Remove Watermark Wondershare PDFelement

Ministry of Higher Education and Scientific
Research University Mustapha Stambouli of
Mascara
Faculty of Exact Sciences

Lecture Notes

High Performance Computing

–HPC–

Wondershare PDFelement

Auteur: Yahyaoui Khadidja

Professeure en informatique

Université de Mascara

Academic year: 2025/2026

Remove Watermark Wondershare PDFelement

Preamble

This course compendium or lecture notes has been designed to provide a clear, structured, and progressive presentation of the fundamental concepts covered in this module. It is primarily intended for students seeking to acquire a solid understanding of both the theoretical concepts and their practical applications.

To facilitate learning and enhance understanding, a consistent pedagogical structure has been adopted throughout this document. Each chapter is designed to guide the reader progressively, highlighting the key concepts that need to be retained.

Thus, each chapter begins with a **summary** presenting the main objectives and key ideas that will be developed. This introduction allows the reader to gain an overall view of the content before addressing the details.

A list of **keywords** is also provided at the beginning of each chapter. It brings together the fundamental terms and concepts to be mastered, serving as an effective tool for memorization and revision.

The chapter content is then presented in detail, accompanied, when necessary, by illustrative examples, explanatory diagrams, and practical applications, in order to enhance the understanding of the concepts covered.

Finally, some chapters may include exercises or applications that allow the reader to put the acquired knowledge into practice.

This pedagogical approach aims to improve the reader's orientation, facilitate the assimilation of concepts, and make learning more effective and structured.

Remove Watermark Wondershare PDFelement

Table of Contents

Preamble i

Table of Contents ii

General Introduction 7

Chapter 1: Introduction to Parallel Computing

1. Introduction 9

2. Need for Parallel Computing 10

3. Types of Parallelism 10

3.1 Data Parallelism 10

3.2 Task Parallelism 11

3.3. Task pipelining 11

3.4. Comparison Between Types of Parallelism 12

4. Parallel and Distributed System Architectures for High Performance Computing 12

4.1. Parallel and Distributed System Architectures: Flynn's Classification 12

4.2. Another Advanced Classification of Parallel Architectures 15

5. Current Trends in Parallel Computing Architectures 17

5.1. Multicore and Heterogeneous Architectures 17

5.2. Computing Accelerators 17

5.3. Energy-Efficient and Exascale Computing – Exascale – 19

6. Examples of Applications Requiring Parallel Computing 20

7. Exercises 20

Chapter 2: Fundamental Laws of Parallel Computing

1. Introduction 23

2. Theoretical Models of Performance and Scalability 24

Wondershare PDFelement

Remove Watermark Wondershare PDFelement

2.1. Amdahl's Law and Performance 24

2.2. Gustafson's Law and Scalability 27

3. Granularity and Parallelism 28

3.1. Definition 28

3.2. Types of Granularity 28

3.3. Impact of Granularity on the Performance of Parallel Systems 30

3.4 Impact of communication time on speedup 31

4. Exercises 33

Chapter 3: Application Model and Structure for Parallel Programming Interfaces

1. Introduction 36

2. Paradigms for Parallel Applications 37

3. Shared Memory Model 37

4. Distributed Memory Model 38

5. Hybrid Model 38

6. Parallel Programming Interfaces 39

6.1. Definition of Parallel Programming Interfaces 39

6.2. Keys features 39

6.3. Examples of Parallel Programming Interfaces 40

7. OpenMP Interface 40

7.1 Introduction to OpenMP 40

7.2 Operating Principle of OpenMP and the fork-join model 41

7.3. Shared Memory Programming Model 41

7.4. Main Directives, Thread Management, and Mechanisms Synchronization 42

8. Message Passing Interface (MPI) 45

8.1. Introduction to MPI 45

8.2. Objectives of MPI 46

Wondershare PDFelement

Remove Watermark Wondershare PDFelement

8.3. Distributed Memory Programming Model 46
8.4. Main Functions of MPI 47
8.5. Message Passing (MPI) 48
9. Hybrid Interface (MPI + OpenMP) 51
9.1 Presentation of the Hybrid Model 51
9.2 Architecture of Hybrid Systems 51
9.3 Operating Principle 52
9.4 Functionality of the MPI-OpenMP Hybrid Model 53
9.5 Example 54
9.6. Comparison of Parallel Programming Models or Paradigms 55
10. Exercises 56

Chapter 4: Generic Computing on a Graphics Processing Unit (GPU)

1. Introduction 59
2. GPU Architectures and Concepts (Graphics Processing Unit) 60
2.1 Introduction of GPUs 60
2.2 Types of GPUs 61
2.3 Comparison of CPUs and External GPUs 62
2.4 Components of a GPU 63
3. GPU Programming Model 65
3.1 Principles of Generic Computing on GPUs (GPGPU) 65
3.2. GPU Programming Architecture (Threads, Blocks, Grid) for GPU Programming 66
3.3. GPU Memory Hierarchy 68
4. Programming GPUs with the CUDA Interface 70
4.1. Presentation of CUDA 70
4.2. CUDA Architecture 70
4.3. CUDA Programming Workflow 70
5. GPU Programming with the OpenCL Interface 73

Wondershare PDFelement

Remove Watermark Wondershare PDFelement

5.1. OpenCL presentation 73

5.2. OpenCL Programming Model 73

A. Principle of Parallel Computing 73

B. Work Items – Work Groups – NDRange Organization 73

C. Executing an OpenCL Program 74

D. OpenCL Program Example 75

6. Exercises 77

General Conclusion 79

Bibliographical references 80

Wondershare PDFelement

Remove Watermark Wondershare PDFelement

General Introduction

High Performance Computing (HPC) is now a central component of modern technological progress, enabling efficient processing of massive datasets and the execution of computation-intensive workloads. With the rapid growth of data and the increasing complexity of numerical models particularly in areas such as artificial intelligence, deep learning, and scientific simulation. The need for advanced parallel and distributed computing techniques has become essential. In this context, HPC serves as a key enabling technology to accelerate computations, optimize system performance, and address the requirements of large-scale, data-driven applications across scientific and industrial domains.

In this context, this course provides a solid foundation in High Performance Computing (HPC) and highlights its essential role in accelerating scientific computing, processing large amounts of data and developing advanced applications, particularly in fields such as Artificial Intelligence.

Targeted Learning Objectives:

- Gain a solid understanding of parallel programming paradigms and their role in modern computing systems.
- Study fundamental techniques for developing High Performance Computing (HPC) applications.
- Explore commonly used HPC platforms and execution environments.
- Understand methods for measuring, evaluating, and analyzing the performance and efficiency of HPC applications.

This course provides a structured and comprehensive introduction to High Performance Computing (HPC), highlighting its fundamental role in accelerating scientific computation and enabling large-scale data processing. It presents key principles of performance and scalability, along with core parallel programming models implemented through dedicated interfaces, including the GPGPU paradigm. The course also explores the utilization of modern computing accelerators such as CPUs and GPUs. The chapters are organized progressively to build a solid theoretical foundation, supported by illustrative examples and practical implementation cases.

The course notes are structured as follows:

Chapter 1_ Introduction to Parallel Computing: This chapter introduces the fundamental concepts of parallel computing, highlighting its motivations, underlying architectures, and application domains. It provides a solid foundation for the study of parallel programming techniques and computational models.

Remove Watermark Wondershare PDFelement

Chapter 2 _Fundamental Laws of Parallel Computing**:** This chapter focuses on the importance of fundamental laws of performance and scalability, such as Amdahl's Law and Gustafson's Law, in parallel computing, particularly in the presence of sequential parts or communication overhead between processes. It is therefore essential to understand both the advantages and the limitations of parallel computing.

Chapter 3_ Application Models and Structures for Parallel Programming Interfaces**:**This chapter presents application models and structures for parallel programming interfaces, including parallel programming paradigms such as OpenMP, multithreading, and MPI.

Chapter 4_ General-Purpose Computing on Graphics Processing Units (GPU): This chapter is dedicated to the new programming paradigm based on graphics processors, known as GPGPU (General-Purpose computing on Graphics Processing Units). It explains how the massive computational power of graphics cards can be exploited to perform a large number of operations in parallel, making them particularly suitable for fields such as artificial intelligence, scientific simulation, and large-scale data processing.

At the end of this lecture notes, students will have acquired a comprehensive and in-depth understanding of the main components of High Performance Computing (HPC), enabling them to explore advanced topics or apply these concepts in scientific research, numerical simulation, and large-scale industrial applications.

Remove Watermark Wondershare PDFelement



Chapter 1: Introduction to parallel computing

Key words

Parallel computing; Flynn Architecture; Novel Architecture; HPC Application

ABSTRACT

Parallel computing involves decomposing a problem into multiple tasks executed simultaneously on different computing units to improve performance and data processing. High-performance computing (HPC) relies on various architectures, ranging from shared-memory systems to clusters and supercomputers. Modern architectures increasingly incorporate accelerators such as GPUs and TPUs, used in fields like artificial intelligence and numerical simulation. This chapter presents the fundamentals of parallel computing, its architectures, and its main applications.

1. Introduction

The principle of parallel computing is based on breaking a problem down into several subtasks that are executed simultaneously on different processing units. This approach makes it possible to make effective use of modern architectures comprising multi-core processors and specialised accelerators. Understanding the need for parallel computing is therefore essential to meeting today's demands for performance and data processing.

Furthermore, current trends in hardware architecture highlight the growing use of accelerators such as GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units), which are designed to efficiently process massively parallel workloads. These architectures play a key role in fields such as artificial intelligence, deep learning, numerical simulation and other applications requiring high computing power.

This chapter therefore introduces the fundamental concepts of parallel computing, highlighting its rationale, architectures and areas of application,

Remove Watermark Wondershare PDFelement

in order to provide a solid foundation for the study of parallel programming techniques and models.

2. The need for parallel Computing

Parallel computing has become an essential requirement in modern computing. Indeed, the massive increase in data volumes and the growing complexity of applications demand fast and efficient processing times. Parallel computing allows a task to be divided into several subtasks that are executed simultaneously across multiple processors or cores, thereby significantly reducing execution time. It is particularly necessary for:

- Accelerate computationally intensive tasks, such as numerical simulation, artificial intelligence or scientific modelling.
- To make effective use of the multi-core architectures of modern processors, whose performance now relies more on parallelism than on increased clock speeds, whilst adapting to new developments in computing architectures.
- Process large volumes of data (Big Data) that cannot be analysed efficiently in a sequential manner.
- Improve performance and energy efficiency, as parallel processing often allows for better utilisation of hardware resources.

3. Types of parallelism

In parallel systems, there are three types of parallelism: data parallelism, task parallelism and pipelining.

3.1 Data parallelism

Principle : Data parallelism involves applying the same operation to a large dataset simultaneously, by distributing the data across multiple cores or processors. The algorithm remains unchanged; only the data is divided.

Example : We want to calculate the sum of 1,000 numbers.

- CPU1 calculates the sum of the numbers from 1 to 250
- CPU2 calculates the sum of the numbers from 251 to 500
- CPU3 calculates the sum of the numbers from 501 to 750
- CPU4 calculates the sum of the numbers from 751 to 1000

Each processor works simultaneously on a different portion of the array.

3.2 Task parallelism

Remove Watermark Wondershare PDFelement

Principle : Parallelism is a parallel computing technique that involves simultaneously executing several different tasks or functions on multiple processors to accelerate the overall execution of a program.

Example : In a scientific application, we want to process data: reading, calculating, and writing.

• Task 1 (CPU1): reading data

• Task 2 (CPU2): processing/calculating data

• Task 3 (CPU3): writing results

The three tasks run in parallel, which reduces the overall processing time.

3.3. Parallélisme en pipeline

Principle : Pipelined parallelism is a parallel processing technique where the processing of a task is divided into several successive steps, each executed by a different processor or thread. Data flows sequentially from one step to the next, much like in an industrial production line. Thus, while one processor executes step 1 on a piece of data, another processor can simultaneously execute step 2 on the previous data, and a third processor can perform step 3 on other data that has already been processed.

Example: Images are processed in three steps: reading, filtering, and writing.

- CPU1 reads the first image (Step 1)
- CPU2 filters the first image (Step 2)
- CPU3 writes the first image (Step 3)

Mean while:

- CPU1 reads the second image
- CPU2 filters the first image
- CPU3 writes the first image

Thus, several images are processed simultaneously, each at a different step.
Time →
t1: CPU1 processes D1 (Step 1)
t2: CPU1 processes D2, CPU2 processes D1 (Step 2)
t3: CPU1 processes D3, CPU2 processes D2, CPU3 processes D1 (Step 3)
t4: CPU1 processes D4, CPU2 processes D3, CPU3 processes D2
...

3.4. Comparison between types of parallelism

Remove Watermark Wondershare PDFelement

A comparative analysis of the three types of parallelism is provided through a structured table, enabling a clearer distinction of their characteristics and facilitating a deeper understanding of these concepts.

Type of Parallelism	Principle	Example	Main Advantages
Data Parallelism	Applying the same operation simultaneously to different parts of the data	Computing the sum of an array by distributing elements across multiple CPUs	Reduces overall computation time; easy to implement for independent data
Task Parallelism	Executing multiple different tasks in parallel	Reading, processing, and writing data on different CPUs	Enables concurrent execution of different tasks; well-suited for multitasking programs
Pipeline Parallelism	Dividing processing into successive stages and handling multiple data items simultaneously at different stages	Image processing: reading (CPU1), filtering (CPU2), writing (CPU3)	Increases overall system throughput; ideal for sequential processes divided into stages

4. Parallel and Distributed System Architectures for High Performance Computing

Ces architectures sont présentées par la classification de FLYNN.

4.1. Parallel and Distributed System Architectures: Flynn's Classification

Flynn's classification, proposed by Michael J. Flynn in 1966, is a system that categorises computer architectures according to the number of instruction streams and data streams they can process simultaneously. It is particularly useful for understanding the different types of parallelism in computer systems. The following table summarises the categories of Flynn's architectures

Remove Watermark Wondershare PDFelement

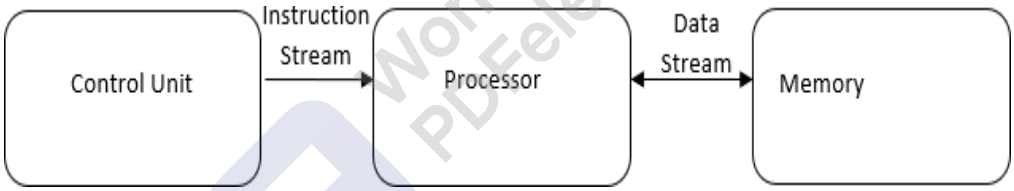
		Flux des données (D)	
		Unique (S : single)	Multiple (M)
Flux des instructions (S)	Unique (S : single)	SISD	SIMD
	Multiple (M)	MISD	MIMD

FLYNN Classification

FLYNN classification is descrited as follows :

SISD : Single Instruction stream-Single Data Stream

This is a classic sequential architecture. A typical computer with a single processor is an example of this architecture. It relies on executing only one instruction at a time on a single piece of data. Sequential processing, as seen in older personal computers or microcontrollers, is an example of its use.

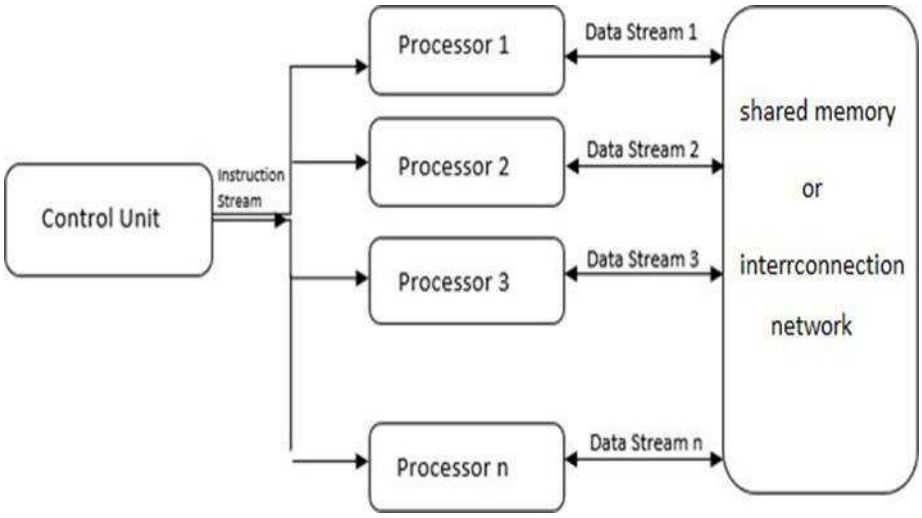


Architecture Single-instruction single-data streams (SISD)

SIMD : Single Instruction-Multiple Data streams

Data parallelism consists of applying the same instruction simultaneously to multiple data elements. It relies on distributing data across several processing units, each executing an identical operation in parallel. This model is commonly implemented in architectures such as Graphics Processing Units (GPUs) and vector processors. Data parallelism is widely used in domains including video games, image processing, and artificial intelligence, where large datasets can be processed concurrently. It is also applicable to mathematical computations such as polynomial evaluation.

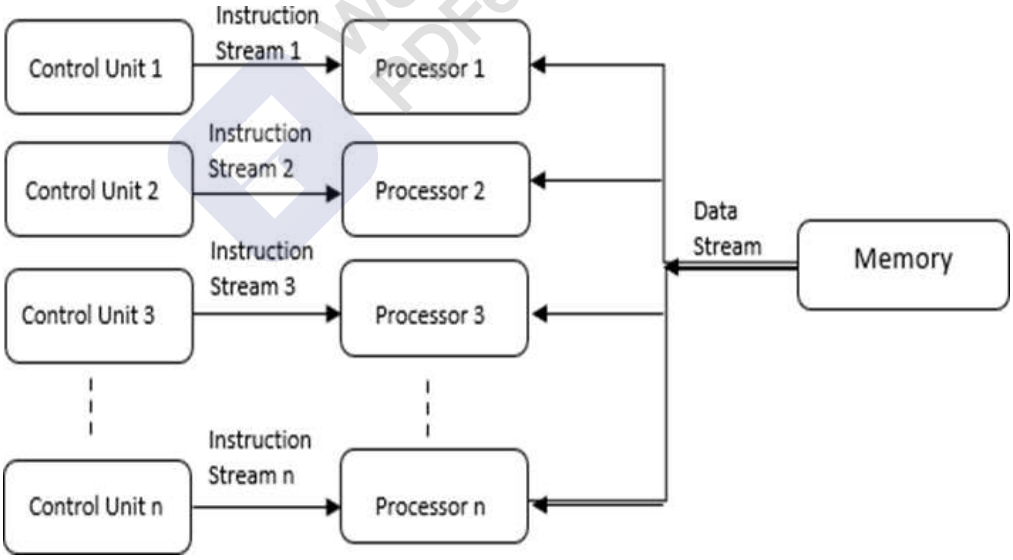
Remove Watermark Wondershare PDFelement



Architecture Single-instruction Multiple-data streams (SIMD)

MISD : Multiple Instruction-Single Data streams.

Multiple instructions executed on the same data (rarely used). Its operation is based on the execution of several instructions on the same data. Redundant control systems in aeronautics or nuclear power plants are an example of this architecture. Its use is very rare; it is found for fault tolerance or cross-checking.



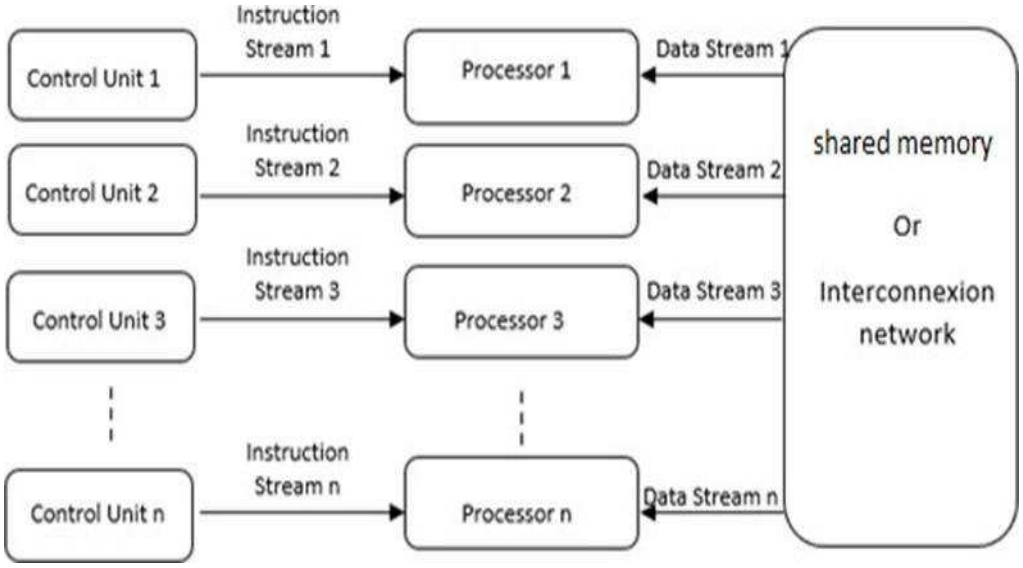
Architecture Muliple-instruction Single-data Stream (MISD)

MIMD : Multiple Instructions-Multiple data Stream

A very common architectural model in modern parallel systems. Its operation is based on each processor executing different instructions on different data. Examples of this architecture include supercomputers, multicore servers, and

Remove Watermark Wondershare PDFelement

PC clusters. They are used in scientific computing, simulation, and cloud computing.



Architecture Multiple-instruction Multiple-Data Stream (MIMD)

From the MIMD category, the following follows:

- **SM-MIMD** : (Shared Memory-Multiple Instructions Multiple data) Systèmes multicœurs avec mémoire partagée (exemple : Open MP sur CPU).

- **DM-MIMD :** (Distributed Memory -Multiple Instructions Multiple Data) clusters ou supercalculateurs avec MPI.
- **VSM-MIMD** : (**Virtual Shared Memory** - Multiple Instructions Multiple data). Environnements distribués avec abstraction logicielle (exemple : PGAS (Partitioned Global Address Space), UPC (Unified Parallel C)

4.2. Another advanced parallel architecture classification

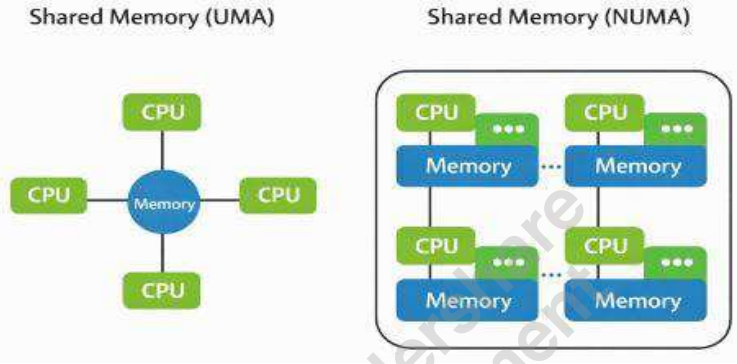
Flynn's classification, however, suffers from certain limitations. The MIMD class, for example, encompasses a wide variety of architectures. For this reason, a more advanced classification of these machines was proposed by Johnson, based on their memory organization (shared or distributed) and the mechanism used for communication/synchronization (shared variables or message passing). This classification is as follows:

- GMSV (*Global Memory, Shared Variables*):
- represents traditional shared-memory multiprocessors, also known as symmetric multiprocessor (SMP) systems. All cores share memory and

Remove Watermark Wondershare PDFelement

input/output (IO) equally and can access the same memory location at the same speed, thus providing uniform memory access (UMA) (see diagram below).

- DMSV (Distributed Memory, Shared Variables) : refers to systems where memory is physically distributed across nodes (or groups of cores), but all these cores share the same memory in parallel, enabling access to both local data (within the nodes) and remote data (in the shared memory of other nodes). The memory is therefore closer to certain cores, which can then access certain memory locations more quickly than others, providing non-uniform memory access: these are known as NUMA (Non-Uniform Memory Access) systems (see diagram below),



Shared Memory Architectures (UMA and NUMA)

The following table provides a comparison between the two types of architectures based on well-defined important criteria.

Aspect	UMA (Uniform Memory Access)	NUMA (Non-Uniform Memory Access)
Memory Access	Uniform access for all CPUs	Access depends on local or remote memory
Memory Organization	Single shared memory	Local memory per CPU + shared memory
Access Time	Identical for all CPUs	Variable depending on memory location
Scalability	Low scalability	High scalability
Programming Complexity	Simple programming model	More complex programming model
HPC Usage	Limited use in High-Performance Computing	Very common in High-Performance Computing

Comparison between Shared memory architectures (UMA and NUMA)

5. Current trends in parallel computing architectures

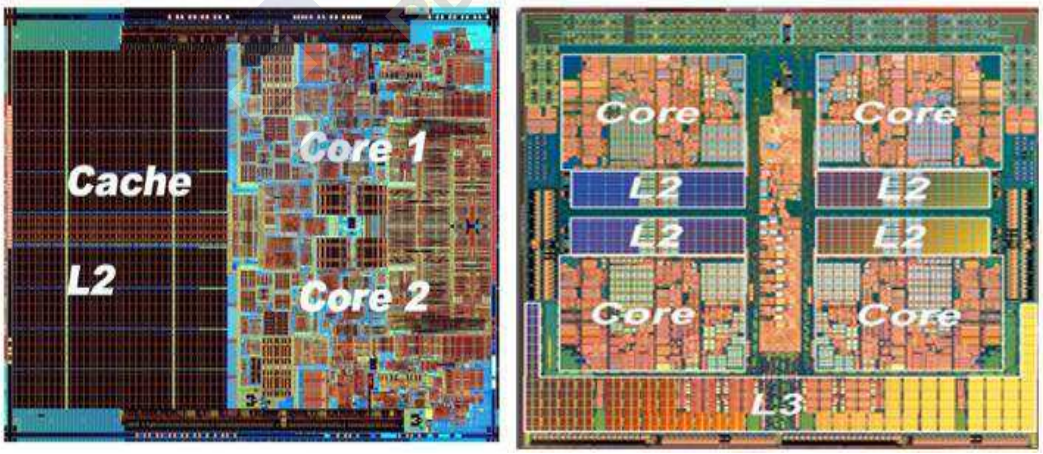
Remove Watermark Wondershare PDFelement

Parallel computing is now at the heart of modern computing: from artificial intelligence to the cloud, and from scientific supercomputers to Big Data platforms. Advances in hardware (GPUs: Graphics Processing Units, TPUs: Tensor Processing Units) and software (languages and frameworks) are making this field both more powerful and more accessible, whilst raising new challenges in terms of scalability, security and energy efficiency.

Hardware architectures are evolving rapidly to meet the growing demands of HPC, artificial intelligence, Big Data and scientific simulations.
Today, performance no longer depends solely on a processor's clock speed, but above all on massive parallelism and hardware specialisation.

5.1. Multi-core and heterogeneous architectures

Multi-core architectures are modern CPUs that integrate several dozen cores onto a single chip. They are characterised by the simultaneous execution of multiple threads/processes, and they improve throughput, responsiveness and energy consumption per task. Heterogeneous architectures combine several types of computing units, namely CPUs for control and general-purpose tasks, GPUs for massive parallel computing, and specialised accelerators for artificial intelligence algorithms or encryption. The aim is to use the right processor for the right task in order to increase performance and energy efficiency.



Dual-core d'AMD **Quad-Core d'AMD**

Internal Architecture of AMD Multi-Core Processor (Advanced Micro Devices)

5.2. Computational accelerators

GPUs (Graphics Processing Units) are chips comprising thousands of small, simple cores. They are recommended for deep learning, matrix computations, scientific simulations and image/video processing. Their advantages lie in their very high computational throughput and their ability to execute a large

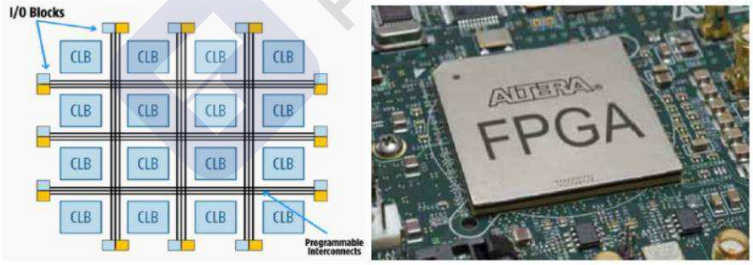
Remove Watermark Wondershare PDFelement

number of threads simultaneously. GPUs are ideally suited for use in HPC centres, generative intelligence and scientific research.



Graphic Processor GeForce de Nvidia

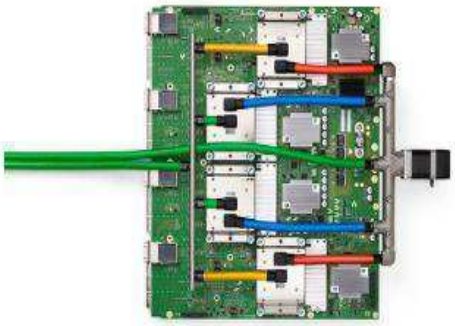
FPGA (Field-Programmable Gate Array) is a reconfigurable hardware device designed to create a circuit adapted to a specific algorithm. It is typically used in the fields of telecommunications, high-frequency finance, edge computing and customised acceleration.



Accelerators : : FPGA

TPUs (Tensor Processing Unit) are application-specific integrated circuits (ASICs) designed by Google to accelerate machine learning workloads. TPUs are designed to perform matrix operations quickly, making them ideal for machine learning workloads. You can run machine learning workloads on TPUs using frameworks such as PyTorch and JAX.

Remove Watermark Wondershare PDFelement



A chip tray used for TPUv4



Google's Oklahoma data center

ASIC (Application Specific Integrated Circuit) is an integrated circuit designed specifically for a particular application or function, as opposed to general-purpose processors.



ASIC chips on a PCB (Printed Circuit Board)

5.3. Energy-efficient computing and exascale

Energy efficiency and exascale are not types of hardware architectures like multi-core CPUs, GPUs or TPUs; they are design objectives or performance levels:

- **Energy-efficient computing** involves designing computing systems that maximise performance whilst minimising energy consumption. It relies on the use of efficient hardware (low-power CPUs, GPUs, FPGAs), software optimisations and advanced cooling techniques to reduce the energy costs and environmental impact of HPC centres.

- **Exascale computing** refers to supercomputers capable of achieving 10^{18} operations per second (1 exaflop) through massive parallelism combining thousands of multi-core nodes and

Remove Watermark Wondershare PDFelement

accelerators. An energy optimisation strategy enabling the resolution of large-scale scientific simulations and AI problems, such as on the Frontier system at Oak Ridge National Laboratory. This strategy can make a CPU energy-efficient, a GPU energy-efficient and a cluster energy-efficient.

6. Exemples d'applications nécessitant le calcul parallèle

In this section, we discuss a few areas that use parallel computing to improve performance.

- **Massive parallelism for artificial intelligence**
  Deep neural networks (Deep Learning) require enormous amounts of parallel computation. GPUs and TPUs enable the training of models with billions of parameters. Libraries such as TensorFlow, PyTorch and CUDA facilitate the use of parallel computing on GPUs. Most recent advances in AI rely on parallel computing.

- **Parallel computing in cloud computing**
  Cloud platforms (AWS, Azure, Google Cloud) offer on-demand parallel clusters. Ability to run distributed tasks at very large scale with Kubernetes, Spark or MPI. Serverless models allow resources to be scaled automatically.

- **High-performance computing (HPC)**
  Simulations and modelling scientific applications (climate, biology, astrophysics) use supercomputers. Examples of use cases : Global climate simulation, fluid dynamics and pharmaceutical research.

- **Big Data and distributed analytics**
  Internet traffic, application logs and IoT data to be processed in real time. Tools such as Apache Hadoop and Apache Spark utilise parallel computing to analyse terabytes or petabytes of data. The focus is shifting towards real-time streaming.

7. Exercises

Exercise 1: Classification of architectures

Associate each item to the correct category: Cluster, GPU, Multi-core computer, Cloud computing

Exercice 2 : Complete the table :

Criterion	CPU	GPU
Number of cores	?	?

Remove Watermark Wondershare PDFelement

Criterion	CPU	GPU
Type of processing	?	?

Exercice 3 : A deep learning application must train a model on a large dataset.

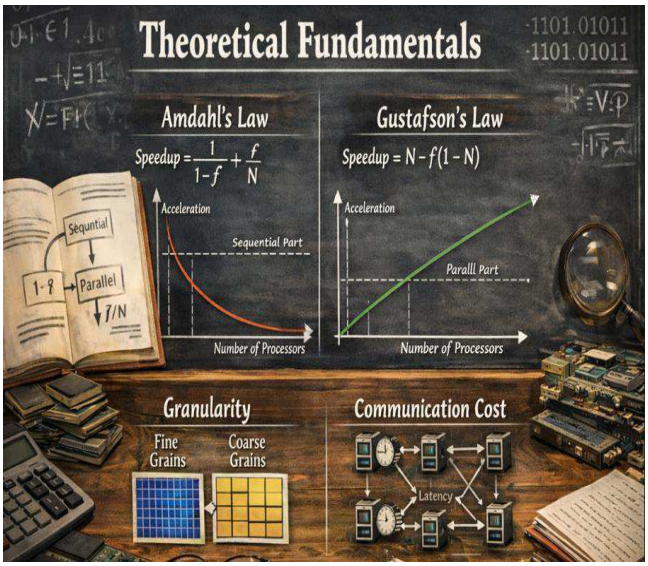
1. Why use a GPU instead of a CPU ?
2. What role does parallelism play in this case ?
3. Give another field requiring HPC.

Exercise 4 : Name a real-world application and explain:

1. Why it uses parallelism
2. What type of architecture is used (CPU, GPU, cluster…)

Wondershare PDFelement

Remove Watermark Wondershare PDFelement



Chapter2 : Fundamental Laws of Parallel Computing

Key Words

Fundamental theoretical laws; Amdahl's law; Gustafson's law; Granularity; Communication cost

Abstract

The fundamental theoretical principles of parallel computing and the key concepts relating to the performance of parallel systems are presented, introducing the main theoretical models of performance and scalability, notably Amdahl's law, which enables the performance gain limited by the sequential part of a programme to be assessed, and Gustafson's law, which highlights the benefits of parallelism for large-scale problems.
Granularity is a key element of parallel computing, as it determines how a problem is divided into subtasks and distributed among the computing units. It directly influences overall performance : appropriate granularity enables efficient execution, whilst too fine a granularity leads to additional communication overhead and too coarse a granularity limits parallelism. It also plays an important role in load balancing by ensuring an even distribution of work across processors, thereby avoiding idle time. Furthermore, granularity affects communication costs, as smaller tasks require more data exchanges. It also determines the system's scalability—that is, its ability to utilise an increasing number of processors—whilst contributing to better utilisation of hardware resources such as CPUs, GPUs and memory.

1. Introduction

Parallel computing involves dividing a problem into several subtasks that are executed simultaneously across multiple computing units (processors, cores or accelerators such as GPUs). This approach significantly reduces execution time and improves the overall efficiency of computing systems.
However, the use of parallelism does not always guarantee unlimited performance gains. Fundamental laws, such as Amdahl's law and Gustafson's law, highlight the limitations of parallelism, particularly in the presence of sequential parts or communication costs between processes. It is therefore essential to understand both the advantages and the constraints of parallel computing.

Remove Watermark Wondershare PDFelement

2. Theoretical Models of Performance and Scalability

These are analytical models (or performance laws) that predict speedup, evaluate the limits of parallelism, and help understand the relationship between the number of processors, the sequential portion of a program, and the achievable performance gain.

Parallel computing aims to reduce program execution time by distributing the workload across multiple processors. However, the resulting speedup is limited by theoretical constraints formalized by several fundamental laws. These models include: **Amdahl's Law (1967), Gustafson's Law (1988), Moore's Law (1965), and Brent's Law (Work-Time Law).**

2.1. Amdahl's Law and Performance

Amdahl's Law is primarily intended to measure the theoretical limit of performance improvement that can be achieved when parallelizing a program. It allows us to predict the maximum speedup that can be obtained using multiple processors, taking into account the portion of the program that remains sequential. Amdahl's Law imposes several constraints:

- A program can never be 100% parallelizable.
- There is always a sequential part (initialization, communication, synchronization, etc.).
- Amdahl's Law shows that this small sequential portion strongly limits overall performance, even when many processors are added.
- Amdahl's Law considers a fixed-size problem. It aims to determine how much faster the same program can run when using multiple processors.

In his book *Parallel Programming in C with MPI and OpenMP*, in the chapter titled *Performance Analysis*, M. J. Quinn defines the execution time for a problem of size *n* using *p* processors as consisting of two parts: the sequential part and the parallel part, as follows:

Remove Watermark Wondershare PDFelement

- ρ = inherently sequential part
- φ = part that can be executed in parallel

Time for one processor:

$$T_{1,n} = \rho + \varphi$$

Time for p processors:

$$T_{p,n} = \sigma + \frac{\varphi}{p}$$

Speedup:

$$\omega \leq \frac{\sigma + \varphi}{\sigma + \frac{\varphi}{p}}$$

f: fraction of operations that must be executed sequentially

$$0 \leq f \leq 1$$

We then have:

$$f = \frac{\sigma}{\sigma + \varphi}$$

$$\sigma + \varphi = \frac{\sigma}{f}$$

And:

$$\varphi = \frac{\sigma}{f} - \sigma = \sigma\left(\frac{1}{f} - 1\right)$$

Therefore:

$$\psi \leq \frac{\sigma + \varphi}{\sigma + \varphi/p}$$

$$= \frac{\sigma/f}{\sigma + \sigma\left(\frac{1}{f} - 1\right)/p}$$

$$= \frac{1/f}{1 + \left(\frac{1}{f} - 1\right)/p}$$

$$= \frac{1}{f + (1 - f)/p}$$

Remove Watermark Wondershare PDFelement

Definition of Speedup – Speedup -
Speedup measures the increase in a programme's speed when multiple processors (parallelism) are used, compared to a single processor. Speedup is the performance gain and is expressed as follows:

Speedup = Execution time with 1 processor / Execution time with N processors

Example

- Time with 1 processor: 10 seconds
- Time with 4 processors: 2 seconds

$$S=10/2=5$$

The speedup is 5, which means the programme is 5 times faster.

Example

A programme of which 90% could run in parallel on an 8-processor parallel architecture.

- Calculate the speedup using Amdahl's law.
- Interpret the result obtained.

Solution

Maximum speedup with eight (8) processors:

$$\psi \leq \frac{1}{0.1+(1-0.1)/8} \approx 4.7$$

A programme of which 75% could run in parallel.

Maximum acceleration on a parallel machine with a limited number of processors

$$\lim_{p\to\infty} \frac{1}{0.25+(1-0.25)/p} \approx 4$$

Interpretation

Theoretically, if the entire programme could be 100% parallelised, we would have an ideal speedup: $S_{ideal} = P = 8$

so the programme would be 8 times faster than in sequential execution.

However, because of the sequential part (10%), the actual speedup is only:

$S_{reel} = 4.7$

Why don't we reach 8 ?

The answer is that there is a part of the programme (10%) that cannot be divided among the processors. Even with 8 processors: this 10% must be handled by a single processor, whilst the other 7 wait.

Remove Watermark Wondershare PDFelement

This is exactly what Amdahl's law describes : the sequential time limits the overall speed.

In conclusion, we can say that Amdahl's law shows that :

- The speed-up of a parallel programme is limited by the sequential part of the programme and
- It allows us to estimate the actual gains from parallelisation for an existing programme.
- It is a conservative estimate because it assumes that the workload remains constant.

2.2. Gustafson's Law and scalability

Gustafson's Law (proposed by John L. Gustafson) demonstrates that the size of a problem can be increased when more processors are available, whilst maintaining a reasonable execution time.

Unlike Amdahl's law (which assumes that the problem size is fixed), Gustafson's law considers that in practice, when processors are added (parallel computing), a larger or more complex problem can be processed within the same time.

The aim of Gustafson's law is to demonstrate:

1. The scalability of the problem: the ability of a parallel system to solve problems of increasing size.
3. The benefit of using multiple processors to increase the amount of work performed.
4. That parallelism is very useful in fields such as:
    - scientific simulations,
    - Big Data,
    - artificial intelligence, and
    - smart grids and complex systems.

Thus, Gustafson's law seeks to demonstrate that parallel systems become more efficient as the size of the problem increases, and to prove that the more processors available, the larger the problem that can be solved efficiently.
Gustafson's law calculates the speed-up using the following formula:

Remove Watermark Wondershare PDFelement

$$S(P) = P - s * (P - 1)$$

Such as:

P: Number of processors
S: Proportion of total parallel time spent in the non-parallelisable part

Example

In a programme with S, the proportion of time spent in the non-parallelisable part is 0.05 on a parallel architecture with 10 processors.

1. Work out the acceleration factor according to GUSTAFSON.
2. Interpret the result obtained

Solution

S(P)=P-s*(P-1)

$S(10)$ =10 - 0.05*(10-1) =10 - 0.45=9.55

Interpretation: the speed-up is almost linear (≈9.55), indicating 100% utilisation of the parallel architecture; therefore, Gustafson is more optimistic.

3. Granularity and Parallelism

3.1. Definition

Granularity in parallel computing refers to the level at which a problem or program is decomposed into independent computational units that can be executed simultaneously. It characterizes the relative size of parallel tasks as well as the ratio between computation time and the cost of communication or synchronization between these tasks.

Example

If we create multiple threads in the Java language, each thread represents a parallel task, and the size of these tasks represents the granularity.

3.2. Types of Granularity

Granularity influences the efficiency of parallelism. Well-chosen granularity allows effective use of parallel resources while minimizing communication costs between tasks. We distinguish three types of granularity: fine-grained, medium-grained, and coarse-grained.

1. **Fine-Grained Granularity**: Refers to parallelism at the level of instructions or operations and typically involves a large number of small, simple processors.

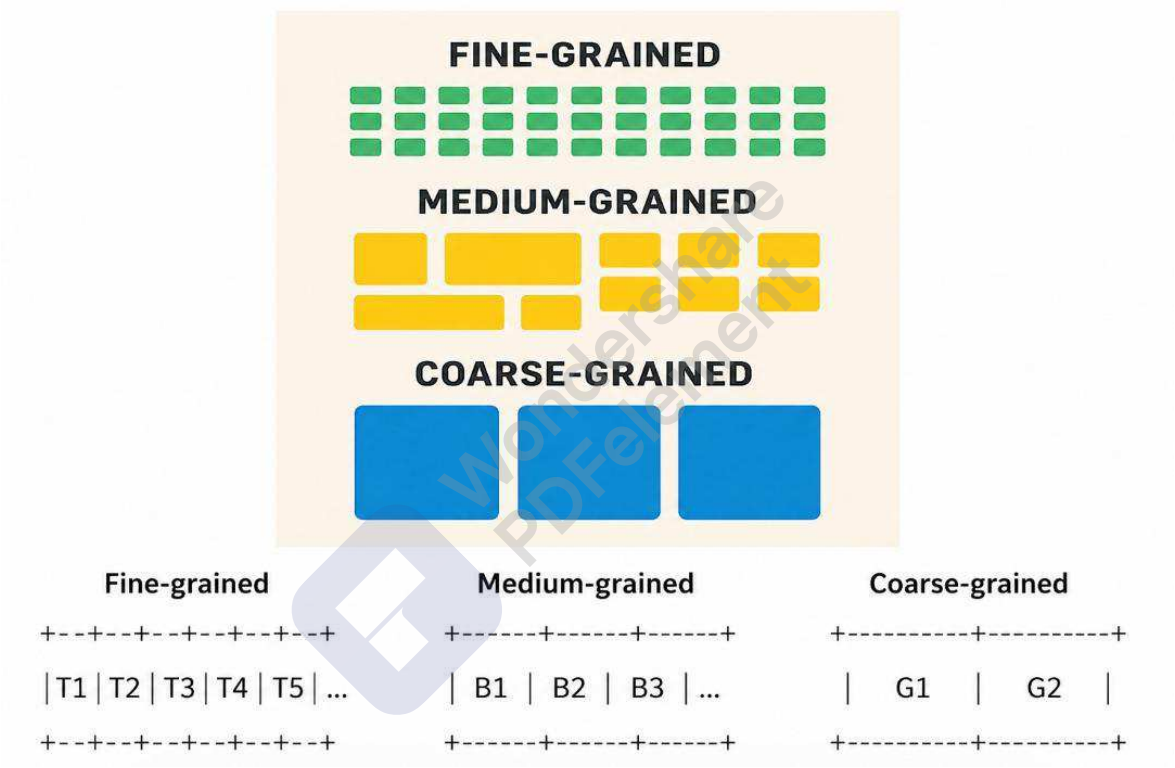
Example: Parallelizing a loop element by element.

Remove Watermark Wondershare PDFelement

2. **Medium-Grained Granularity**: Involves executing procedures in parallel.
   Example: Processing a block of data rather than a single element.

3. **Coarse-Grained Granularity**:Refers to parallelism at the level of processes and programs and involves a small number of powerful, complex processors.
   Example: Running different algorithms or modules on separate processors.

The following diagram illustrates the different types of granularity.



Types of granulariy

Example

We want to compute the sum of the numbers from 1 to 1000 using multiple threads.

1. Propose a solution with fine-grained granularity.
2. Propose a solution with medium-grained granularity.
3. Propose a solution with coarse-grained granularity.

1. Fine-Grained Granularity
Each thread computes a single number.

Remove Watermark Wondershare PDFelement

Thread	Task
T1	computes 1
T2	computes 2
T3	computes 3
...	...
T1000	computes 1000

Number of threads: 1000 ➞ Tasks are very small → fine-grained granularity.

2. Medium-Grained Granularity
Each thread computes a block of 100 numbers.

Thread	Task
T1	1 → 100
T2	101 → 200
T3	201 → 300
...	...
T10	901 → 1000

Number of threads: 10 ➞ Tasks are medium-sized → **medium-grained granularity**

3. Coarse-Grained Granularity
Each thread computes a large portion of the work.

Thread	Task
T1	1 → 500
T2	501 → 1000

Number of threads: 2 ➞ Tasks are large → **coarse-grained granularity**

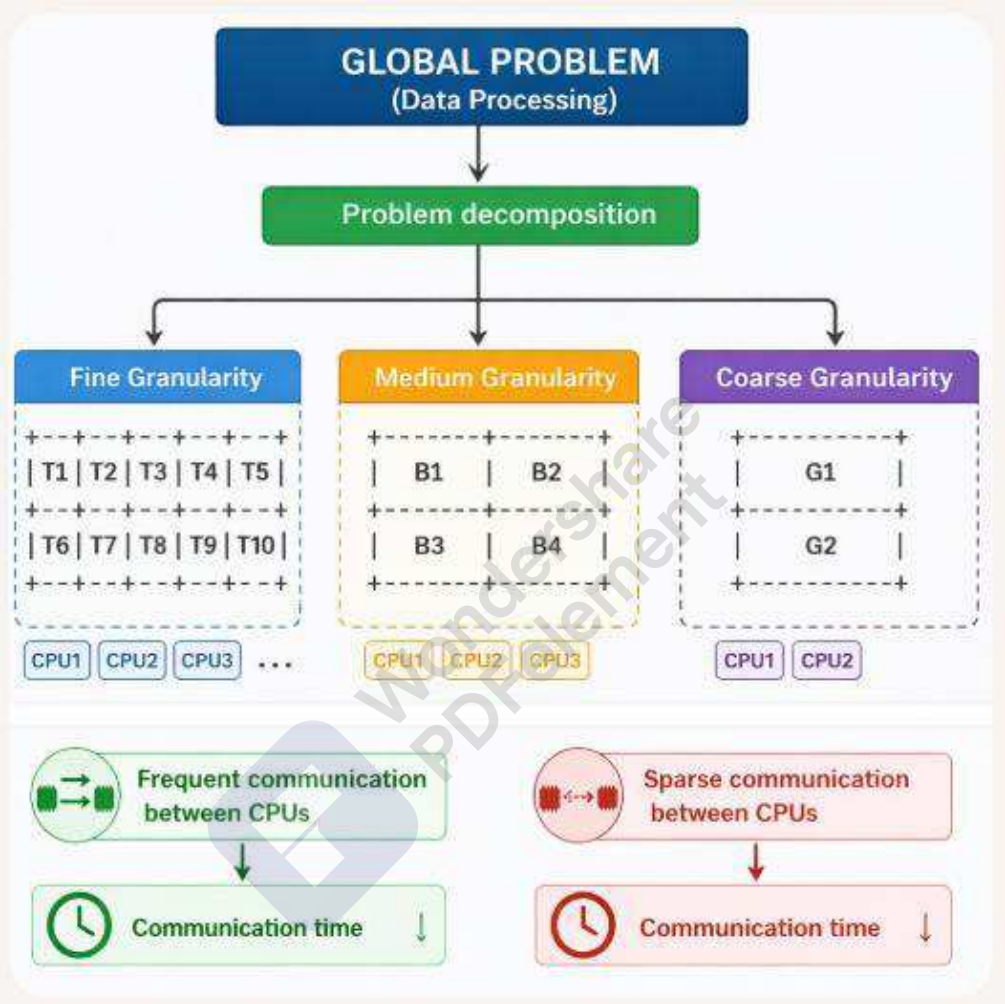
3.3. The impact of granularity on the performance of parallel systems

Granularity is directly linked to two main aspects : processing time and communication time between processors. When granularity is high (coarse granularity), each processor executes a large task, which often reduces management and synchronisation time. Conversely, when granularity is fine, the problem is divided into several small tasks executed by a large number of processors. In this case, even though parallelism increases, the communication time between CPUs may become longer, as the processors must exchange more information and synchronise more frequently. This additional cost can reduce the speed-up gained from parallelism. Consequently, granularity has a significant impact on the overall performance of the parallel system.

Remove Watermark Wondershare PDFelement

It is therefore necessary to choose an appropriate granularity in order to limit communication whilst making effective use of parallelism. The aim is therefore to find an optimal granularity that balances computation time and communication time.

An illustration of the impact of granularity on the performance of parallel systems is shown in the diagram below.



Impact of Granularity on Communication Time

3.3. Impact of Communication Time on Speedup: Application

We consider a parallel program whose total work is:

$$W = 1000$$

The program is executed on $p = 10$processors

The communication time per task is:

$$t_{\text{comm}} = 2$$

We study two strategies for workload decomposition:

Remove Watermark Wondershare PDFelement

- **Case 1:** Fine-grained decomposition
- **Case 2:** Coarse-grained decomposition

Symbol	Meaning	Value
W	Total work	1000
p	Number of processors	10
t_{comm}	Communication time per task	2
t_{comp}	Computation time per task	depends on the case
N	Number of tasks	depends on the case

Case 1: Fine-Grained Decomposition

The work is divided into many small tasks: $N = 1000, t_{\text{comp}} = 1$

Sequential Time: The execution time on a single processor is:

$$T_1 = W = 1000$$

Task Distribution : Each processor receives: $\frac{N}{p} = 100$tasks

Time per Task :

$$t_{\text{task}} = t_{\text{comp}} + t_{\text{comm}}$$
$$t_{\text{task}} = 1 + 2 = 3$$

Time per Processor: $T_{\text{per_proc}} = 100 \times 3 = 300$

Parallel Time: $T_p = 300$

Speedup : $S = \frac{T_1}{T_p}, \; S = \frac{1000}{300} \approx 3.33$

The speedup is low because the communication cost is high compared to the computation time.

Case 2: Coarse-Grained Decomposition

The work is divided into a small number of large tasks: $N = 10, t_{\text{comp}} = 100$

Sequential Time: $T_1 = 1000$

Task Distribution : Each processor receives: 1 task

Time per Task :

$$t_{\text{task}} = t_{\text{comp}} + t_{\text{comm}}$$
$$t_{\text{task}} = 100 + 2 = 102$$

Remove Watermark Wondershare PDFelement

Parallel Time: $T_p = 102$

Speedup: $S = \frac{T_1}{T_p}, S = \frac{1000}{102} \approx 9.80$

The speedup is almost ideal because the communication time is negligible compared to the computation time.

4. Exercises :

Exercise 1

A program has a parallelizable portion of 80%.
What is the maximum theoretical speedup if a large number of processors is used ($\rightarrow \infty$)?

Exercise 2

Indicate whether the following statements are **true or false**:

1. Parallel computing completely eliminates execution time.
2. A GPU is suitable for sequential tasks.
3. Distributed computing uses multiple machines.
4. The more processors are added, the speedup becomes infinite.

Exercise 3

A power flow computation algorithm in a smart grid is:
70% parallelizable and 30% sequential, with 4 processors.

1. Calculate the theoretical speedup.
2. Interpret the result.
3. What is the maximum speedup if an infinite number of processors is used?

Solution:

1. Amdahl's Law :

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

$$S = \frac{1}{0.3 + \frac{0.7}{4}} = \frac{1}{0.3 + 0.175} = \frac{1}{0.475} \approx 2.10$$

Speedup ≈ 2.1

2. Interpretation: Even with 4 processors, the gain remains limited due to the sequential portion (30%).

Remove Watermark Wondershare PDFelement

3. Even with an infinite number of processors, the speedup can not exceed 3.33 because of the sequential portion.

Exercise 4 : In a smart grid, a large-scale electrical network is simulated:
Sequential part: 10% Parallel part: 90% Number of processors: 8

1. Calculate the speedup using Gustafson's Law.
2. Interpret the obtained result.

Solution :

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

Avec : $N = 8 \;\; P = 0.9$

$$S = 8 - (0.1)(7) = 8 - 0.7 = 7.3$$

The gain is close to linear because the problem size increases (e.g., more sensors, more data).

Exercise 5: Amdahl vs. Gustafson Comparison (Smart Grid)

We consider:
Parallelizable portion = 80%
Number of processors = 10

1. Calculate the speedup using Amdahl's Law.
2. Calculate the speedup using Gustafson's Law.
3. Compare and interpret the results.
4. Which law "predicts the correct speedup"?

Amdahl : $S = \frac{1}{0.2+\frac{0.8}{10}} = \frac{1}{0.2+0.08} = \frac{1}{0.28} \approx 3.57$

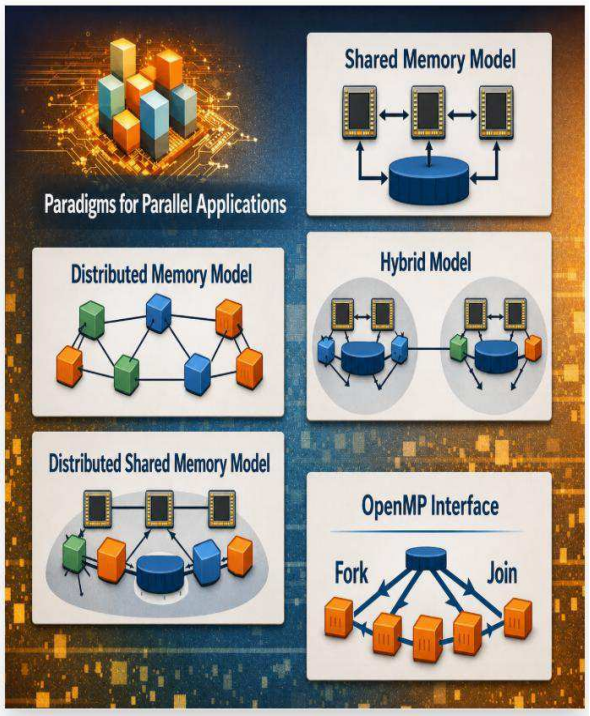
Gustafson : $S = 10 - (0.2 \times 9) = 10 - 1.8 = 8.2$

Interpretation:

- **Amdahl** → limited speedup (3.57)
- **Gustafson** → high speedup (8.2)

The difference is due to the fact that Gustafson assumes an increase in the size of the smart grid (more data, more nodes).

Remove Watermark Wondershare PDFelement



Chapter 3 : Application model and architecture for parallel programming interfaces

Wondershare PDFelement

Key Words

Interfaces; parallel programming paradigm; shared memory; distributed memory; OpenMP; MPI.

Abstract

Parallel programming relies on models and interfaces that enable the efficient use of multiple processing units to speed up data processing. Parallelism paradigms define how tasks are organised, whether through data parallelism, task parallelism or hybrid approaches. Three main models underpin this discipline: the shared-memory model, where multiple processors access a common memory and must manage synchronisation; the distributed-memory model, where each processor has its own local memory and communicates via message passing; and the hybrid model, which combines the two to optimise performance in modern architectures. Parallel programming interfaces play a central role in providing suitable tools and libraries, characterised by their portability, efficiency and simplicity, with examples such as MPI, OpenCL or OpenMP. Among these, OpenMP stands out as a standardised interface for shared memory, using the fork–join model: a master thread creates several threads to execute tasks in parallel before merging them. Taken together, this highlights the complementary nature of models and interfaces in harnessing the potential of contemporary parallel architectures.

1. Introduction

Parallel programming models and paradigms represent the conceptual approaches used to organise the simultaneous execution of multiple tasks within an application. A parallel programming model defines the general organisation of computation, including how data is distributed, shared and processed across different computational units. Paradigms, on the other hand, describe the strategies for structuring parallelism and managing interaction between processes or threads. To implement these models and paradigms on

Remove Watermark Wondershare PDFelement

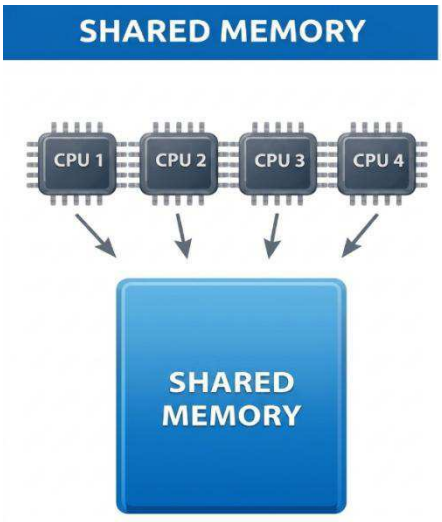
modern computing architectures, developers use parallel programming interfaces (APIs: Application Programming Interfaces) which provide a set of functions, libraries and mechanisms to facilitate the management of parallelism. These interfaces simplify the creation, coordination and communication between different computing units. In this chapter, we will discuss the various existing parallel programming paradigms and the most commonly used interfaces for each model.

2. Paradigms for parallel applications

Parallel programming models or paradigms define the various approaches to organising the simultaneous execution of multiple tasks within an application. They describe how processes or threads communicate, share data and synchronise to make effective use of parallel architectures. These models facilitate the design of applications capable of taking advantage of multiprocessor and multicore systems, as well as distributed computing environments. The main parallel programming models include the shared-memory model, the distributed-memory model and the hybrid model.

3. Shared-memory model

In the shared-memory model, all processes or threads have access to a common memory in which data is stored. The various computing units can read and modify the same variables within this shared memory space.
In this model, communication between processes occurs implicitly through shared memory. However, when multiple threads access the same data simultaneously, concurrency issues may arise. To avoid these problems, it is necessary to use synchronisation mechanisms such as locks (mutexes), semaphores or monitors. This model is generally used in multi-core, multi-processor systems, where the processors share a central memory. The figure below illustrates the shared memory model.

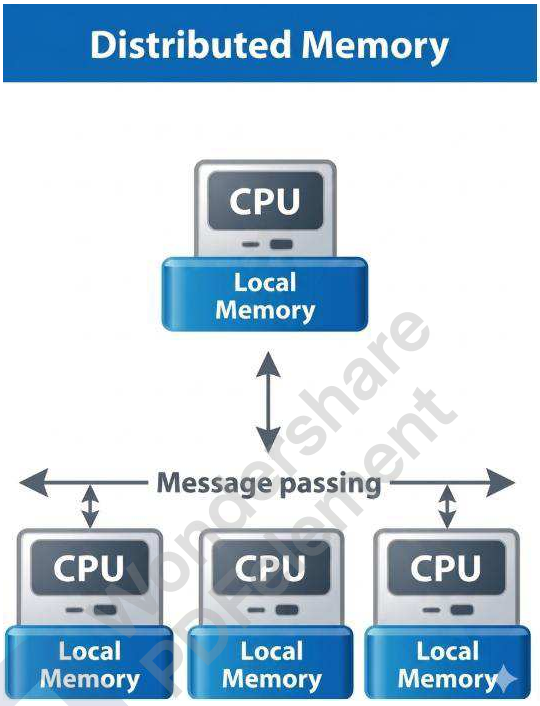


Parallel programming models: shared memory

Remove Watermark Wondershare PDFelement

4. Distributed memory model

In the distributed memory model, each processor or node has its own local memory. Processes cannot directly access the memory of other processes. Consequently, communication between them takes place via message passing. In this model, processes must explicitly send and receive messages to share data. This type of programming is often used in clusters and high-performance computing systems. The following figure illustrates this model.

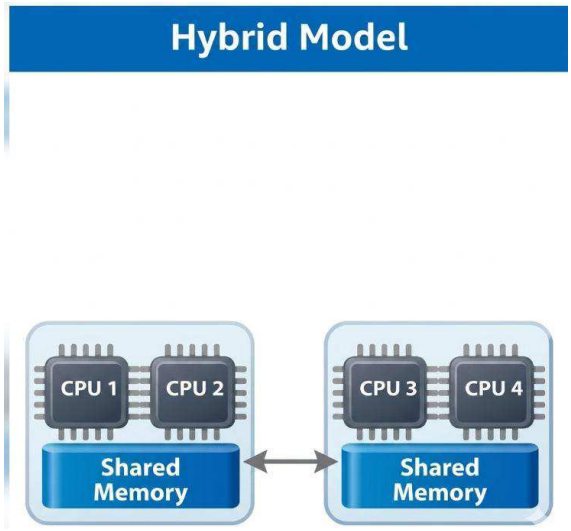


Parallel programming models: Distributed Memory

5. Hybrid model

The hybrid model combines the characteristics of shared-memory and distributed-memory models. In this approach, a system consists of several nodes connected by a network, and each node has several processors or cores that share a local memory. Thus, communication between nodes takes place via message passing (distributed memory), whilst communication within a node is achieved through shared memory. This model is widely used in modern supercomputers, as it enables the efficient utilisation of complex parallel architectures.

Remove Watermark Wondershare PDFelement



Parallel programming models : Hybrid model

6. Interfaces for parallel programming

In this section, we outline the main interfaces used in parallel programming.

6.1. Definition of parallel programming interfaces

The interfaces used in parallel programming paradigms are sets of libraries, functions and software mechanisms that enable developers to design and implement applications capable of executing multiple tasks simultaneously. These interfaces provide the tools needed to manage the creation of processes or threads, communication between computing units, data sharing and task synchronisation. They thus serve as a bridge between parallel programming models or paradigms and their practical implementation on hardware architectures. Among the most commonly used interfaces are OpenMP for shared-memory systems, MPI for distributed-memory systems, and CUDA for parallel computing on graphics processing units. We will detail below the interfaces for shared-memory systems, distributed systems and hybrid systems. We will cover parallel computing on graphics processing units in Chapter Four of this manuscript.

6.2. Key Features

Parallel programming interfaces possess several features that facilitate the development and execution of parallel applications on different computing architectures.

- **Parallelism Management :** Interfaces enable the creation and management of multiple processes or threads in order to execute several tasks simultaneously.

Remove Watermark Wondershare PDFelement

- **Inter-process communication**: provide mechanisms for exchanging information between different computing units. This communication can take place via shared memory or message passing, depending on the model used.
- **Task synchronisation:** Interfaces provide tools to coordinate the execution of parallel tasks and avoid conflicts over access to shared resources. This includes the use of locks, semaphores or synchronisation barriers.
- **Portability:** Applications developed using these interfaces can be run on different hardware architectures without major modifications.
- **Scalability:** The interfaces enable applications to operate efficiently with an increasing number of processors or compute nodes.
- **Performance and optimisation:** They provide mechanisms to optimise the use of hardware resources and improve the overall performance of the application.

These features are implemented in several parallel programming interfaces such as OpenMP (Open Multi-Processing) for shared-memory architectures, MPI (Message Passing Interface) for distributed-memory systems, and CUDA (Compute Unified Device Architecture) for parallel computing on GPUs.

6.3. Examples of parallel programming interfaces

Among the most widely used interfaces in parallel programming paradigms are OpenMP for shared-memory architectures, MPI for distributed-memory systems, and CUDA for parallel computing on GPUs (Chapter 4). In the following, we will describe these interfaces in detail.

7. OpenMP interface

In this section, we detail the interface for shared-memory architectures: OpenMP

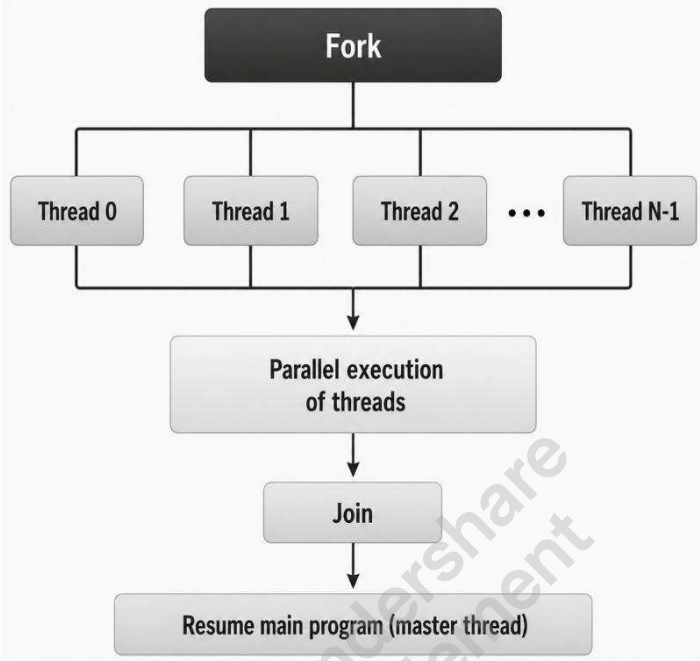
7.1 Introduction to OpenMP

OpenMP (Open Multi-Processing) is a parallel programming interface based on shared-memory parallelism, widely used in HPC (High Performance Computing). It enables the utilisation of the parallelism available within a single compute node—that is, a computer with multiple processor cores—by creating and coordinating threads that execute different parts of a programme in parallel.

Remove Watermark Wondershare PDFelement

7.2 Operating Principle of OpenMP and the fork-join model

The fork–join model is the basic operating principle of OpenMP. A main (master) thread creates several threads to execute a part of the programme in parallel, and they then synchronise at the end.



Fork–Join Execution Process

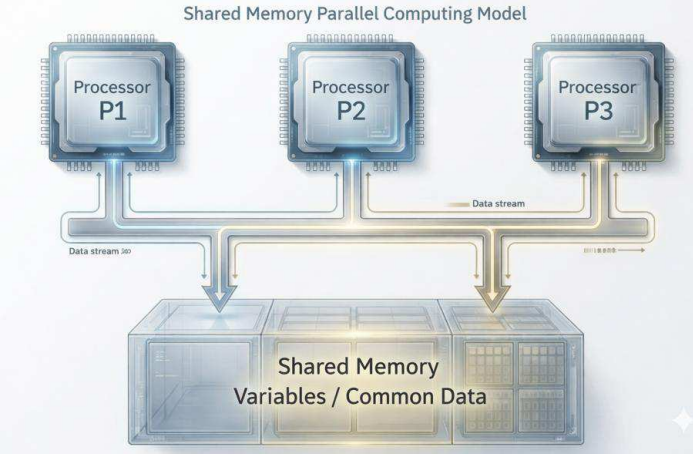
Operational steps:

1. Main programme: the master thread executes the sequential code.
2. Fork: creation of multiple threads (parallelism).
3. Parallel execution: each thread executes a portion of the code simultaneously.
4. Join: the threads terminate and the programme returns to the master thread to continue sequential execution.

7.3. Shared Memory Programming Model

The diagram of the Shared Memory Model in parallel programming shows several processors or threads accessing the same central memory. All processes can read from and write to this shared memory space.

Remove Watermark Wondershare PDFelement



Shared-memory programming model

In the shared-memory model:

- Multiple processes or threads run in parallel.
- They share the same memory.
- Data is exchanged directly by reading from and writing to memory.

7.4. Main directives, thread management and synchronisation mechanisms

1. **Key OpenMP directives:** OpenMP directives are used to define parallel regions and organise thread execution.

Directive	definition	Example -syntaxe-	Explication
Directive **parallel**	Create a parallel region where multiple threads execute the same block of code.	*#pragma omp parallel* *{* *printf("Hello from thread\n");* *}*	-A group of threads is created. -Each thread executes the code block in parallel.
Directive **for (ou parallel for)**	Distribute loop iterations among threads.	*#pragma omp parallel for* *for(int i=0; i<10; i++){* *printf("Iteration %d\n", i);* *}*	-The loop iterations are divided among the threads. -Each thread executes a portion of the loop.
Directive **sections**	Allow multiple different blocks of code to be executed in parallel.	*#pragma omp parallel sections* *{ #pragma omp section* *{printf("Section 1\n");* *}*	Each section is executed by a different thread.

Remove Watermark Wondershare PDFelement

		#pragma omp section *{* *printf("Section 2\n");* *}* *}*	
Directive **single**	Indicate that only one thread executes a section of code.	*#pragma omp single* *{ printf("Executed by one thread only\n");* *}*	Only one thread executes the single block. The other threads wait by default at the end of the block (implicit barrier).
Directive **critical**	Protect a critical section so that only one thread can access it at a time.	*#pragma omp critical* *{* *x = x + 1;* *}*	To avoid race conditions.

Main OpenMP Interface Directives for Parallelism

1. **Thread management in OpenMP :**Thread management involves controlling their creation, number, and identification.

Set the number of threads

omp_set_num_threads(4);ou #pragma omp parallel num_threads(4)

Identificate un thread

int id = omp_get_thread_num();

This function returns the identifier of the current thread.

Get the total number of threads

int n = omp_get_num_threads();

This function returns the total number of the threads in the parallel region parallèle.

2. **Synchronisation mechanisms :** Synchronisation mechanisms in OpenMP enable control over access to shared resources and ensure data consistency in parallel programmes. The main directives are barrier, critical, atomic, ordered and flush, each suited to a specific type of synchronisation. The directives that provide these synchronisation mechanisms are summarised in the table below.

Remove Watermark Wondershare PDFelement

Directive	definition	Example -syntaxe-	Explication
Directive **barrier**	- Forces all threads to wait until each thread reaches this point in the program before continuing execution.	*#pragma omp barrier*	-All threads stop at the barrier. - Execution resumes only when all threads have arrived. -Synchronize threads.
Directive **critical**	Protects a critical section so that only one thread can execute this block of code at a time.	*#pragma omp critical* *{* *x = x + 1;* *}*	- Only one thread accesses the critical section. - The other threads wait for their turn.
Directive **atomic**	Ensures that a simple operation on a shared variable is executed atomically (intended for simple operations).	*#pragma omp atomic* *x++ ;*	The operation is performed without interruption by another thread.
La directive **ordered**	• Allows certain parts of a loop to be executed in sequential order, even if the loop is parallel.	*#pragma omp parallel for ordered* *for(int i=0; i<10; i++){* *#pragma omp ordered* *printf("%d\n", i);* *}*	-Loop iterations are executed in parallel. -The ordered section respects the iteration order.
Directive **flush** .	Enables synchronization of variable values between threads by forcing memory updates.	*#pragma omp flush(x)*	Ensures that the shared variable has the same value across all threads.

Main directives of the OpenMP Interface for Thread Synchronisation Mechanisms

Remove Watermark Wondershare PDFelement

8. MPI Interface (Message Passing Interface)

This section describes the MPI interface for the distributed-memory parallelism model.

8.1. Introduction to MPI

MPI is a standard parallel programming interface based on message passing between processes. It is mainly used in parallel computing systems and distributed-memory architectures, such as clusters and supercomputers. MPI allows multiple processes running on different machines or nodes of a system to communicate and exchange data by sending and receiving messages. Each process has its own local memory and communicates with other processes using communication functions defined by the MPI interface.

This interface was developed in the early 1990s to provide a portable and efficient standard for parallel programming. Today, MPI is widely used in scientific computing, numerical simulation, big data processing, and high-performance computing (HPC).

In practice, MPI provides a set of functions or routines that allow processes to be created, messages to be sent and received, processes to be synchronized, and collective communications between multiple computing units to be managed.

8.2. Objectives of MPI

The MPI interface was designed to achieve several key objectives in parallel programming:

1. **Standardize communication between processes:** MPI provides a standard interface that allows processes to exchange data in a structured and portable way across different parallel architectures.
2. **Enable execution on distributed systems:** MPI facilitates the development of parallel applications that can run on machine clusters or supercomputers, where each process has its own memory.
3. **Improve application performance:** By enabling simultaneous execution of multiple processes and distributing computations, MPI helps reduce the execution time of computationally intensive applications.
4. **Ensure application portability:** Programs developed with MPI can run on different parallel systems and hardware platforms without major code modifications.
5. **Provide efficient communication mechanisms:** MPI offers several types of communication, including point-to-point communication and collective communication.

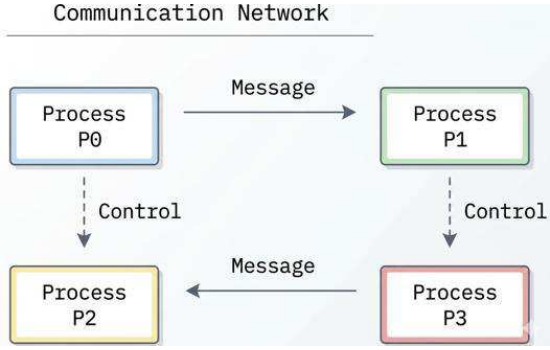
8.3. Distributed-Memory Programming Model

The most widely used parallel programming architecture today is MPI for distributed-memory systems. Each process has its own memory and communicates

Remove Watermark Wondershare PDFelement

with other processes by sending and receiving messages through a communication network.

The diagram below illustrates the MPI operation based on distributed memory systems.



All processes are part of the global communicator. MPI_COMM_WORLD.

8.4. Main MPI Functions

The main functions of the MPI interface are:

Primitive : MPI_Comm_rank() :

Definition: It allows obtaining the number (rank) of the current process within an MPI communicator. Each process has a unique identifier called a **"rank"**.

Syntax in C language: int MPI_Comm_rank(MPI_Comm comm, int *rank);

Parameters:

- **comm**: the communicator (usually MPI_COMM_WORLD)
- **rank**: variable that will store the process rank

Example

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Je suis le processus %d\n", rank);

  MPI_Finalize();
  return 0; }
```

Remove Watermark Wondershare PDFelement

3. Primitive : MPI_Comm_size()

Definition: MPI_Comm_size() is used to determine the total number of processes in an MPI communicator (i.e., how many processes are working together).

Syntax: int MPI_Comm_size(MPI_Comm comm, int *size);

Parameters:

- comm: the communicator (e.g., MPI_COMM_WORLD)
- size: variable that stores the total number of processes

Example

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {

   int rank, size;

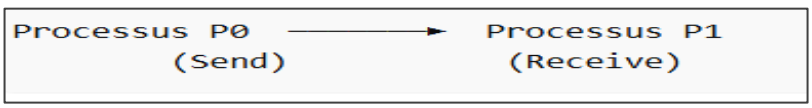
   MPI_Init(&argc, &argv);// démarrer MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);// connaître le rang

MPI_Comm_size(MPI_COMM_WORLD, &size);// connaître le nombre de processus
    printf("Processus %d sur %d\n", rank, size);// calcul paralléle
    MPI_Finalize(); //términer MPI
return 0;}
```

8.5. Message Passing Communications (MPI)

In MPI, communications are divided into two main categories: **point-to-point communications** and **collective communications**. Each category provides specific primitives. These primitives allow efficient management of data exchange in parallel computing applications.

a. Point-to-point communications: Point-to-point communications are operations in which a source process communicates directly with a destination process. The point-to-point communication primitives in MPI enable the direct exchange of messages between two processes. They are mainly based on the functions MPI_Send and MPI_Recv, which respectively handle sending and receiving data between a source process and a destination process. This is illustrated in the diagram below.



Point to Point Communication

Remove Watermark Wondershare PDFelement

This type of communication is based on the following main primitives :

Primitive	Role
MPI_Send()	send a message
MPI_Recv()	receive a message
MPI_Isend()	non-blocking send
MPI_Irecv()	non-blocking receive
MPI_Sendrecv()	send and receive at the same time

Example :

Sending a message: MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

Parameter	Description
&data	data to be sent
1	number of elements
MPI_INT	data type
1	destination process
0	message tag
MPI_COMM_WORLD	Communicator

Receiving a message:
MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

Parameter	Description
&data	variable where the received data is stored
1	number of elements to receive
MPI_INT	data type
0	source process (sender)
0	message tag
MPI_COMM_WORLD	communicator
MPI_STATUS_IGNORE	ignore status information

b. Collective communications: They allow data exchange between all processes in a group simultaneously. They are more efficient than point-to-point communications when multiple processes need to participate in the communication. The main collective operations are:

- MPI_Bcast
- MPI_Scatter
- MPI_Gather
- MPI_Reduce

Remove Watermark Wondershare PDFelement

Primitive	Fonctionality	Syntax
MPI_Bcast	Broadcast data from one process to all other processes	int data; if(rank==0){ data=100;} MPI_Bcast(&data,1,MPI_INT,0,MPI_COMM_WORLD); All the processes receive the value **100**
MPI_Scatter	Distribute different parts of an array to multiple processes	int data[4] = {1,2,3,4} ; int recv; MPI_Scatter(data,1,MPI_INT,&recv,1,MPI_INT,0,MPI_COMM_WORLD); Each process receives an element from the array
MPI_Gather	Gather data from all processes into a single process	int send = rank; int recv[4]; MPI_Gather(&send,1,MPI_INT,recv,1,MPI_INT,0,MPI_COMM_WORLD); Process 0 collects all the values.
MPI_Reduce	Combine values from multiple processes into a single value (sum, maximum, minimum, etc.)	int value = rank; int sum; MPI_Reduce(&value,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD); The process 0 obtains the som of all values.

Remove Watermark Wondershare PDFelement

9. Interface Hybride interface (MPI + Open MP)

9.1 Presentation of the hybride model

The OpenMP + MPI hybrid model combines two parallel programming models to make effective use of modern architectures (clusters with multi-core processors). The MPI-OpenMP hybrid model is a parallel programming approach that combines distributed-memory and shared-memory parallelism.

- MPI is used for communication between nodes in a cluster (distributed memory).
- OpenMP is used for parallelism between cores within the same processor (shared memory).

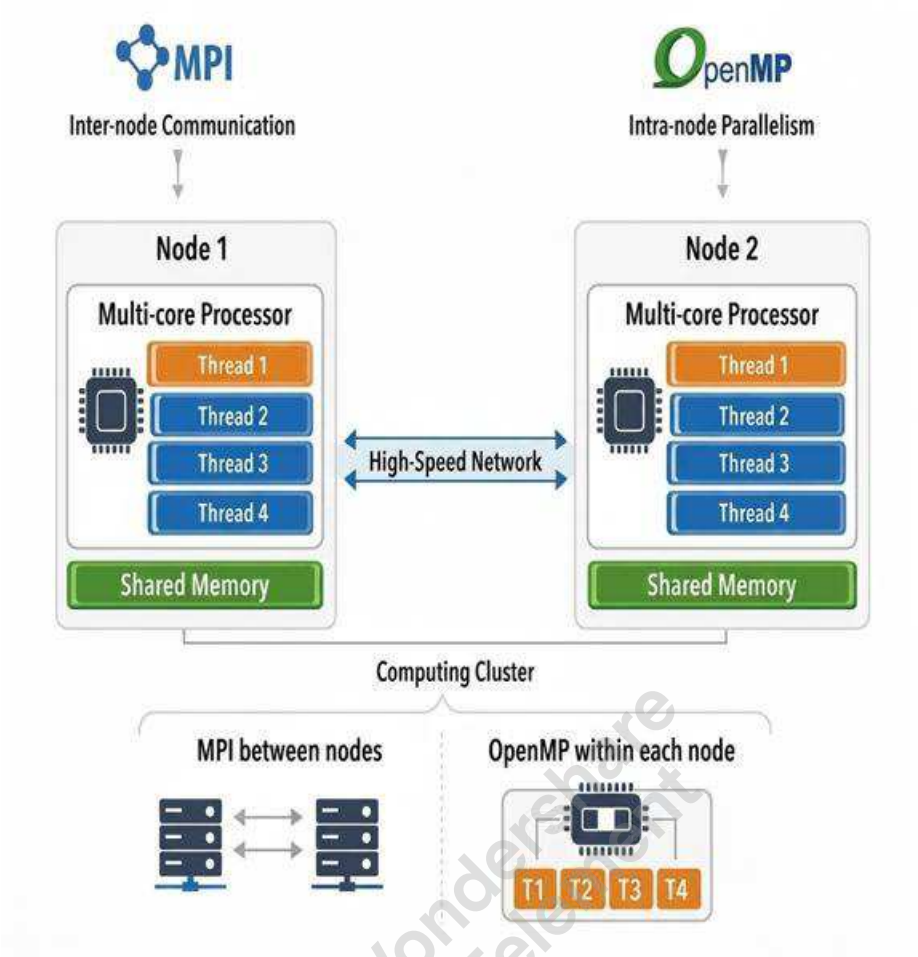
Thus, each MPI process can create multiple OpenMP threads to take advantage of the available cores.

9.2. Architecture of hybrid systems

In a hybrid system, the architecture consists of several compute nodes connected by a high-speed network. Each node contains a multi-core processor, shared memory and multiple compute threads. Thus, the MPI tool manages communication between the nodes, whilst the OpenMP tool manages parallel execution across the cores of the same processor. The main components of the architecture are:

- **Computing nodes**: Each node is a machine or server containing multi-core CPUs, local memory and an operating system
- **Interconnection network** : This enables communication between the nodes in the cluster. Examples : High-speed Ethernet or InfiniBand.
- **MPI processes**: Each node runs one or more MPI processes to communicate with the other nodes.
- **OpenMP threads**: Within each MPI process, several OpenMP threads are created to utilise the processor cores.

Remove Watermark Wondershare PDFelement



Hybrid Architecture MPI Open MP

9.3. Principale of operation

The operation can be summarised as follows:

1. The application starts with several MPI processes (often one per node or per CPU).

2. Each MPI process creates several OpenMP threads.

   - MPI manages communication between processes.

   - OpenMP manages local parallelism.

3. Distributed memory refers to the fact that each node has its own local memory and that these memories are not shared directly. Data is exchanged via MPI, whilst OpenMP uses the shared memory within each node.

Remove Watermark Wondershare PDFelement

9.4. Functionality of the MPI-OpenMP hybrid model

Steps of the Hybrid OpenMP–MPI Approach

The steps of the hybrid OpenMP–MPI process are organized as follows:

1. **MPI Initialization**

   The program starts and initializes the MPI environment.
2. **Creation of MPI Processes**

   Multiple processes are launched across the different nodes of the cluster.
3. **Process Identification**

   Each process obtains:
   - its rank
   - the total number of processes
4. **Creation of OpenMP Threads**

   Each MPI process creates multiple threads to exploit the CPU cores.
5. **Parallel Computation**

   Threads execute tasks in parallel within shared memory.
6. **MPI Communication**

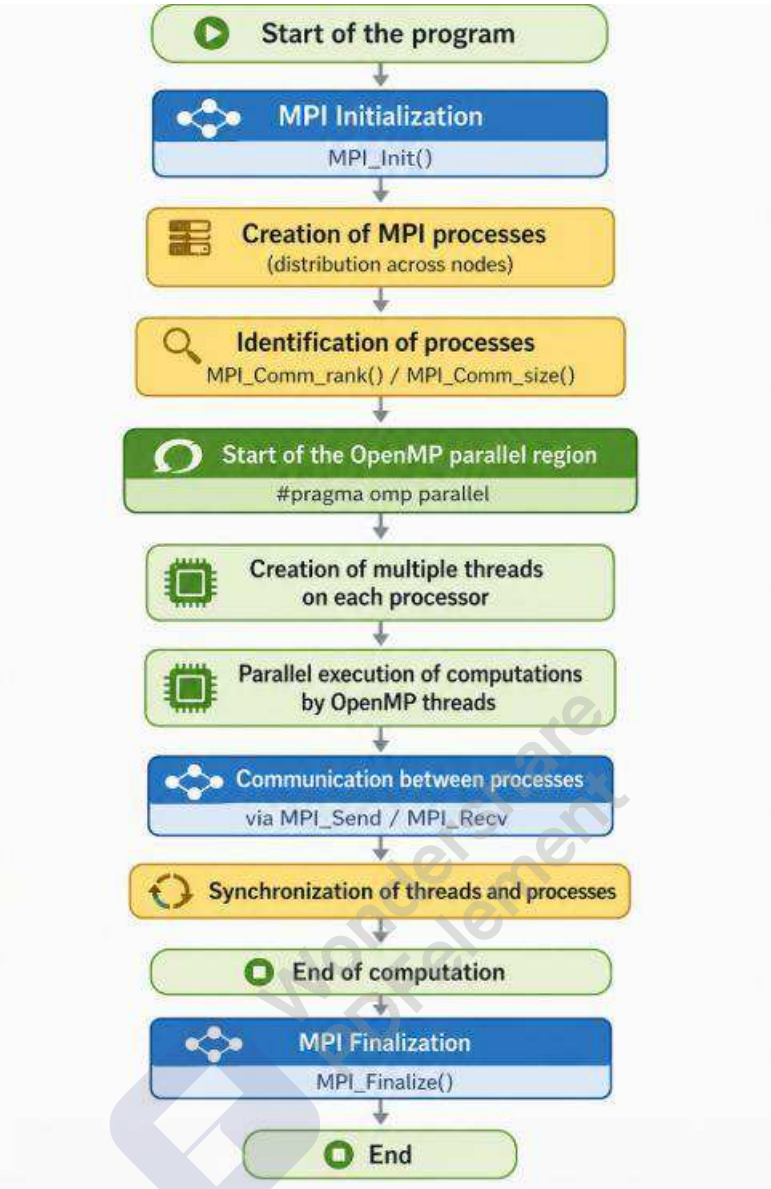
   Processes exchange data through message passing.
7. **Synchronization**

   Threads and processes synchronize before the end of computation.
8. **Finalization**

   The program terminates the MPI environment.

The following diagram illustrates the operating process of the hybrid MPI–OpenMP approach.

Remove Watermark Wondershare PDFelement



Operating Process of the Hybrid MPI–OpenMP Model with using Parallel Programming Interface Primitives

9.5. Example

```
//In programming language c :
#include <mpi.h>
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
```

Remove Watermark Wondershare PDFelement


```
{   int rank, size;
MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   #pragma omp parallel
   {
      int thread_id = omp_get_thread_num();
      printf("Processus MPI %d, Thread OpenMP %d\n", rank, thread_id);
   }

   MPI_Finalize();
   return 0;
}
```

Possible results :

Processus MPI 0, Thread OpenMP 0
Processus MPI 0, Thread OpenMP 1
Processus MPI 1, Thread OpenMP 0
Processus MPI 1, Thread OpenMP 1

Wondershare PDFelement

Key points :

- Each MPI process independently spawns its own OpenMP threads.
- Consequently, Thread 0 in Process 0 is different from Thread 0 in Process 1.
- These threads belong to separate processes and operate in distinct local memory spaces, consistent with the distributed memory model.

9.6. Comparison of Parallel Programming Models or Paradigms

A comparison between the three types of parallel programming paradigms is summarized in the table below:

Criterion	Shared Memory Model	Distributed Memory Model	Hybrid Model
Memory Organization	A single shared memory accessible by all threads	Each processor has its own local memory	Combination of shared and distributed memory
Communication	Through shared variables	Through message passing	Message passing between nodes and shared memory within each node

Remove Watermark Wondershare PDFelement

Criterion	Shared Memory Model	Distributed Memory Model	Hybrid Model
Synchronization	Requires mechanisms such as mutexes and semaphores	Synchronization via explicit communication	Combination of both mechanisms
Programming Complexity	Relatively simple	More complex	More complex as it combines both models
Scalability	Limited to a small number of processors	Highly scalable for large systems	Very efficient for modern architectures
Example Interface	OpenMP	MPI	Combination of MPI + OpenMP

Comparative Table of Parallelization Paradigms

10.Exercises

Exercise 1: Consider 4 processes: P0, P1, P2, P3. Each process has an array of 4 elements:

P0: [1 2 3 4]
P1: [5 6 7 8]
P2: [9 10 11 12]
P3: [13 14 15 16]

1) Apply All To All and give the final contents of each process.

2) Apply Reduce(sum) column by column to P0. What is the final result?
3) Apply Reduce_Scatter(sum) and give the final result in each process.
4) Apply Scan(sum) on the rows (element by element).
5) After an AllGather on the results of question 3, what does each process contain?

Exercise 2: Find the minimum of an array in OpenMP

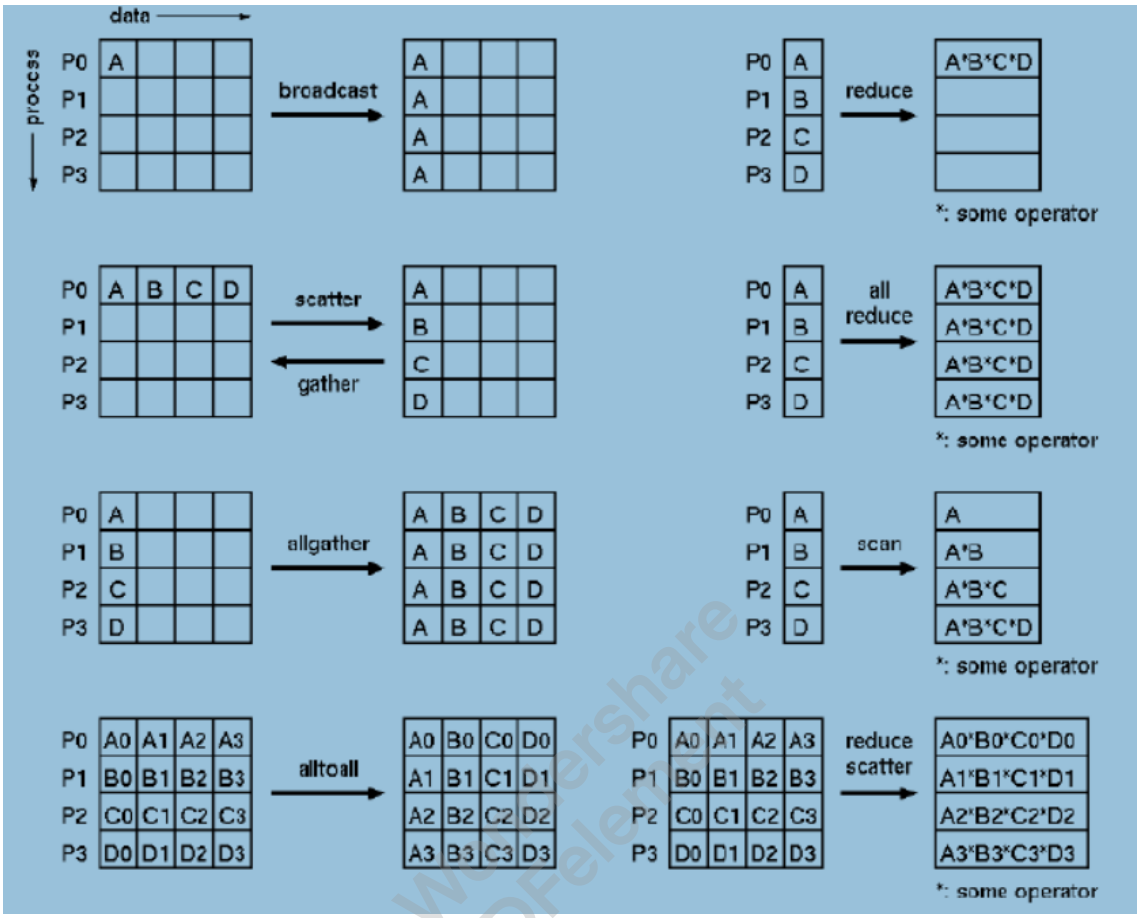
Exercise 3: Parallelize the following loop with OpenMP:

```
for (int i = 0; i < n; i++)
C[i] = A[i] + B[i];
```

Exercise 4: Write an OpenMP version that calculates the sum of an array tab[]

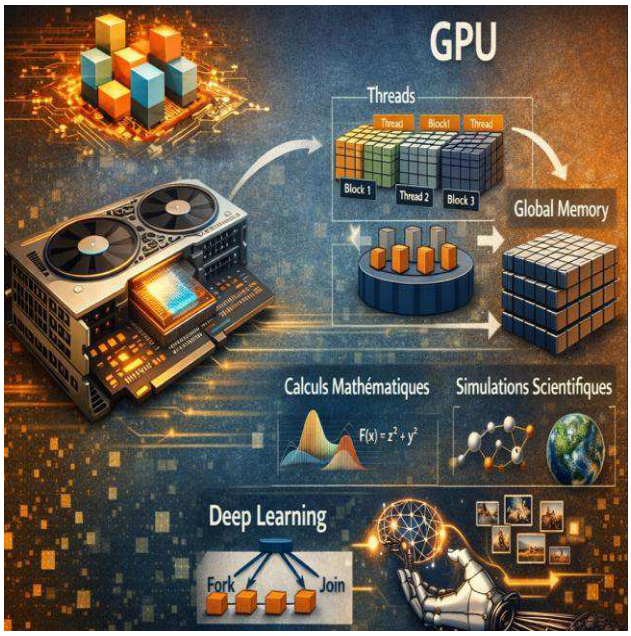
Remove Watermark Wondershare PDFelement

Exercise 5: Consider the following image:



i. Give the result (inputs and outputs) of each operation shown in this diagram.
ii. What does the diagram above represent?

Remove Watermark Wondershare PDFelement



Chapter 4:

General-purpose computing on a graphics processing unit (GPU)

Key Words

Parallel computing;GPU; GPGPU; CUDA; OPenCL

Abstract

The utilization of the parallel computing capabilities of graphics cards to execute tasks beyond image rendering is known as general-purpose computation on graphics processing units (GPUs). Modern GPUs are distinguished by their ability to simultaneously process a large number of operations through *thousands of cores. There are several types of GPUs, integrated or external, whose performance varies depending on their design and intended use.*

Graphics Processing Units (GPUs) now play a central role in modern computing due to their ability to process a very large number of operations simultaneously. Unlike traditional Central Processing Units (CPUs), which are mainly designed for complex sequential tasks, GPUs are particularly well suited for massively parallel processing. This makes them essential in fields such as artificial intelligence, scientific simulation, data analysis, and image processing. This evolution has led to the emergence of the GPGPU (General-Purpose Computing on Graphics Processing Units) concept, which involves using the computational power of GPUs for applications far beyond simple graphics rendering.

To efficiently exploit this architecture, several programming models and interfaces have been developed. Among them, CUDA, developed by NVIDIA, enables parallel computing by executing thousands of tasks simultaneously on the GPU. OpenCL provides an open and portable alternative compatible with different hardware vendors and processor types. Thanks to these technologies, GPUs have become essential tools for accelerating intensive computations and meeting the growing performance demands of modern scientific and industrial applications.

1. Introduction

Les GPU ont progressivement évolué vers des plateformes de calcul générique, donnant naissance au paradigme du GPGPU (General-Purpose computing on Graphics Processing Units). Cette évolution a ouvert de nouvelles perspectives pour l'optimisation des

Remove Watermark Wondershare PDFelement

performances dans divers domaines, notamment le calcul scientifique, l'apprentissage automatique et les systèmes énergétiques intelligents.
The objective of this chapter is to present a structured overview of the fundamentals of general-purpose computing on GPUs. First, we will examine GPU architectures and components, as well as their key differences from central processing units (CPUs). Second, we will analyze parallel programming models, focusing on the organization of threads, blocks, and grids, as well as the memory hierarchy. Finally, this chapter will introduce the main programming interfaces, notably CUDA and OpenCL, to understand their role in the development of high-performance parallel applications.

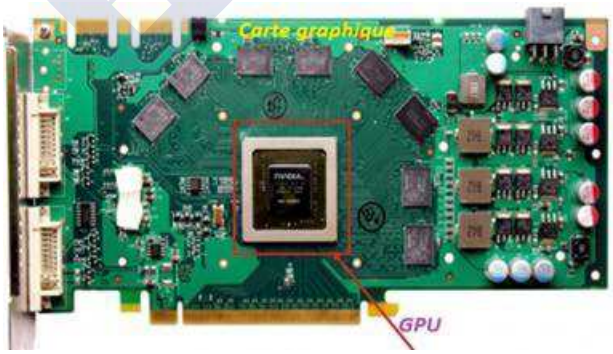
2. Architectures and GPU concepts (Graphic Processing Unit)

This section focuses on GPU architecture, components, and types.

2.1 Introduction to GPUs

The graphics card, also called a video card, is the component responsible for generating the images displayed on the screen. It can be integrated into the processor (integrated GPU) or installed as an external card (dedicated GPU), allowing the user to choose the graphics power required according to their needs.

At the heart of the graphics card is the GPU (*Graphics Processing Unit*), a specialized processor responsible for computing 2D and 3D graphics as well as video processing, as shown in the figure below. The GPU works in conjunction with the video memory (VRAM), which temporarily stores data and images before sending them to the monitor. The GPU can be considered the equivalent of the CPU for graphics processing: it manages computations, while the video memory handles the fast storage of intermediate data. Thanks to this combination, the graphics card is capable of performing massively parallel tasks, not only for image and video rendering, but also for scientific computing, deep learning, and other GPGPU applications.



Integration of the GPU component in the graphics card

A GPU is a specialized processor initially designed for graphics rendering and image processing. It has a large number of single cores capable of executing thousands of operations in parallel, making it particularly well-suited for massively parallel computing, scientific processing, and GPGPU (GPU general-purpose computing). Unlike CPUs, which are optimized for sequential computing and low latency, GPUs prioritize throughput to quickly process large amounts of data.

Remove Watermark Wondershare PDFelement

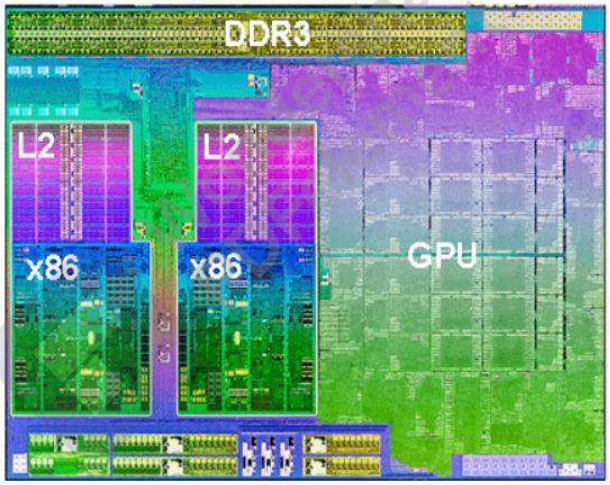
2.2. Types of GPU

GPUs can exist in different forms depending on their integration and intended use:

1. Integrated GPU

Most graphics processing units available today are integrated into the Central Processing Unit (CPU) and installed directly on the motherboard. As a result, the graphics card is no longer a separate component but becomes part of the processor's physical package. An integrated GPU is directly embedded into the CPU or the motherboard chipset. It generally shares the system memory with the CPU instead of using dedicated memory.

- Advantages: low cost, low power consumption, compact design.
- Disadvantages: limited performance, less suitable for massively parallel computing, gaming, or demanding scientific applications.
- Examples: Intel UHD Graphics, integrated AMD Radeon Vega.



Integrated GPU: This AMD Trinity chip integrates a GPU with four x86 processing cores and a DDR3 memory controller. Each x86 section is a dual-core processor with its own L2 cache. (Image courtesy of Advanced Micro Devices, Inc.)

Remove Watermark Wondershare PDFelement

2. Discrete / External GPU) : A dedicated GPU is an external graphics card equipped with its own memory (VRAM) and graphics processor. It is connected to the motherboard through a PCIe bus.

- **Advantages:** high computational power, fast dedicated memory, ideal for scientific computing, deep learning, 3D rendering, and gaming.
- **Disadvantages:** higher cost, greater power consumption, larger physical size.
- **Examples:** NVIDIA GeForce RTX, AMD Radeon RX, professional NVIDIA Quadro cards, and AMD Radeon Pro.



Dedicated GPU: NVIDIA GeForce 6600 GT graphics processing unit

2.3. Comparison of CPUs and External GPUs

Unlike the CPU, which has a few powerful cores optimized for sequential processing, the GPU contains thousands of lightweight cores organized into Streaming Multi-processors (SMs), each capable of handling 32-thread warps according to the SIMT (Single Instruction, Multiple Threads) model. A comparison between CPUs (Computing Processing Units) is given as follows:

Remove Watermark Wondershare PDFelement

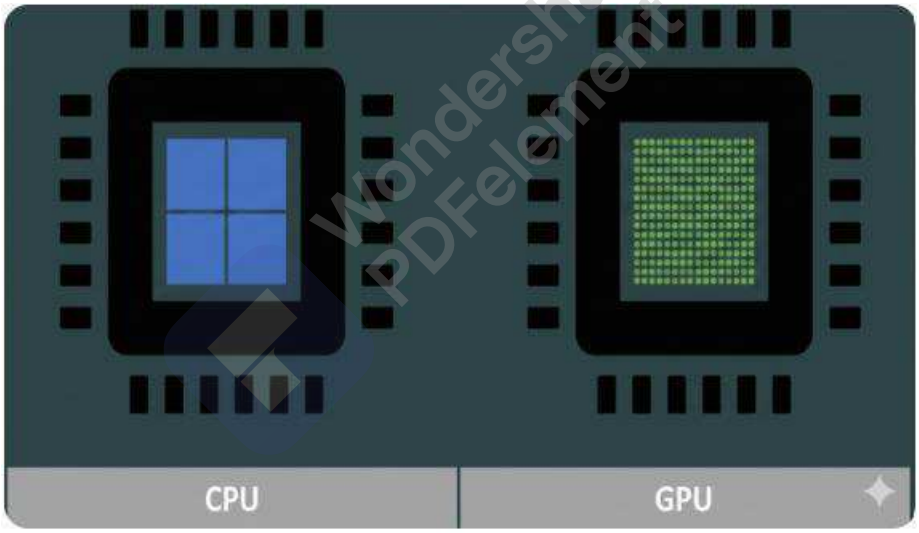
CPU :

- **General computing:** The CPU is preferred for executing tasks that require fast sequential processing, such as running software applications, web browsing, and standard office tasks.
- **Real-time decision making:** Applications requiring high responsiveness, such as interactive gaming or database queries, benefit from the CPU's ability to efficiently handle single-threaded tasks.

GPU:

- **Parallel computing:** The GPU excels at massively parallel computations, particularly for rendering complex graphics, processing large datasets, and training machine learning models.
- **Specialized workloads:** Fields such as scientific research, video editing, and cryptocurrency mining fully exploit GPUs due to their ability to efficiently handle intensive parallel tasks.

The following diagram provides a comparison between a CPU and a GPU.



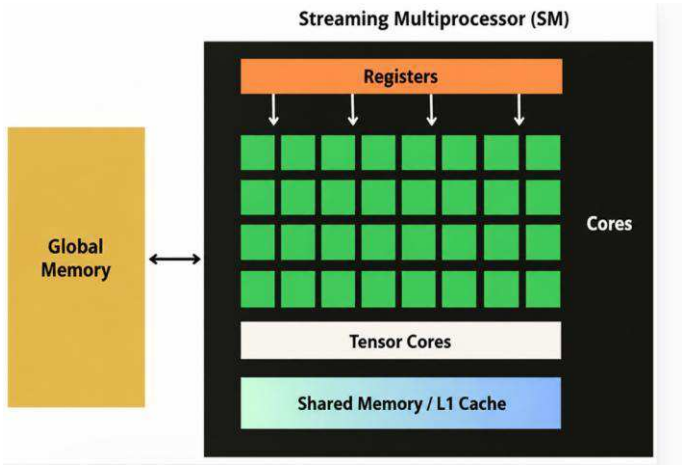
CPU Architecture to GPU Architecture

2.4. Components of GPU

A GPU is composed of several components that enable parallel computing.

1. **CUDA cores / Stream processors:** These are small cores optimized for massive parallel computing. Each Streaming Multi-processor (SM) contains multiple cores capable of performing arithmetic and logical operations on data simultaneously. Some modern GPUs include Tensor Cores, which are specialized computing units designed primarily for artificial intelligence and deep learning.

Remove Watermark Wondershare PDFelement



Streaming Multi-processors

2. **Internal memory (SRAM):** This ultra-fast memory includes:

   - **Registers:** used to store immediate values for the cores.
   - **L1 Cache:** first-level cache dedicated to each SM for frequently used data.
   - **L2 Cache:** shared cache among multiple SMs to reduce accesses to global memory.

Main memory (VRAM / DRAM / HBM): This memory stores large amounts of data such as textures, model weights, and buffers. It is slower than SRAM but offers a much larger capacity. High-performance GPUs often use HBM (High Bandwidth Memory) for AI and intensive computing workloads.

Control units and thread scheduler: These components manage the assignment of threads and blocks to SMs, synchronize operations, and optimize GPU resource utilization.

Memory interfaces and buses: They connect the GPU to the rest of the system and to VRAM, ensuring efficient data transfer between global memory and local caches.

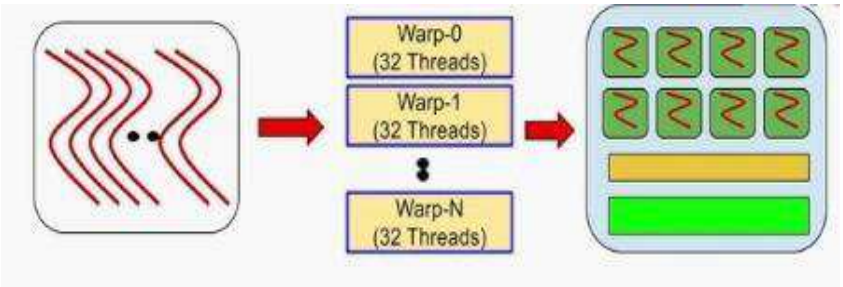
Warps (execution unit): Threads within the same block are grouped into warps of 32 threads. The SM warp scheduler selects ready warps for execution and hides memory latency by rapidly switching between active warps. If threads within the same warp follow different execution paths (e.g., if/else conditions), execution becomes sequential, reducing effective parallelism. Therefore, algorithm optimization aims to minimize this divergence.

Thread block: When processing exceeds the size of a warp, the GPU divides data into blocks (tiling). Each block is processed independently, and partial results are then combined. A thread block is caracterized by :

   - Contains multiple warps (groups of 32 threads) (see figure below,

Remove Watermark Wondershare PDFelement

- Has shared memory accessible to all its threads and
- Can synchronize the execution of its threads to collaborate efficiently



Threads grouped into Warps

3. GPU Programming Model

This model is also known by the acronym GPGPU (General-Purpose GPU). This type of computing involves diverting the GPU's processing power to run non-graphics applications, such as scientific simulations, machine learning, image processing, or data analysis.

3.1 Principles of GPU-based Generic Computing (GPGPU)

GPU-based generic computing involves using the graphics processing unit (GPU) to perform calculations that are not solely related to graphics rendering. GPUs have a large number of cores capable of executing multiple operations in parallel, which significantly accelerates certain scientific and technical applications.

The principle of GPU-based generic computing is based on the Single Instruction, Multiple Threads (SIMT) model, where the same instruction is executed simultaneously by several threads on different data. The program is generally structured in two parts: a part executed on the CPU (called the host), which prepares the data and initiates the operations, and a part executed on the GPU (called the device), where intensive calculations are performed in the form of parallel kernels. The CPU generally acts as the program controller, while the GPU executes the parts of the code requiring a large number of parallel operations. This model makes it possible to significantly accelerate applications with a high degree of parallelism, by efficiently utilizing the thousands of cores present on a GPU.

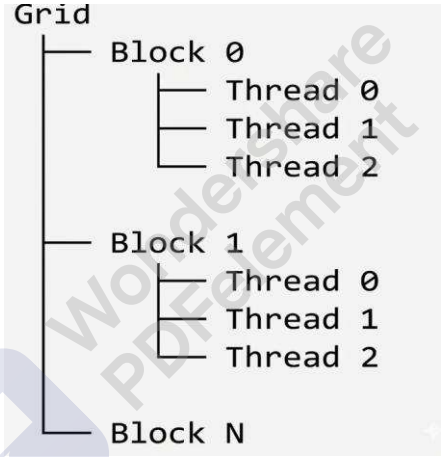
Remove Watermark Wondershare PDFelement

GPUs were originally designed for image and graphics processing. However, thanks to their massively parallel architecture, they are now used in many fields such as scientific computing, artificial intelligence, numerical simulation, and large-scale data processing.

The most widely used technologies for GPU programming include the CUDA platform developed by NVIDIA and the OpenCL interface, which allows parallel programs to run on different types of processors. More details are provided in the following sections.

3.2. GPU Programming Architecture (Threads, Blocks, Grid)

The GPU programming architecture is based on a hierarchically organized parallel execution model. In this model, calculations are performed by a large number of threads organized into blocks, which are themselves grouped in a grid. This organization allows for the efficient exploitation of the massive parallelism offered by GPUs. It is illustrated by the diagram below.



Hierarchical Organization for GPU Programming (CUDA)

This hierarchy **Grid → Block → Thread** forms the basis of GPU programming. GPU performance strongly depends on the organization of threads, blocks, and grids, and proper management of these elements allows full exploitation of the GPU's parallel computing power.

1. Threads

A thread is the smallest execution unit in a GPU program. Each thread executes an instance of a kernel (a function running on the GPU). Threads have the following characteristics:

- Executes a portion of the computation
- Has its own private registers
- Can access global memory and shared memory

Remove Watermark Wondershare PDFelement

A large number of threads run in parallel to process multiple data elements simultaneously.

2. Blocks (Thread Blocks)

Threads are grouped into blocks called thread blocks. They have the following characteristics:

- Threads within the same block can communicate with each other
- They share a shared memory space
- They can synchronize during execution

Each block is executed on a GPU multiprocessor.

3. Grids

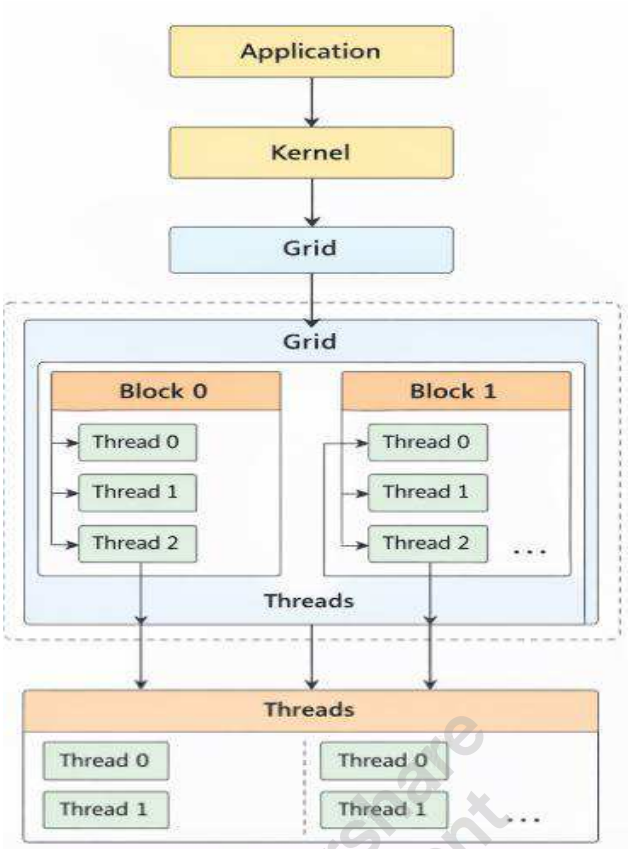
Blocks are organized into a structure called a grid, which has the following characteristics:

- A grid contains multiple blocks
- Each block contains multiple threads
- Blocks execute independently from one another

This organization allows thousands of threads to be executed in parallel on the GPU. The following diagram illustrates the steps of programming an application on a GPU. The next figure shows the execution process of an application under the GPGPU parallel programming paradigm.

Wondershare PDFelement

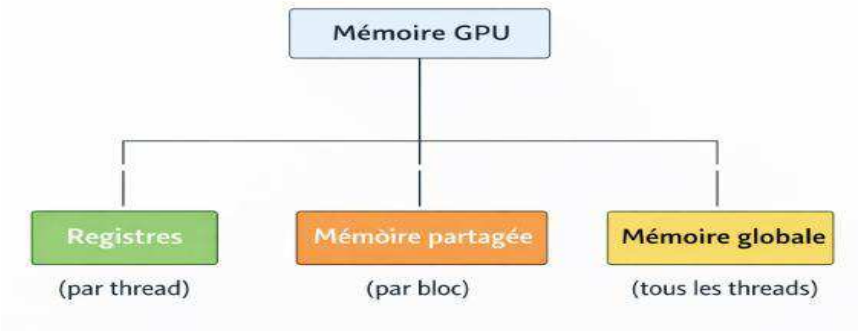
Remove Watermark Wondershare PDFelement



Decomposition of an Application for GPU Programming

3.3. GPU Memory Hierarchy

To optimize performance, GPUs contain several types of memory organized in a hierarchical structure. Each memory type has different characteristics in terms of speed, size, and accessibility. The main types of memory include registers, shared memory, global memory, constant memory, and texture memory.



GPU Memory Hierarchy

- **Registres (Registers) :** Les registres sont la mémoire la plus rapide du GPU.Cette mémoire possède les caractéristiques suivantes :
  - Mémoire privée pour chaque thread

Remove Watermark Wondershare PDFelement

- Utilisée pour stocker les variables temporaires
- Accès extrêmement rapide

- **Mémoire partagée (Shared Memory) :** La mémoire partagée est une mémoire rapide accessible par tous les threads d'un même bloc. Elle est caractérisée par :
  - Permettre la communication entre threads d'un bloc
  - Beaucoup plus rapide que la mémoire globale
  - Utilisée pour optimiser les performances
- **Mémoire globale (Global Memory) :** La mémoire globale est la mémoire principale du GPU. Elle est :
  - Accessible par tous les threads
  - Grande capacité
  - Plus lente que les registres et la mémoire partagée
- **Mémoire constante (Constant Memory) :** C'est est une mémoire en lecture seule. Elle est caractérisée par :
  - Accessible par tous les threads
  - Utilisée pour stocker des valeurs constantes
  - Optimisée pour les accès simultanés
- **Mémoire texture (Texture Memory) :** La mémoire texture est optimisée pour certains types d'accès mémoire, notamment dans le traitement d'images, les calculs graphiques et matrices et données 2D/3D. Elle est caractérisée par :
  - Mémoire en lecture seule
  - Mise en cache (cache texture)
  - Basée sur la mémoire globale

A comparative table of GPU memory types is established as follows :

Memory Type	Access	Speed	Usage
Global	Read/Write	Slow	General
Shared	Read/Write	Very fast	Block
Texture	Read-only	Fast (cached)	2D/3D data

Remove Watermark Wondershare PDFelement

Type de mémoire	Accès	Vitesse	Usage
Globale	Lecture/Écriture	Lent	Général
Partagée	Lecture/Écriture	Très rapide	Bloc
Texture	Lecture seule	Rapide (cache)	Données 2D/3D

The performance of a GPU strongly depends on the organization of threads, blocks, and grids, as well as the efficient use of the memory hierarchy. Proper management of these elements allows full exploitation of the GPU's parallel computing power.

4. GPU Programming with the CUDA Interface

The most widely used technologies for GPU programming include the CUDA platform developed by NVIDIA and the OpenCL interface, which enables parallel programs to run on different types of processors.

4.1 Presentation of CUDA

The CUDA architecture allows the computational power of graphics processors to be used for parallel applications. It is based on a hierarchical organization of GPU hardware resources to enable the simultaneous execution of a large number of threads. This architecture was developed by NVIDIA to facilitate GPU programming in fields beyond graphics processing.

4.2 CUDA Architecture

The CUDA architecture is based on two main components:

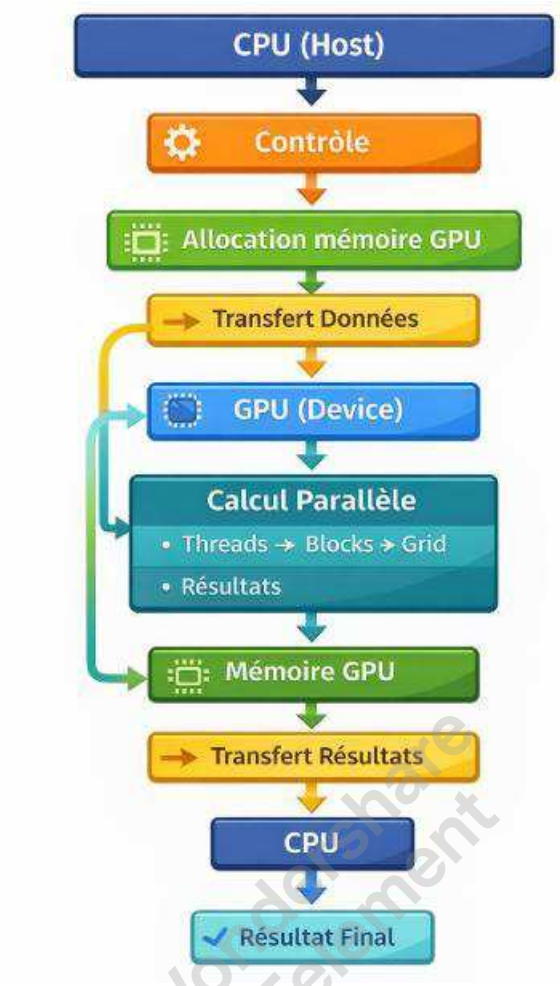
- **Host (CPU):** the central processor that executes the main program and launches computations on the GPU.
- **Device (GPU):** the graphics processor that performs parallel computations.

The CPU sends data and instructions to the GPU, which then executes operations in parallel using many processing cores.

4.3 CUDA Programming Workflow

The following diagram presents the CUDA programming process. This process consists of five successive operations: control operation, data transfer, kernel launch, GPU computation, and result transfer from GPU to CPU.

Remove Watermark Wondershare PDFelement



GPGPU Programming Model Process

Explanation of the programming process and interpretation

1. Control (CPU → GPU)

The CPU controls the execution of the CUDA program. The Host (CPU) is responsible for:

- managing the overall program execution
- preparing data to be processed
- configuring GPU parameters
- launching CUDA kernels
- coordinating data transfers

The GPU does not operate independently; it waits for instructions from the CPU.

Examples of CPU actions:

- calling cudaMalloc()
- calling cudaMemcpy()
- launching the kernel

Remove Watermark Wondershare PDFelement

2. Data Transfer (CPU → GPU)

This step involves copying data from the CPU memory to the GPU memory. Before parallel computation begins:

- data must be copied into the GPU global memory
- because the CPU and GPU have separate memory spaces

This operation is performed using the function cudaMemcpy(). It allows the GPU to access the data required for parallel computation.

3. Kernel Launch

The CPU triggers the execution of parallel computation on the GPU. A kernel is a special function executed on the GPU by many threads in parallel.

CUDA syntax:
kernel<<<Grid, Block>>>(parameters);

Where:

- **Grid** → number of blocks
- **Block** → number of threads per block

Example:
add<<<4,256>>>(A,B,C); → 4 blocks and 256 threads per block, meaning the GPU creates 1024 threads to perform the computation.

4. GPU Computation (Results)

GPU threads execute the kernel and compute results. During this stage:

- each thread runs the kernel
- threads are grouped into blocks
- blocks form a grid

The computed results are stored in the GPU global memory.

5. Result Transfer (GPU → CPU)

This step involves copying the results from the GPU back to the CPU. Once computation is completed:

- results remain in GPU memory
- they must be transferred to CPU memory

This is done using the cudaMemcpy() function.

Remove Watermark Wondershare PDFelement

5. GPU Programming with the OpenCL Interface

This section presents the OpenCL interface, GPU programming using this framework, the different directives used, and examples of kernel programs in the OpenCL environment.

5.1 OpenCL Présentation

OpenCL (Open Computing Language) is an open programming interface for parallel computing on heterogeneous systems, including CPUs, GPUs, FPGAs, and other specialized processors. It enables the development of applications that efficiently exploit available computing resources across different types of hardware. The main objectives of OpenCL are:

- To provide a universal standard for parallel programming on heterogeneous systems.
- To optimize the use of hardware resources (CPU, GPU, FPGA, etc.).
- To ensure application portability across different hardware platforms.

OpenCL is an open standard developed and maintained by the Khronos Group, the same organization responsible for OpenGL and Vulkan.

- It ensures interoperability between different hardware vendors.
- Recent versions of OpenCL include features for massively parallel computing and performance optimization on heterogeneous architectures.

5.2 OpenCL Programming Model

A. Principle of Parallel Computing

The OpenCL programming model is based on massive parallel computing to efficiently exploit heterogeneous architectures:

- Tasks are decomposed into smaller units called **work-items**, which can be executed simultaneously on different processing elements.
- Work-items can access different types of memory (local, global, private) depending on their granularity and scope.
- This approach accelerates applications requiring many independent computations, such as image processing, scientific simulation, or machine learning.

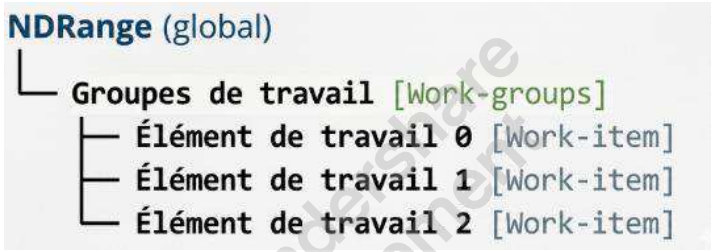
B. Work-items – Work-groups – NDRange Organization

OpenCL organizes parallel computation using a hierarchical structure:

Remove Watermark Wondershare PDFelement

1. **Work-item:**
   - The smallest execution unit in OpenCL.
   - Each work-item has a unique ID that allows it to access its own data.
2. **Work-group:**
   - A set of work-items grouped together to execute a kernel.
   - Each work-group has access to shared local memory, enabling communication between work-items within the same group.
3. **NDRange (N-Dimensional Range):**
   - The complete set of work-items organized in a 1D, 2D, or 3D structure.
   - Defines the global size (total work-items) and local size (work-items per work-group).
   - Maps computation directly onto the processing element grid of a device.

The figure below illustrates an example of hierarchical organization for GPU programming (OpenCL)



Hierarchical organization for GPU programming (OpenCL)

C. Execution of an OpenCL Program

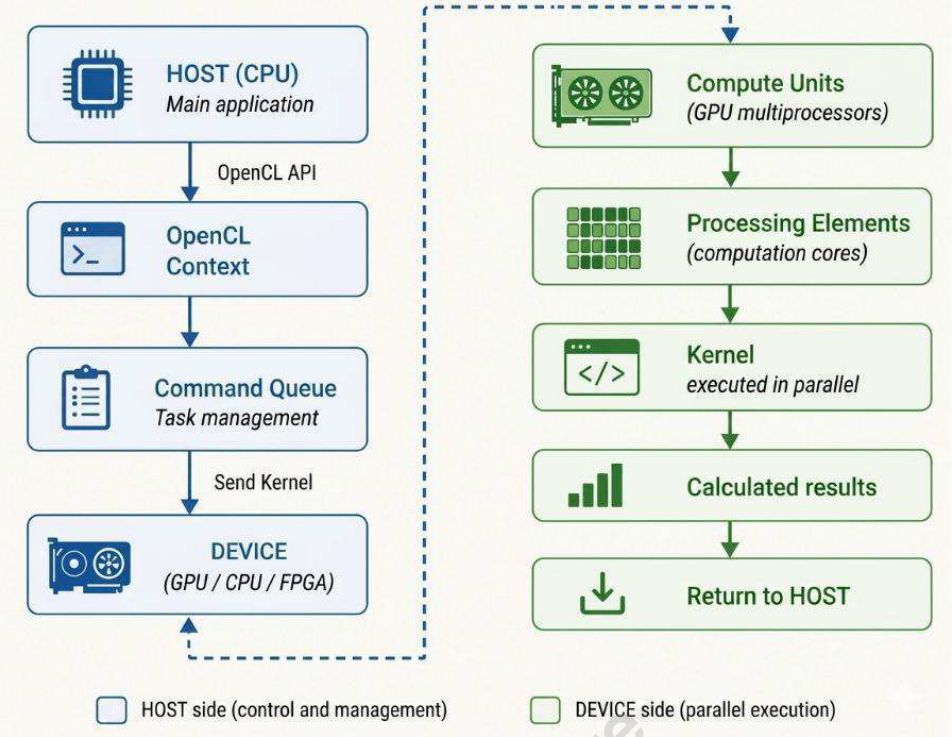
Host and Device

OpenCL distinguishes two types of components during program execution:

- **Host:** The CPU that controls program execution. It is responsible for:
  - Initializing OpenCL
  - Creating the context and managing devices
  - Allocating and transferring memory
  - Launching kernels on the device via the Runtime API
  - Retrieving results
- **Device:** The computing unit (GPU, CPU, FPGA, etc.) that executes the kernel in parallel. Each device is composed of Compute Units and Processing Elements used to execute work-items.

The following diagram illustrates the hierarchical structure of GPGPU programming under OpenCL.

Remove Watermark Wondershare PDFelement



Programming Process under OpenCL – GPGPU – Heterogeneous Architectures

Execution Process (Global Workflow)

- The Host prepares the data.
- The OpenCL kernel is compiled for the target devices.
- The Runtime API loads the kernel.
- The Host launches the execution.
- The devices execute the kernel in parallel.
- The results are returned to the Host.

D. OpenCL Program Example

This example shows how to add two vectors using an OpenCL kernel executed on a device (GPU or CPU).

1. OpenCL Kernel Code : The kernel is the function executed in parallel on the device.

- __kernel: indicates a function executed on the device.
- __global: global memory accessible by all work-items.
- get_global_id(0): identifier of the work-item in the 1D dimension.

Each work-item performs one addition operation.

Remove Watermark Wondershare PDFelement

Code Host (Main programme in C)

```
#include <CL/cl.h>
#include <stdio.h>
#define N 8
int main() {

float A[N] = {1,2,3,4,5,6,7,8};
float B[N] = {10,20,30,40,50,60,70,80};
float C[N];

cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;
clGetPlatformIDs(1, &platform, NULL);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
&device, NULL);

context = clCreateContext(NULL, 1, &device, NULL,
NULL, NULL);
queue = clCreateCommandQueue(context, device, 0,
NULL);
cl_mem bufferA = clCreateBuffer(context,
CL_MEM_READ_ONLY,
sizeof(float)*N, NULL, NULL);
cl_mem bufferB = clCreateBuffer(context,
CL_MEM_READ_ONLY,
sizeof(float)*N, NULL, NULL);
cl_mem bufferC = clCreateBuffer(context,
CL_MEM_WRITE_ONLY,
sizeof(float)*N, NULL, NULL);
clEnqueueWriteBuffer(queue, bufferA, CL_TRUE, 0,
sizeof(float)*N, A, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, bufferB, CL_TRUE, 0,
sizeof(float)*N, B, 0, NULL, NULL);

// compilation du kernel (simplifiée)
const char *kernelSource =
"__kernel void vector_add(__global const float *A,"
"__global const float *B,__global float *C) {"
"int id=get_global_id(0);"
"C[id]=A[id]+B[id];}";
cl_program program =
clCreateProgramWithSource(context,1,
&kernelSource,
 NULL, NULL);
```

Remove Watermark Wondershare PDFelement


```
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vector_add", NULL);

   clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufferA);
   clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufferB);
   clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufferC);
    size_t global_size = N;
   clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
                  &global_size, NULL,
                  0, NULL, NULL);

   clEnqueueReadBuffer(queue, bufferC, CL_TRUE, 0,
                sizeof(float)*N, C, 0, NULL, NULL);
printf("Resultat :\n");

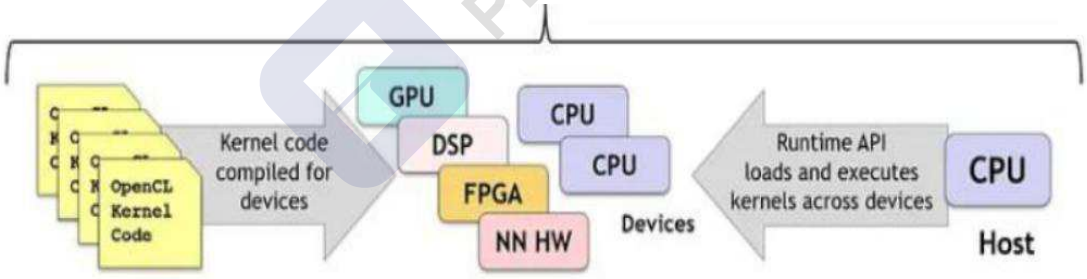
   for(int i=0;i<N;i++)
      printf("%f\n", C[i]);

   return 0;
}
```

Wondershare PDFelement

6. Exercises

Exercise 1: Consider the following diagram:



Q1_ Identification of the elements in the diagram
Q2_ Role of the CPU (Host)
Q3_ Role of the OpenCL kernel code
Q4_ Execution process (overall workflow)
Q5_ How the kernel is compiled for different devices
Q6_ Role of the Runtime API
Q7_ Why is this considered a heterogeneous GPGPU paradigm?
Q8_ CPU vs GPU comparison
Q9_ Why is OpenCL suitable for heterogeneous architectures?
Q10_ OpenCL (portability) vs CUDA (performance)

Remove Watermark Wondershare PDFelement

Q11_ Is this diagram valid for CUDA?
Q12_ Advantages and disadvantages
Q13_ Possible performance issues

Exercise 2: Vector Addition Performance Analysis

Task: Compute $C[i] = A[i] + B[i]$

- Size: $N = 1{,}000{,}000$
- Data type: float

Travail à faire:

1. Implement the parallel version using OpenCL (GPU) and measure execution time.
2. Implement the parallel version using OpenMP (CPU multi-core) and measure execution time.
3. Implement the sequential version (single CPU thread) and measure execution time.
4. Compare the execution times of the three approaches.
5. Compute and analyze the speedup between models.

Exercise 3: CUDA Dot Product (NVIDIA GPU)

Write a CUDA C program to compute the dot product of two vectors.

1. Case 1: Single block execution
   - Vector size: $N = 512$
   - The dot product must be computed using threads within the same block.
2. Case 2: Multiple blocks execution
   - Vector size: $N = 512$
   - The dot product must be computed using threads distributed across multiple blocks.

Exercise 4: CUDA Vector Addition with Multiple Blocks

Write a CUDA C program that performs the addition of two vectors of size $N = 1000$using an NVIDIA GPU.

- The computation must use threads distributed across multiple blocks.

Remove Watermark Wondershare PDFelement

General Conclusion

High Performance Computing (HPC) has evolved from a specialized research field into a fundamental pillar of modern scientific and industrial computing, driving innovation in data-intensive and computation-intensive applications.Throughout these lecture notes, we have studied the essential principles that enable the operation of HPC systems, ranging from parallel programming models and performance laws to modern system architectures, paradigms, and parallel programming interfaces such as MPI and OpenMP. Each of these elements plays a fundamental role in the development of efficient parallel systems capable of exploiting modern multi-core processors, distributed clusters, and GPU accelerators to address large-scale and computationally intensive problems while ensuring high performance and scalability. As High Performance Computing (HPC) continues to evolve, its applications are expanding across domains such as healthcare, finance, robotics, climate modeling, and scientific research, providing advanced capabilities for large-scale simulation, modeling, and data-intensive processing. However, these advances come with significant challenges, including the energy consumption of supercomputers, the complexity of parallel programming, the scalability of distributed systems, and the security of computing infrastructures. The future of High Performance Computing (HPC) will rely not only on advances in computing architectures and hardware accelerators, but also on the development of efficient, scalable, and energy-aware systems capable of delivering high performance while optimizing resource utilization for large-scale computing applications.

This course compendium has provided a structured pathway for understanding the fundamental mechanisms of High Performance Computing (HPC), giving readers the essential foundations to critically analyze and efficiently exploit parallel computing systems. Whether the objective is to pursue advanced research, develop HPC applications, or simply understand the computing infrastructures that support modern science, the principles presented here constitute an essential foundation.

Remove Watermark Wondershare PDFelement

Bibliographical References

Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture : A Quantitative Approach. Morgan Kaufmann.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). Introduction to Parallel Computing. Addison-Wesley.

Wilkinson, B., & Allen, M. (2005).. Parallel Programming: Techniques and Applications. Prentice Hall

Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. McGraw-Hill

Foster, I. (1995). Designing and Building Parallel Programs. Addison-Wesley.

Culler, D. E., Singh, J. P., & Gupta, A. (1999). Parallel Computer Architecture. Morgan Kaufmann

Hwang, K. (2013). *Advanced Computer Architecture*. McGraw-Hill.

Kirk, D. B., & Hwu, W. M. W. (2016). Programming Massively Parallel Processors. Morgan Kaufmann.

Sanders, J., & Kandrot, E. (2010). *CUDA by Example*. Addison-Wesley.

Rauber, T., & Rünger, G. (2013). Parallel Programming: For Multicore and Cluster Systems. Springer

Amdahl, G. M. (1967). Validity of the single processor approach. AFIPS.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. Communications of the ACM.

Karimi, K., et al. (2010). CUDA vs OpenCL performance comparison. arXiv.

Du, P., et al. (2012). From CUDA to OpenCL. Parallel Computing.

Owens, J. D., et al. (2008). GPU computing. Proceedings of the IEEE.

Remove Watermark Wondershare PDFelement

Flynn, M. J. (1966). Very high-speed computing systems. Proceedings of the IEEE

Gowanlock, M., & Karsin, B. (2019). Hybrid CPU/GPU sorting optimization. Parallel Computing.

Hijma, P., et al. (2023).. Optimization techniques for GPU programming. ACM Computing Surveys

AMD. AMD Radeon Cartes graphiques de bureau https://www.amd.com/fr/products/graphics/desktops/radeon.html (consulté le 21/11/2025)

OpenACC. Directive-based parallel programming for accelerators https://www.openacc.org (consulté le 21/11/2025)

NVIDIA Blog. https://blogs.nvidia.com (consulté le 21/11/2025)

Wondershare PDFelement