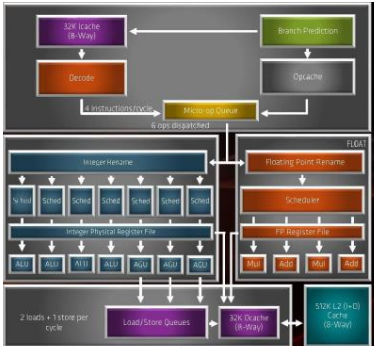


Zen 2 architecture

- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)
- Vector units (AVX2, AVX512, Arm Neon, ...)
- Execution of some threads (1-4) simultaneously
- Able to exploit parallelism at the instruction level (micro-op buffer, instruction renaming, register renaming, ...)

CPU

"Central Processing Unit, a general computing unit consisting of



Zen 2 architecture

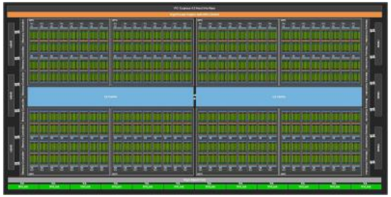
- Several execution units (hearts)
- Multiple memory levels (registers, L1, L2, L3, RAM)
- Several execution ports in each core (ALUs, vector units)
- Vector units (AVX2, AVX512, Arm Neon, ...)
- Execution of some threads (1-4) simultaneously
- Able to exploit parallelism at the instruction level (micro-op buffer, instruction renaming, register renaming, ...)
- A considerable part of the circuit is dedicated to ILP +

GPU

cache

"Graphical Processing Unit", a specific virtual computing unit consisting 6

CPU

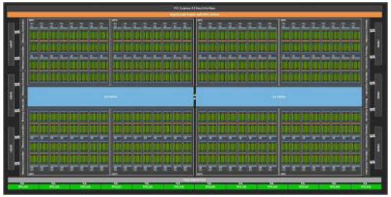


The Nvidia Ampere architecture

GPU

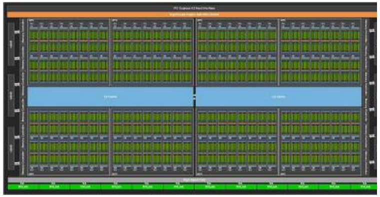
"Graphical Processing Unit", a specific virtual computing unit consisting of

- Multiple execution units (symmetric multiprocessors (SM))



The Nvidia Ampere architecture

GPU

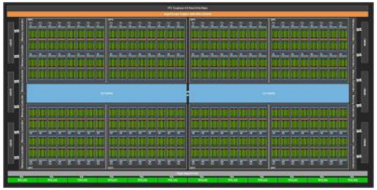


The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting of



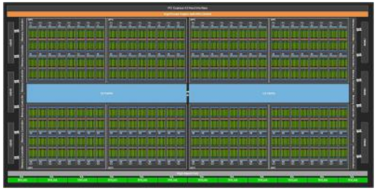
The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting of

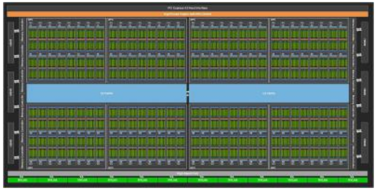
- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of hundreds of simultaneous threads



The Nvidia Ampere architecture

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting ɓ

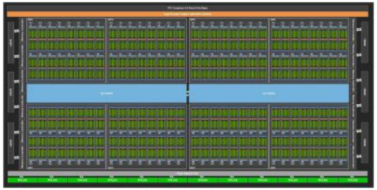


The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of hundreds of simultaneous threads Large
- register array (65K)

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting 6

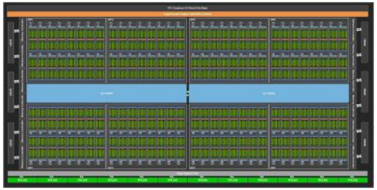


The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of hundreds of simultaneous threads Large
- register array (65K)
- Very fast thread exchange

GPU

"Graphical Processing Unit", a specific virtual computing unit consisting 6

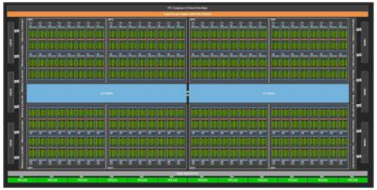


The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of hundreds of simultaneous threads Large
- register array (65K)
- Very fast thread exchange
- Most of the circuit is devoted to vector units

Supercomputer / Cluster

"Graphical Processing Unit", a specific virtual computing unit consisting of



The Nvidia Ampere architecture

- Multiple execution units (symmetric multiprocessors (SM))
- Several memory levels (registers, shared memory, L1, L2, RAM)
- Several large (2-4) vector units (16-32 floats) in each DM
- Execution of hundreds of simultaneous threads Large
- register array (65K)
- Very fast thread exchange
- Most of the circuit is devoted to vector units
- Parall'elisation takes the effort

GPU

A set of machines (CPU+GPU) connected

Jolio Curie supercomputer, 300K CPU cores, 1024 GPUs

Supercomputer / Cluster

A set of machines (CPU+GPU) connected

Jolio Curie supercomputer, 300K CPU cores, 1024 GPUs

- Connection by a network with a particular topology (ring, grid, torus, clique, etc.)

Supercomputer / Cluster

A set of machines (CPU+GPU) connected

Jolio Curie supercomputer, 300K CPU cores, 1024 GPUs

- Connection by a network with a particular topology (ring, grid, torus, clique, etc.)
- Topology-adapted communications libraries

Supercomputer / Cluster

A set of machines (CPU+GPU) connected

Jolio Curie supercomputer, 300K CPU cores, 1024 GPUs

- Connection by a network with a particular topology (ring, grid, torus, clique, etc.)
- Topology-adapted communications libraries
- Able to address very large problems

Supercomputer / Cluster

A set of machines (CPU+GPU) connected

Jolio Curie supercomputer,
300K CPU cores, 1024 GPUs

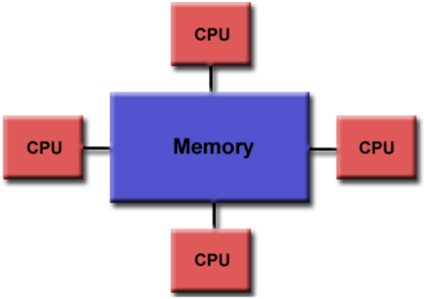
- Connection by a network with a particular topology (ring, grid, torus, clique, etc.)
- Topology-adapted communications libraries
- Able to address very large problems
- Today, at the exaflops scale (^{1018} floating operations per second)

Shared memory: general characteristics

- All processors can access the memory as a global address space
- Plusieurs processeurs peuvent opèrer de fa¸con indèpendante mais partagent les m^emes ressources moire
- Modifications by a processor in amemory area are visible to all other processors

Mèmoire partagèe : Uniform Memory Access (UMA)

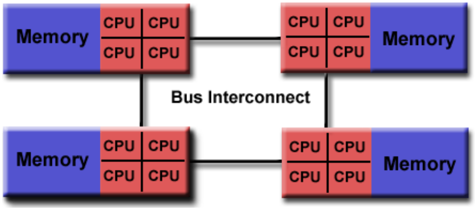
- Presented by SMP (Symmetric Multiprocessor) machines Identical
- processors
- Time of access the memory ègal for all processors



Shared Memory (UMA)

Non-Uniform Memory Access (NUMA)

- In most casesphysically built by linking two or more SMPs An SMP can directly
- access the memory of another SMP
- All processors do not have an equal access time for all memories Memory
- accesses the memory of another SMP are slower
- If the cache coherence is assured, we speak of cc-NUMA



Shared Memory (NUMA)

Mèmoire partagèe:

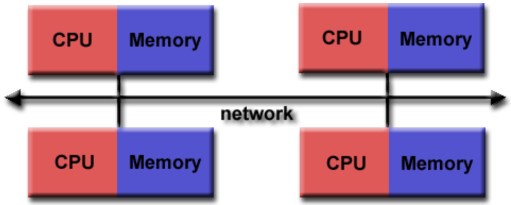
1. Benefits:
   - The global address space allows a simpler programming from the point of view of memory management
   - The sharing of data between threads is fast and uniform gr^ace `a proximit'e de la m'emoire du CPU
2. Inconvènients :
   - Lack of scalability between memory and CPUs: adding more CPUs will increase the use of the shared bus and for systems with a coherent cache, it will increase the effort of managing the coherence between the cache and the memory
   - It is the programmer's responsibility to make the necessary synchronizations to ensure "correct" accesses to the global memory

Distributed memory: general characteristics

- Distributed memory ysdo not require a network to ensure the connection between the processors
- Each processor has its own memory and address space
- It is up to the programmer to explicitly indicate how and when the data should ˆetre communicated and when the synchronizations between the precessors should ˆetre effectu'ees

Distributed memory: general characteristics

- The network used to transfer data between processors is very varied, it can be as simple as Ethernet

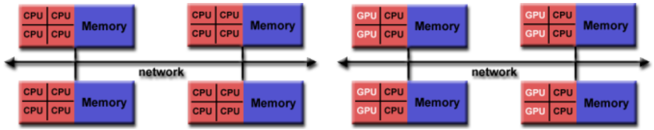


Mémoire distribuée

1. Benefits:
   - Memory is scalable with the number of processors
   - Chaque processeur acc`ede `a m emoire rapidement sans ni interference avec les autres processeurs ni de coût additionnel pour maintenir une coh'erence globale du cache
2. Inconvénients :
   - The programmer is responsible for several details associated with data communication between processors
   - Il peut ˆetre difficile de distribuer des structures de donn'ees conc¸ues sur une base de m'emoire globale sur cette nouvelle organisation m'emoire

Hybrid memory: general characteristics

- The most efficient machines in the world are machines that use shared and distributed memories
- Les composantes àm'emoire partagèe peuvent ˆetre des machines àmèmoire partagèe ou des GPUs (graphics processing units)
- The processors of a computing node share the same memory space n'ecessitent
- communications to èchanger les données entre les noeuds



Outline

Parallel programming models

Parallel programming models exist as an abstraction on top of parallel architectures

1. Mèmoire partagèe:
   - **Intrinsics** SIMD instructions (Intel SSE2, ARM NEON), low level (**More details in upcoming courses**)
   - Posix Threads library
   - **OpenMP** is based on compiler directives to be played in a sequential code (**More details in the next courses**)
   - CUDA (**More details in upcoming courses**) OpenCL
2. Mèmoire distribuèe:
   - Library sockets, low level
   - **MPI** Message Passing Interface the standard for distributed memory architectures, the parallel code is generally very different from the serial code (**More details in the next courses**)

Parallel programming models

Parallel programming models exist as an abstraction on top of parallel architectures

1. Memoire partagee:
   - **Intrinsics** SIMD instructions (Intel SSE2, ARM NEON), low level (**More details in upcoming courses**)
   - Posix Threads library
   - **OpenMP** is based on compiler directives to be played in a sequential code (**More details in the next courses**)
   - CUDA (**More details in upcoming courses**) OpenCL
2. Memoire distribuee:
   - Library sockets, low level
   - **MPI** Message Passing Interface the standard for distributed memory architectures, the parallel code is generally very different from the serial code (**more than**

More details in the next courses)

What kind of pro ramming used?

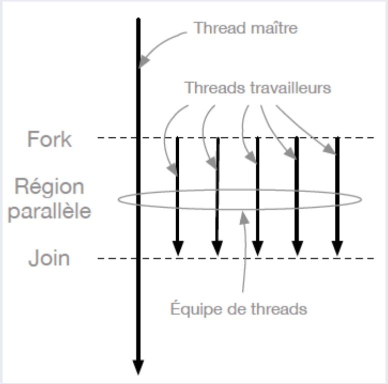
université PARIS-SACLAY

Thread model

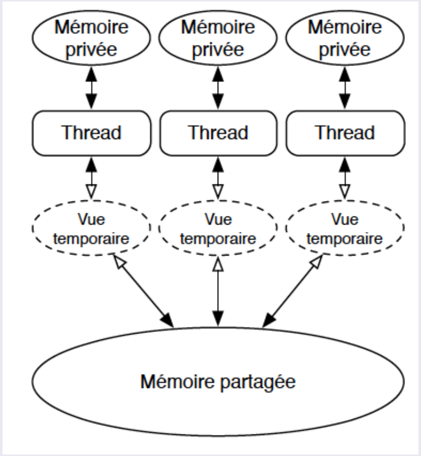
- IEEE POSIX Threads (PThreads)
  - A Standard UNIX API, also exists under Windows
  - More than 60 functions: *pthread create*, *pthread join*, *pthread exit*, ...
- OpenMP
  - A higher level interface, based on
    - compiler directives library
    - functions a runtime
  - An orientation towards high performance computing (HPC) applications

OpenMP

Fork-join execution model



Memory model



Message Passing Interface

- Specification and management by the MPI forum
  - The library provides a set of communication primitives: point-to-point or collective
  - C/C++ and Fortran
- A low-level programming model
  - data distribution and communications must be done manually Primitives are easy to
  - use but the development of parallel programs can be quite difficult
- Communications
  - Point to point (messages between two processors)
  - Collective (messages in groups of processors)

Scalar product : Sequentiel

```
#include <stdio.h>
#define SIZE 256

int main() {
   double sum, a[SIZE], b[SIZE];

// Initialization
   sum = 0;
   for (size_t i = 0; i < SIZE; i++) {
      a[i] = i * 0.5;
      b[i] = i * 2.0;
   }

   // Computation
   for (size_t i = 0; i < SIZE; i++) sum =
      sum + a[i]*b[i];

   printf("sum = %g\n", sum);
   return 0;
}
```

Scalar product: SSE instructions

```
# include < immintrin.h>
# include < iostream >
# include < algorithm
> # include < numeric>

int main ()
{
  std::size_t const size = 4 * 5; s
  td::srand(time (nullptr));
  float *array0 = static_cast < float *  > ( mm_malloc ( size *  sizeof ( float ) , 16 ) ); float
  *array1 = static_cast_< float *  > ( mm_malloc ( size *  sizeof ( float ) , 16 ) ); std::ge
  nerate n(array0, size, []() { return std::rand ()%10; } );
  std::generate n(array1, size, []() { return std::rand ()%10; } );

  auto r0 = mm_mul_ps (_mm_load_ps ( &array0[0]),          _mm_load_ps ( &array1[0]);

  for ( std::size_t i = 0; i < size; i+=4 )
  {
    r0 = mm_add_ps ( r0, mm_mul_ps ( mm_load_ps ( &array0[i]), mm_load_ps ( &array1[i])));

  }

  float tmp [4] attribute((aligned(16)));
  _mm_store_ps ( tmp, r0 );
  auto res = std::accumulate ( tmp, tmp + 4, 0.0f );

   mm_free ( array0 );
```

Scalar product : Pthreads

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 256
#define NUM_THREADS 4
#define CHUNK SIZE/NUM_THREADS

int id[NUM_THREADS];
double sum, a[SIZE], b[SIZE]; pt
hread_t tid[NUM_THREADS];
pthread_mutex_t mutex_sum;

void *dot(void *id) {
   size_t i;
   int my_first = *(int *)id * CHUNK;
   int my_last = (*(int *)id + 1) * CHUNK;
   double sum_local = 0;

   // Computation
   for (i = my_first; i < my_last; i++) su
      m_local = sum_local + a[i]*b[i];

   pthread_mutex_lock(&mutex_sum);
   sum = sum + sum_local;
   pthread_mutex_unlock(&mutex_sum);
   return NULL;
}
```

```c
int main() {
   size_t i;

  // Initialization
  sum = 0;
  for (i = 0; i < SIZE; i++) {
     a[i] = i * 0.5;
     b[i] = i * 2.0;
  }

  pthread_mutex_init(&mutex_sum, NULL);

  for (i = 0; i < NUM_THREADS; i++) {
     id[i] = i;
     pthread_create(&tid[i], NULL, dot,
                    (void *)&id[i]);
  }

  for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);

  pthread_mutex_destroy(&mutex

  sum); printf("sum␣=␣%g\n", sum);
  return 0;
}
```

Scalar product : OpenMP

```
# include < s t d i o . h>
# define SIZE 256

int main () {
   double sum , a [ SIZE ] , b [ SIZE ] ;

// Initialization
   sum = 0 ;
   for ( s i z e t i = 0 ; i < SIZE ; i ++) {
      a [ i ] = i  ∗  0 . 5 ;
      b [ i ] = i  ∗  2 . 0 ;
   }

   // Computation
   #pragma omp p a r a l l e l for r e d u c t i o n (+: sum )
   for ( s i z e t i = 0 ; i < SIZE ; i ++) {
      sum = sum + a [ i ] ∗b [ i ] ;
   }

   p r i n t f ( " sum u=u% g\ n" , sum ) ;
   return 0 ;
}
```

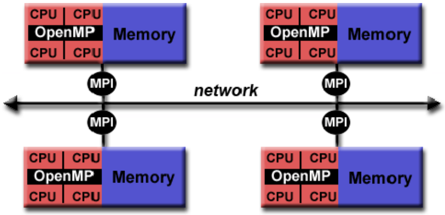
Scalar product : MPI

```
# include < s t d i o . h>
# include " mpi. h" #
define SIZE 256

int main ( int argc , char∗ a r g v [ ] ) {
   int numprocs , my rank , m y f i r s t , m y l a s t ;
   double sum , s u m l o c a l , a [ SIZE ] , b [ SIZE ] ; M PI
   I n i t (& argc , &a r g v ) ;
   MPI Comm size (MPI COMM WORLD, &numprocs );
   MPI Comm rank (MPI COMM WORLD, &my rank ); m y
   f i r s t = my rank ∗   SIZE/ numprocs ;
   m y l a s t = ( my rank + 1 )  ∗   SIZE/ numprocs ;

   // Initialization
   s u m l o c a l = 0 . ;
   for ( s i z e t i = 0 ; i < SIZE ; i ++) {
      a [ i ] = i  ∗   0 . 5 ;
      b [ i ] = i  ∗   2 . 0 ;
   }

   // Computation
   for ( s i z e t i = m y f i r s t ; i < m y l a s t ; i ++) s u
      m l o c a l = s u m l o c a l + a [ i ] ∗b [ i ] ;
   M PI A l l r e d u c e (& s u m l o c a l , &sum , 1 , MPI DOUBLE , MPI SUM , MPI COMM WORLD ) ;

   if ( my rank == 0 )
      p r i n t f ( " sum u=u% g\ n" , sum ) ;
```

Hybrid model

- Several MPI processes, each managing a number of threads
  - Inter-process communication via message sending (MPI)
  - Intra-process (thread) communication via shared memory

- Well suited to hybrid architectures
  - one process per node
  - one thread per core



Outline

Plan

7 Around the development of parallel programs

A performance model; the roofline model

La performance qu'on peut atteindre (Gflop/s) est bornée par

$$\min \begin{cases} \text{The performance crête of the machine,} \\ \text{The maximum bandwidth} \times \text{operational intensity} \end{cases}$$

where the operational intensity is the number of float operations performed per byte of DRAM transferred

A performance model; the roofline model

La performance qu'on peut atteindre (Gflop/s) est bornée par

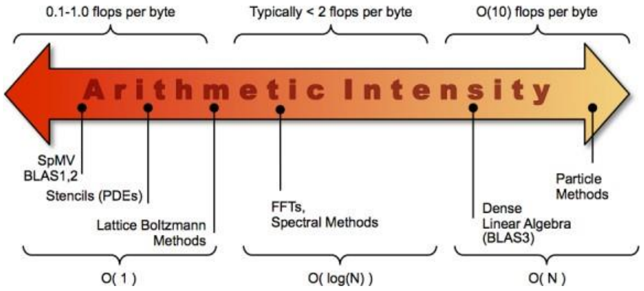
The performance crête of the machine,

The maximum bandwidth × operational intensity ,

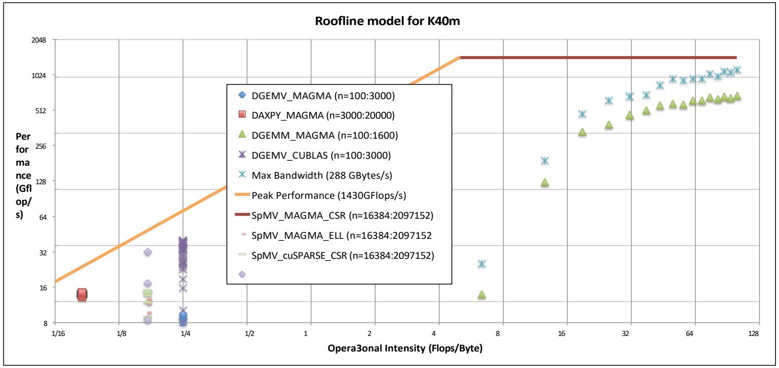
wh the operational intensity is the number of float operations performed per byte of DRAM transferred

- depends on the algorithm and the target architecture
- dense matrix-vector product: $I_{dgemv} \leq \frac{1}{4}$
- Hollow matrix-vector product: $I_{SpMV} \leq \frac{1}{6}$, depending on the storage format (CSR here)

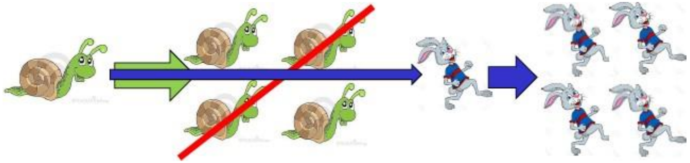
the roofline model: operational intensity



Performance model for NVIDIA Tesla K40



Wtart? : Core Optimization



Performance at the heart

- Reducing cache errors: blocking, tilling, loop ordering, ... Vectorization
- (SSE/AVX units)

What to do next?

- Identify program bottlenecks:
  - to know which parts consume the most running time
  - performance analysis tools can help here (profilers . . .) Focus on
  - parallelizing bottlenecks
- Re-structure the program or use/develop another algorithm to reduce the parts that are very slow
- Use existing: optimized parallel software and libraries (IBM's ESSL, Intel's MKL, AMD's AMCL, LAPACK, C++ Parallel STD, . . .)

universite PARIS-SACLAY

Qu'est ce qui doit ˆetre considèrè? : quelques èlèments

- Data distribution: 1D, 2D, block, block cyclic, tiles ... Granularity
- Communications
- Synchronization
- Overlapping of calculations and communications Load
- balancing between threads and/or processors
- . . .

université PARIS-SACLAY

References

Contact

Oguz Kaya
Université Paris-Saclay and LRI, Paris, France
oguz.kaya@lri.com
www.oguzkaya.com