Parallel Algorithms and Programming

# Parallel architectures and programming models

**Thomas Ropars**

**Email:** thomas.ropars@univ-grenoble-alpes.fr

**Website:** tropars.github.io

# In this lecture

- Architecture of parallel computers
    - Flynn's taxonomy
- Models of parallel computation
    - SPMD -- MPMD
- Programming models and communication abstraction
    - Shared memory
    - Message passing
    - Data parallel

# References

The content of this lecture is inspired by:

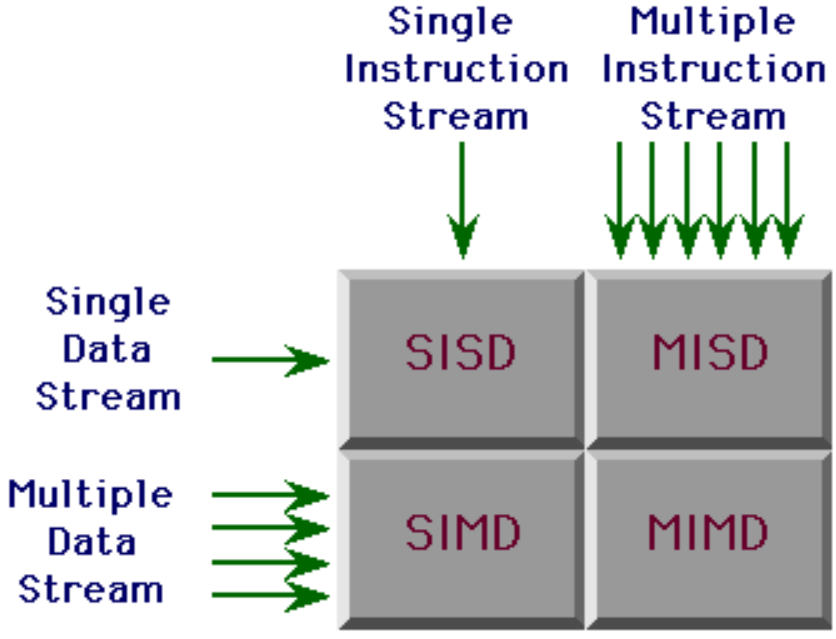
- The lecture notes of F. Desprez
- Introduction to Parallel Computing by B. Barney
- The lecture notes of K. Fatahalian
    - CS149: Parallel Computing @Standford
    - 15418: Parallel Computer Architecture and Programming @CMU
- Parallel Programming – For Multicore and Cluster System. T. Rauber, G. Rünger
- The lecture nodes of S. Lantz

# Architecture of parallel computers

# Classification of parallel computers

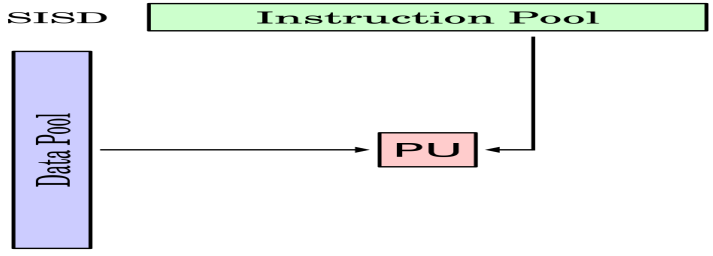
## Flynn's Taxonomy

- Proposed by Michael J. Flynn in 1966
- Overview of the design space of parallel computers
- Characterization according to the data flow and the control flow



# Single-Instruction, Single-Data

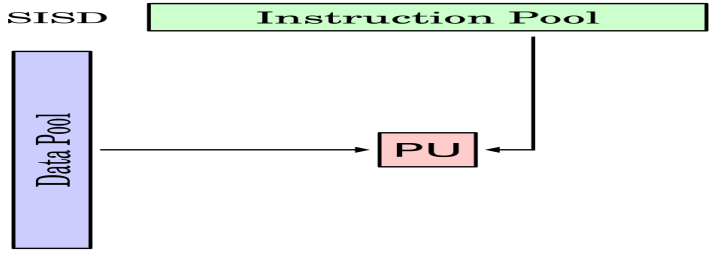
- One processing element
- It has access to a single program and a single data source




## Implementation

# Single-Instruction, Single-Data

- One processing element
- It has access to a single program and a single data source

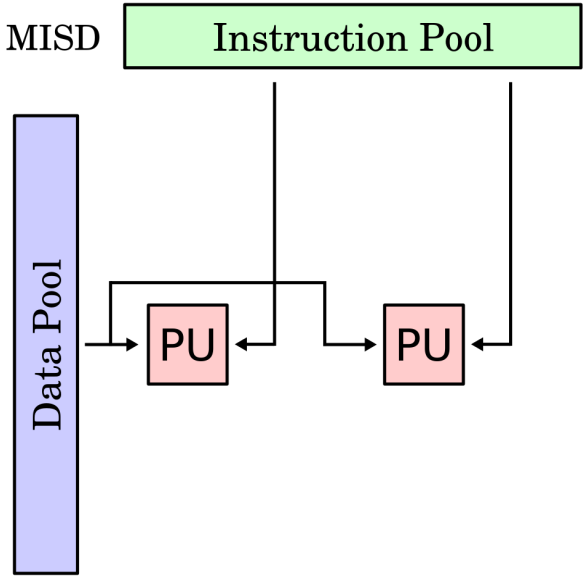



## Implementation

- Simple single processor architecture
- No parallelism

# Multiple-Instructions, Single-Data

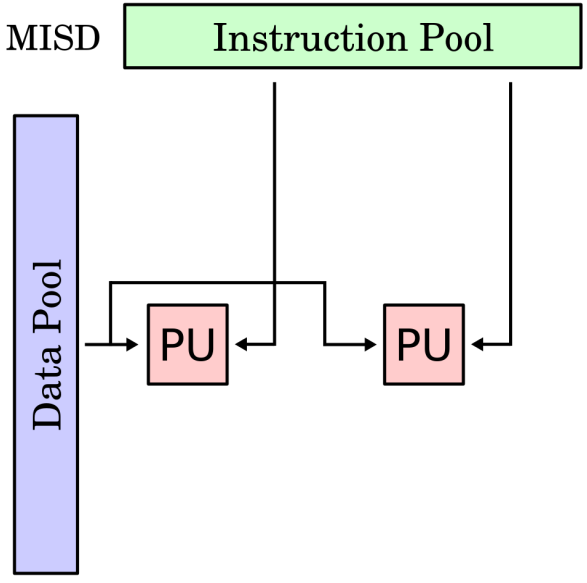
- Multiple processing elements
- Each one executes a different program on the same data




## Implementation

# Multiple-Instructions, Single-Data

- Multiple processing elements
- Each one executes a different program on the same data

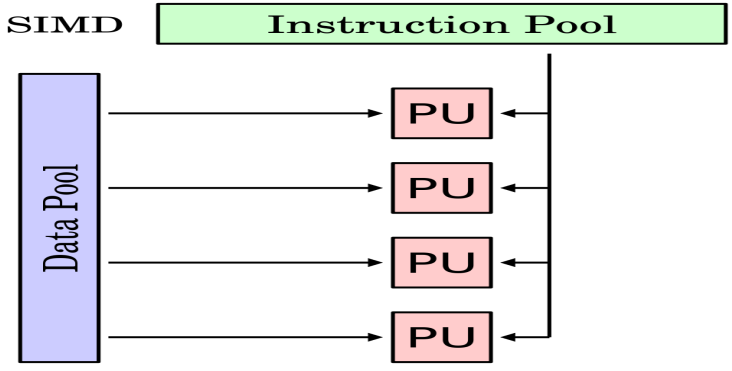



## Implementation

- Not a very common use case
- Security and fault tolerance
  - Hardware replication
  - Very high availability and safety requirements
- Majority voting where each processing unit executes a different algorithm
  - Triple Modular redundancy
- Space shuttles and planes
  - Boeing 777

# Single-Instruction, Multiple-Data

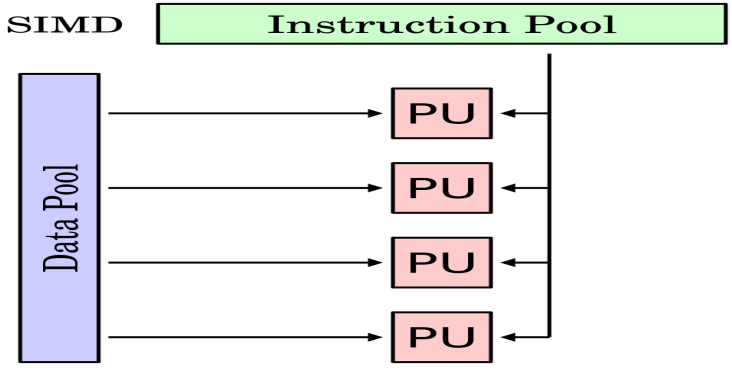
- Multiple processing elements
- Each one works on its own data
    - The physical memory might be shared
- All execute the same instruction at the same time




## Implementation

# Single-Instruction, Multiple-Data

- Multiple processing elements
- Each one works on its own data
    - The physical memory might be shared
- All execute the same instruction at the same time

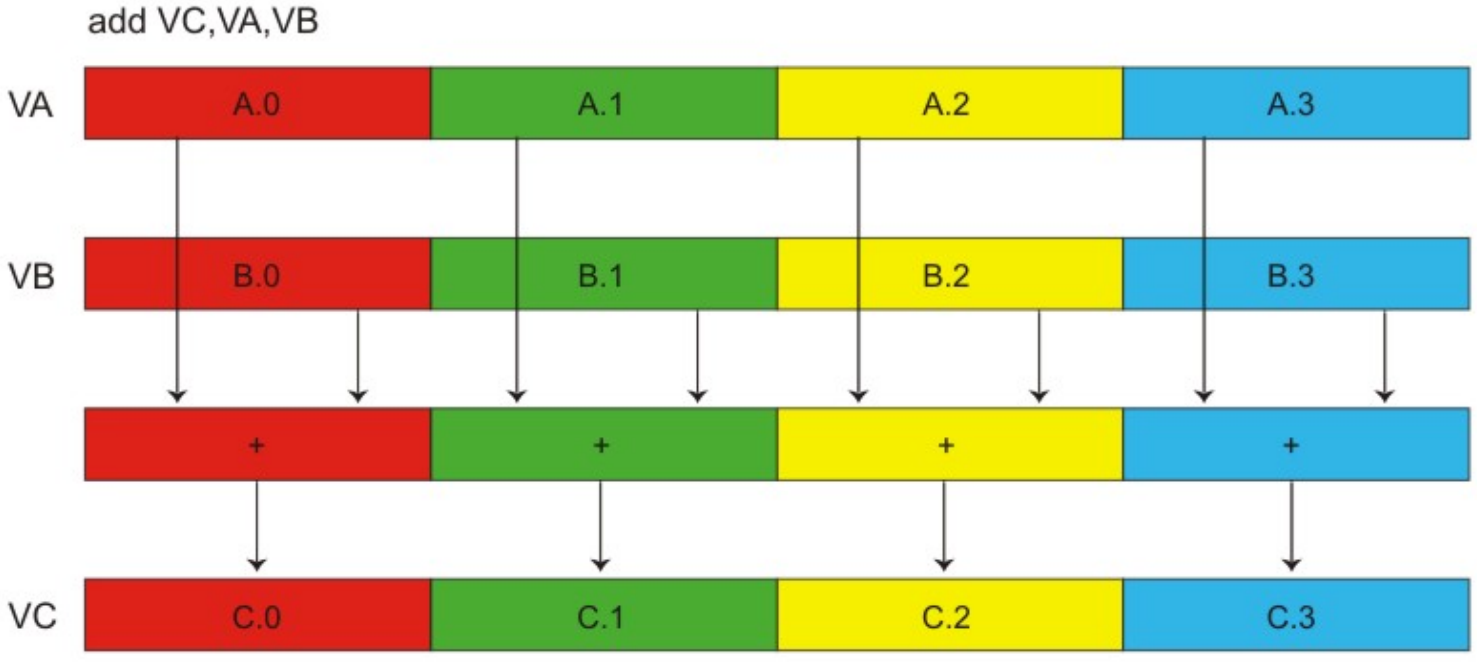



## Implementation

- Operations on vectors and matrices
- CPU
    - SSE and AVX vector operation extensions
- Basic architecture of GPUs

# Single-Instruction, Multiple-Data

## About vector operations




- SIMD registers can store multiple operands
- With AVX2 (latest Advanced Vector Extensions)
    - 512-bits registers (= 8 64-bit double-precision floating-point numbers)

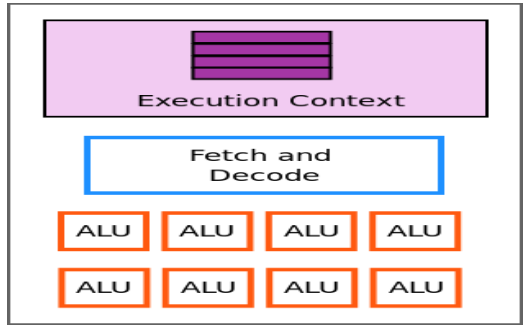
# Performance of SIMD

## A classic processor pipeline

1. **Fetch**: fetch the next instruction to be executed from memory
2. **Decode**: Decode the fetched instruction
3. **Execute**: Load the operands and execute the instruction
4. **Write-back**: Write back the result into a register

## Advantages of SIMD

- Amortize the cost and the complexity of managing instruction streams for multiple ALUs
- Same instruction broadcast to all ALUs




*Credit: K. Fatahalian*

# Performance of SIMD

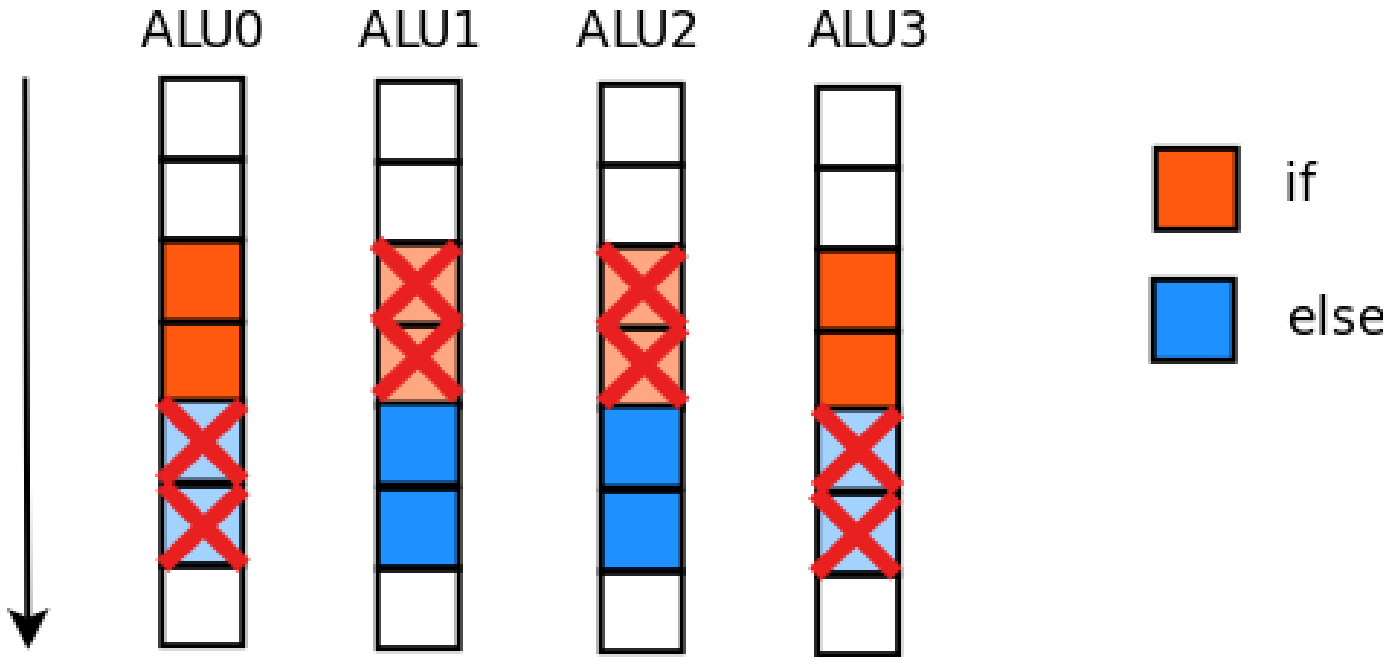
## Limitations

- Adapted to very regular processing

## The case of conditional branch

```
x = A[i];

if (x<0){
    y = x + K1;
    z = y * y;
}
else{
    y = x - K2;
    z = y;
}
```

# Performance of SIMD




- Not all ALUs do useful work
- In this case, only 50% of peak performance

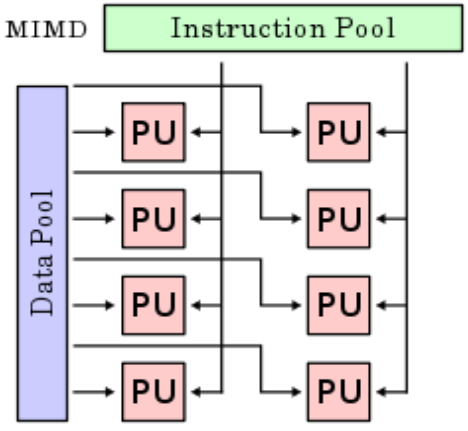
# SIMD in practice

## Instructions are generated by the compiler

- The compiler can find parallelism automatically by analyzing the dependencies between loops
    - Difficult problem
- The programmer can explicitly ask for SIMD parallelism
    - Using compiler intrinsics
- The programmer can give hints through annotations in a parallel programming language
    - *Parallel for* loops

# Multiple-Instruction, Multiple-Data

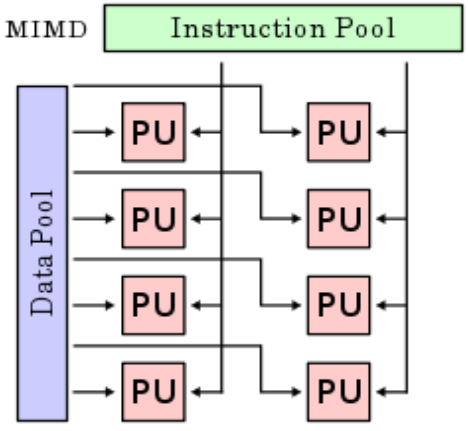
- Multiple processing elements
- Each one works on its own data
    - The physical memory might be shared
- Each one executes its own instruction stream.




**Implementation**

# Multiple-Instruction, Multiple-Data

- Multiple processing elements
- Each one works on its own data
    - The physical memory might be shared
- Each one executes its own instruction stream.




## Implementation

- Most parallel systems today
    - Multi-core processors
    - Cluster of commodity machines
    - Supercomputers
- Note that many MIMD architectures include SIMD sub-components

# Models of parallel applications

# Models of parallel applications

Two mains ways of structuring a parallel application.

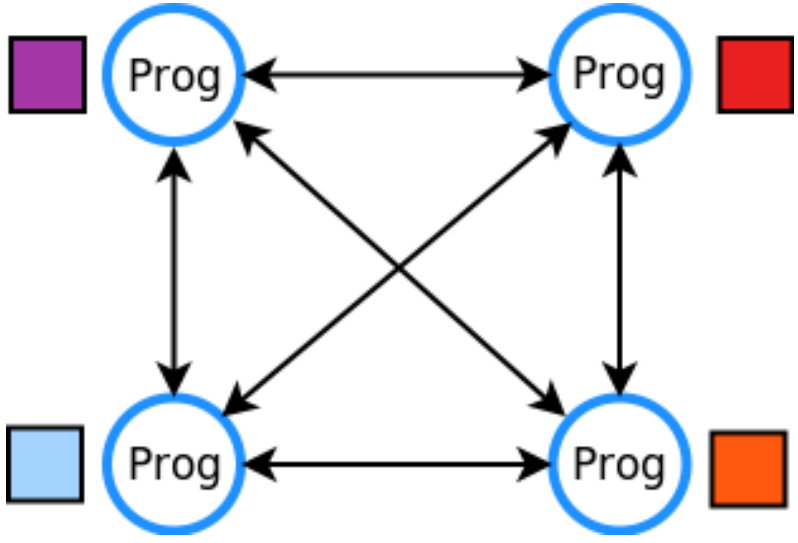
```
*SPMD*: Single program, multiple data

*MPMD*: Multiple programs, multiple data
```

Definition from the **point of view of the programmer**:

- The programmer sees her application as composed of multiple instruction streams:
    - Processes/Threads/Tasks
    - *Single program* means that all of them execute the same program
- A SPMD application could (theoretically) be translated into a single stream of SIMD instructions.
- Most often, we will execute our programs on MIMD architectures

# SPMD

- All threads/processes are executing the same program
    - Most parallel applications are SPMD
    - MPI (message passing), OpenMP (shared memory)
- They run computation on a different subset of the data
- They communicate via shared memory or message passing

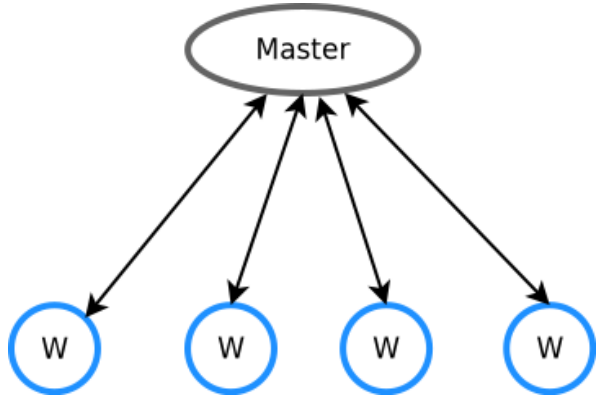


# MPMD

- Processes execute at least 2 different programs
    - Master-worker (also called master-slave)
    - Workflow of tasks

## Master-worker

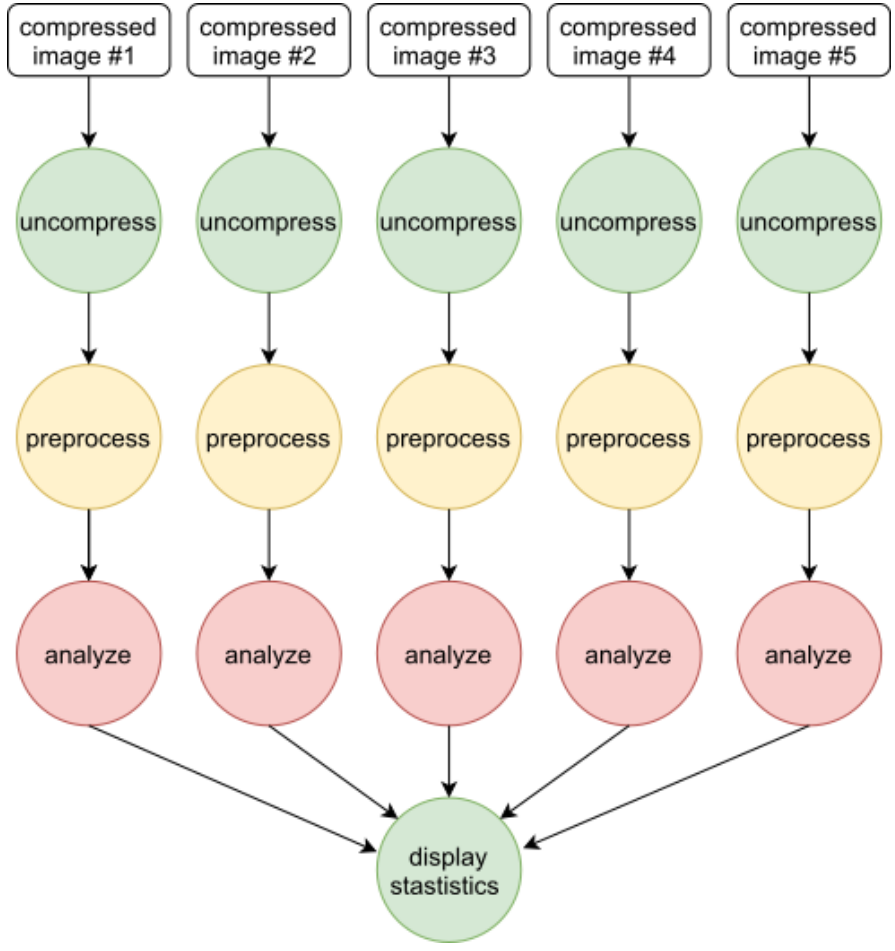
- Role of the master
    - Assigning work to workers
    - Coordinating the workers



# Example of MPMD application

## Image analysis workflow

- Assumes that *uncompress*, *preprocess*, etc. are implemented by different programs.



# Programming models

# Programming models

## 3 programming models

*What abstraction is offered to the programmer?*

1. Shared memory
2. Message passing
3. Data parallelism

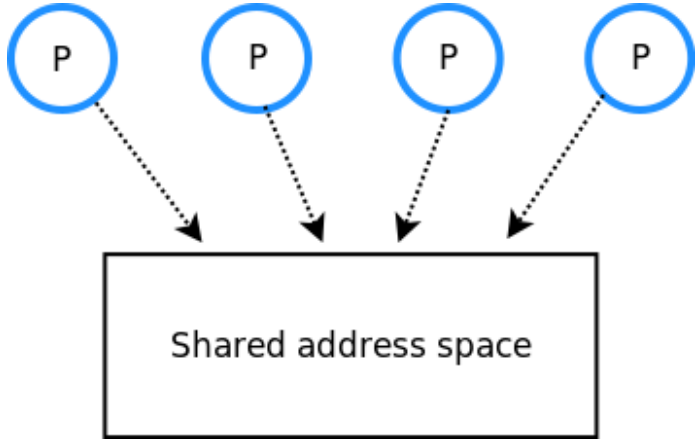
## Two communication architecture

*What is the communication interface offered by the hardware?*

1. Shared memory
2. Message passing

# Shared memory

```
Threads/processes communicate by running operations in a
*shared address space*
```



# Shared memory

## Communication

- Read and write operations in the shared address space
- Synchronization primitives (locks, semaphores, etc.)

## Comments

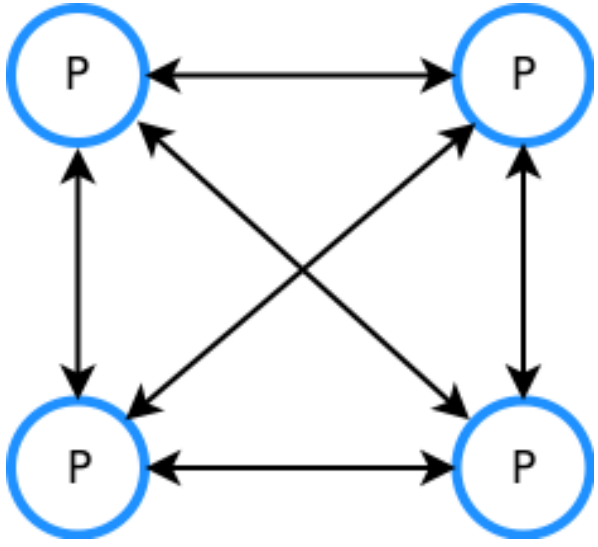
- *Simplest* way of parallelizing a sequential program
    - All the communications are implicit
    - Achieving good performance can be challenging

## Most common hardware implementation

- A multicore processor with coherent shared memory

# Message passing

```
Threads/processes communicate by *sending and receiving messages*
```




Also called **distributed memory** system

# Message passing

## Communication

- Sending and receiving messages

## Comments

- All communications are explicit
    - More effort for the programmer
    - Allow to better optimize performance
        - Control on the data movements
        - Better overlap between communication and computation

## Most common hardware implementation

- A set of computers interconnected through a network

# Programming model vs communication abstraction

The programming model is independent of the underlying communication architecture.

```
In general, if the programming model corresponds to the underlying
communication abstraction, better performance are to be expected.
```

# Programming model vs communication abstraction

```
Adopting a programming model on top of a different communication
abstraction also has advantages
```

## Message passing over shared memory

- Better control of the data movements than with shared memory
- Sending data is implemented by writing into a mailbox in shared memory
- Receiving data is implemented by reading in a mailbox
- Note that a shared memory system is ultimately a message passing system
    - A network interconnects the processor cores and the memory controllers

## Shared memory over message passing

- Simplified work for the programmer
- Distributed shared memory (Beyond the scope of this course)
    - Shared memory abstraction implemented in software
    - Partitioned Global Address Space (PGAS) adds the notion of locality

# Hybrid architectures and applications

## Parallel architectures

- A cluster of nodes each equipped with one or several multicore processors
    - Shared memory inside a node
    - Message passing between nodes

## Hybrid applications

- Applications combining message passing and shared memory programming models
    - Message passing: Processes execute on different nodes (MPI)
    - Shared memory: Each process is composed of multiple threads (OpenMP)

# Data parallelism

- A collection of data
- The programmer defines functions (transformation) to be applied on all the data
    - MapReduce programming model

## Transformation (Map)

- Defines a function to be applied on all elements independently
    - $map(f)[x_0, \ldots, x_n] = [f(x_0), \ldots, f(x_n)]$

```
map(*2)[2, 4, 5]= [4, 8, 10]
```

## Aggregation (Reduce)

- Defines a function to be applied on all elements (with the same key)
    - $reduce(f)[x_0, \ldots, x_n] = [f(x_0, f(x_1, \ldots, f(x_{n-1}, x_n)))]$

```
map(+)[2, 4, 5]= 2 + (4 + 5) = 11
```

# Data parallelism

- Very popular approach to process large amount of data
    - Hadoop MapReduce, Apache Spark, Apache Flink
- Allows expressing very simply some algorithms
    - Other algorithms might be very difficult to program efficiently

## Basic idea:

```
Reducing the flexibility on the operations that a programmer can run
```

- Allows implementing optimizations at the level of the middleware
- Avoids having to deal with many *exceptions* that can impair performance
- Allows implementing very efficient fault tolerance techniques

# An example of data parallel program: wordcount

## Description

```
*Input*: A set of lines including words

*Output*: A set of pairs < word, nb of occurrences >
```

## Input

```
< "aaa bb ccc" >
< "aaa bb" >
```

## Output

```
< "aaa", 2 >
< "bb", 2 >
< "ccc", 1 >
```

# An example of data parallel program: wordcount

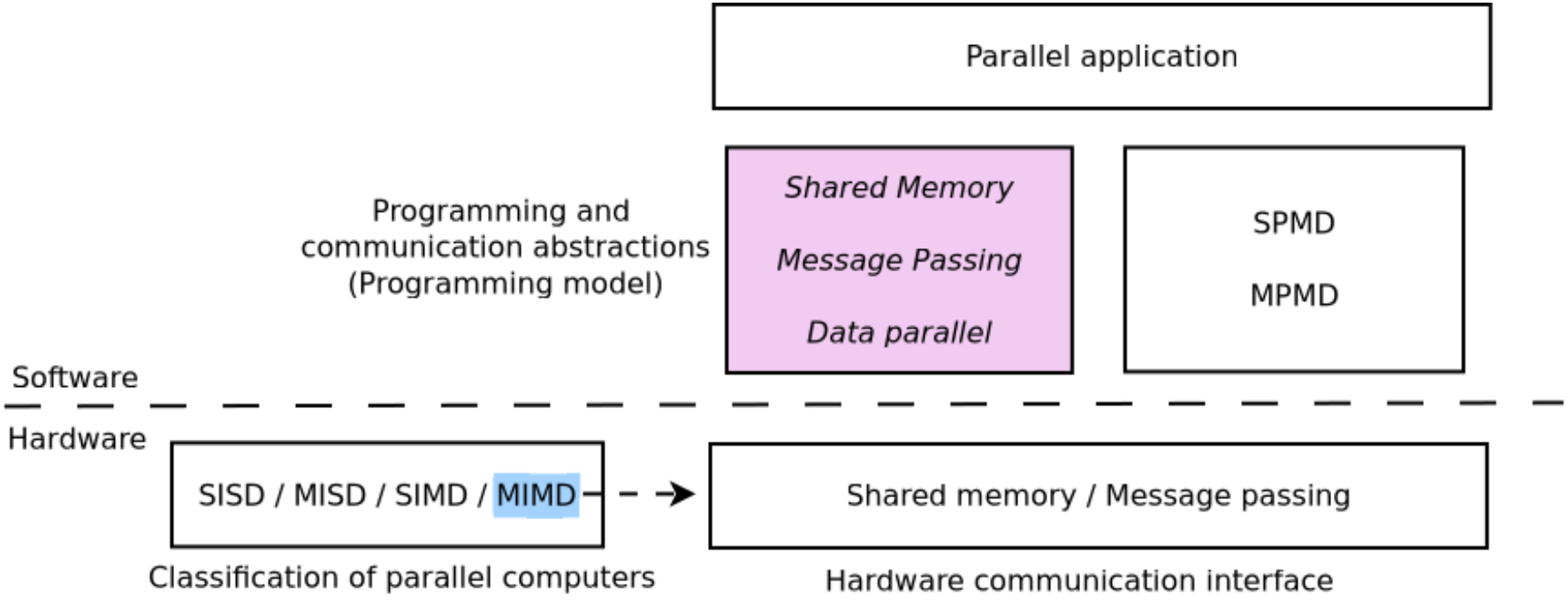
```
map(value):                 * each value = an input line *
    foreach word in value.split():
        emit(word, 1)



reduce(key, values):        * {word, collection of '1'} *
    result = 0
    for value in values:
        result += value
    emit(key, result)       * -> {word, nb occurences} *
```

# Implementation of data parallelism

- Data parallelism can be implemented as SPMD or MPMD programs
    - Most MapReduce frameworks are MPMD

- Data parallelism can be implemented on top of shared memory and message passing communication systems
    - It usually targets message passing systems
    - **Scale-out** systems

# Relation between the different concepts



# About programming models

- Programming models provide a way to think about the structure of parallel programs

- Trade-off between simplicity for the programmer and performance
    - A high-level abstraction
        - Easy to program
        - Might be difficult to optimize for performance
    - A low-level abstraction
        - Harder to program
        - The programer has a better control of what is happening

- Having well-defined abstractions help designers of hardware/compilers/runtimes to think about optimizations that can improve the performance of parallel programs

# Conclusion

- Complex design space with multiple levels of abstraction
    - Architecture of parallel computers
    - Models of parallel computation
    - Programming models
    - Communication abstraction

- Different programming models, all with advantages and drawbacks
    - Shared memory
    - Message passing
    - Data parallelism

# Additional slides

# About message passing and shared memory

## The case of manycore processors

# Manycore processors

Several processor architecture featuring a high number of cores have emerged:

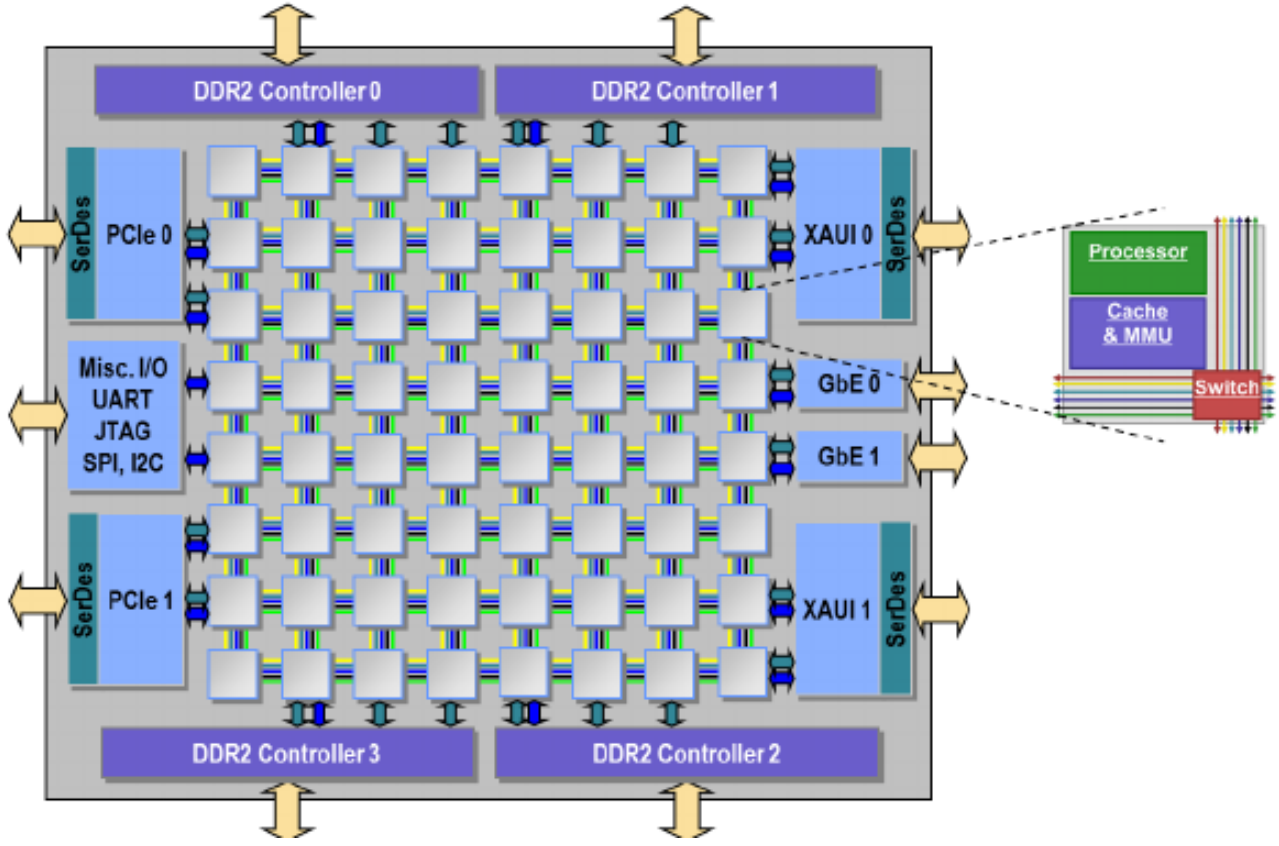
- 72-core tilera processor
- 260-core Sunway processor
- 80-core Kalray processor
- etc.

All these architectures allow to communicate via message passing on the chip:

- Tilera also offers a coherent shared memory
- On other processors, not all cores have access to the same global coherent shared memory

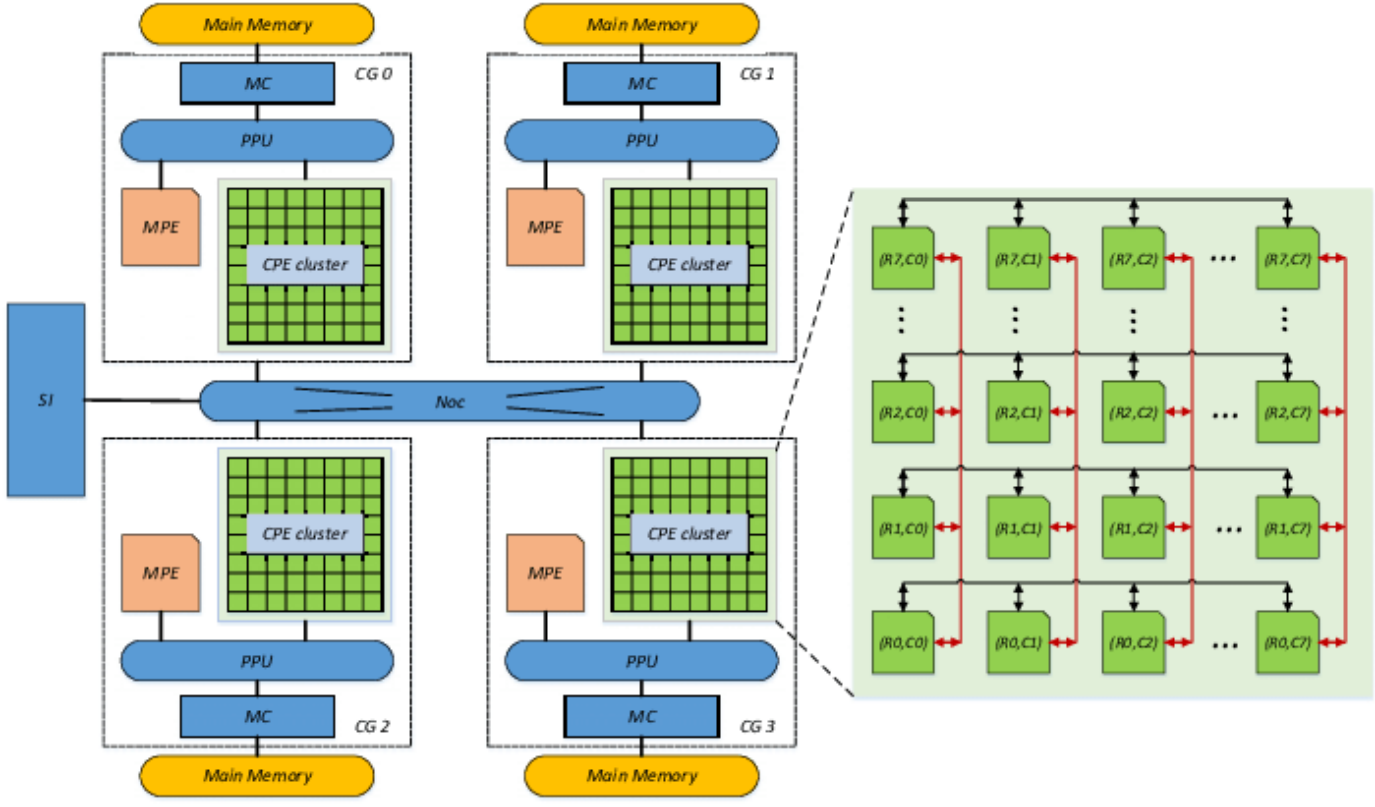
# Manycore processor

## Tilera processor



# Manycore processor

## Sunway SW26010



# Shared memory vs message passing

## Limits of coherent shared memory

- Cost of maintaining coherence
- Coherence protocols are mostly based on broadcast protocols
    - Scalability of such protocols ?
    - Efficiency ? (performance and energy)
- Difficult to build a performance model of the system
    - Many implicit interactions that are difficult to model

## About message passing

- Control of the data movements
    - Scalability and energy efficiency
    - Performance
    - Increases the determinism of the system
- No direct support for legacy applications
- About performance of message passing vs shared memory:
    - *Leveraging hardware message passing for efficient thread synchronization* by D. Petrovic et al.