Parallel Algorithms and Programming

Architecture, programming and communication models

Thomas Ropars

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

Website: tropars.github.io

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

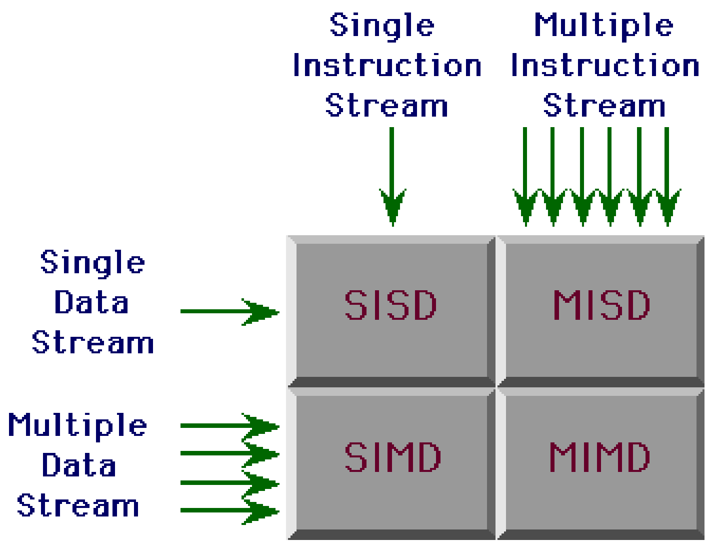
Outline

- Architecture of parallel computers
- Models of parallel computation
- Programming models and communication abstraction

Classification of parallel computers

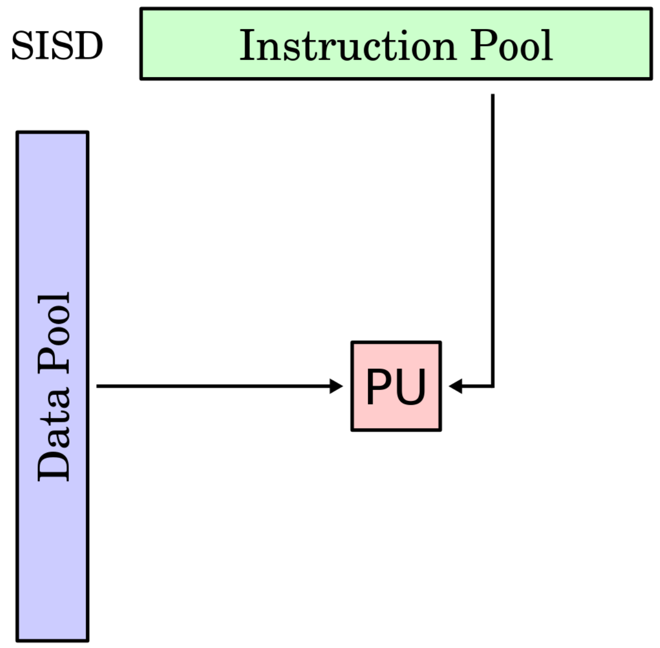
Flynn's Taxonomy

- 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



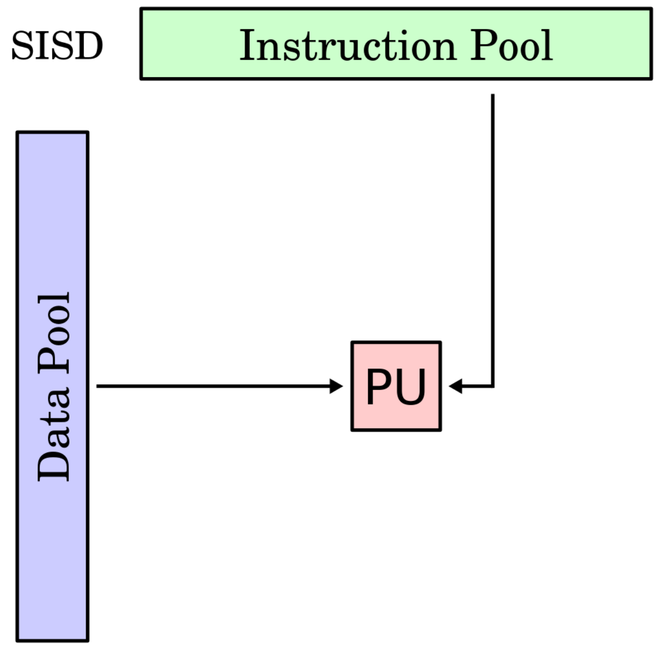
Single-Instruction, Single-Data

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



Single-Instruction, Single-Data

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

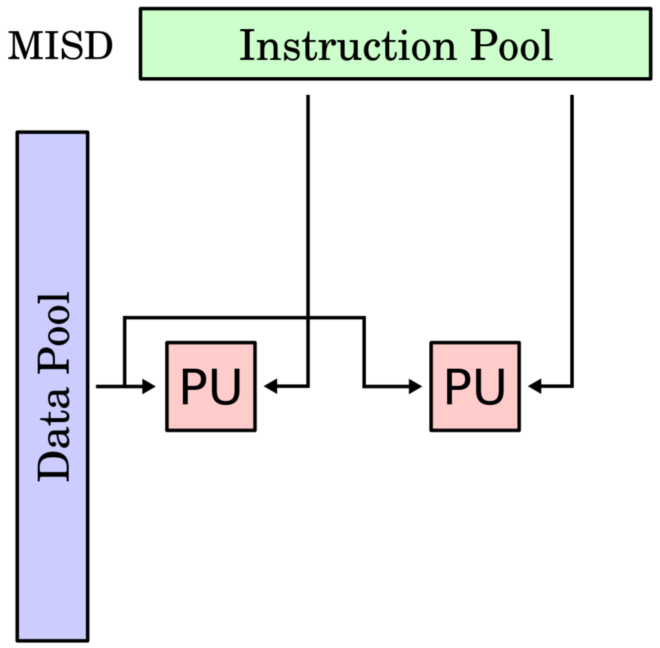


Implementation

- Simple single processor architecture
- No parallelism

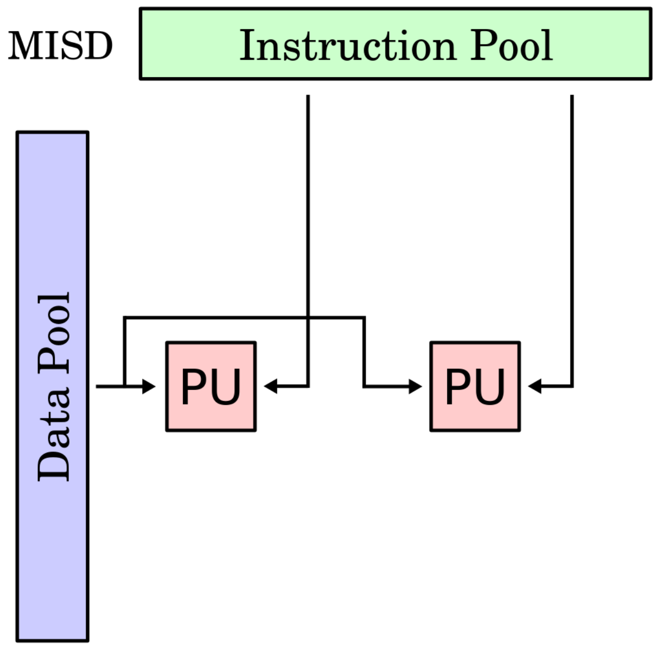
Multiple-Instructions, Single-Data

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



Multiple-Instructions, Single-Data

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

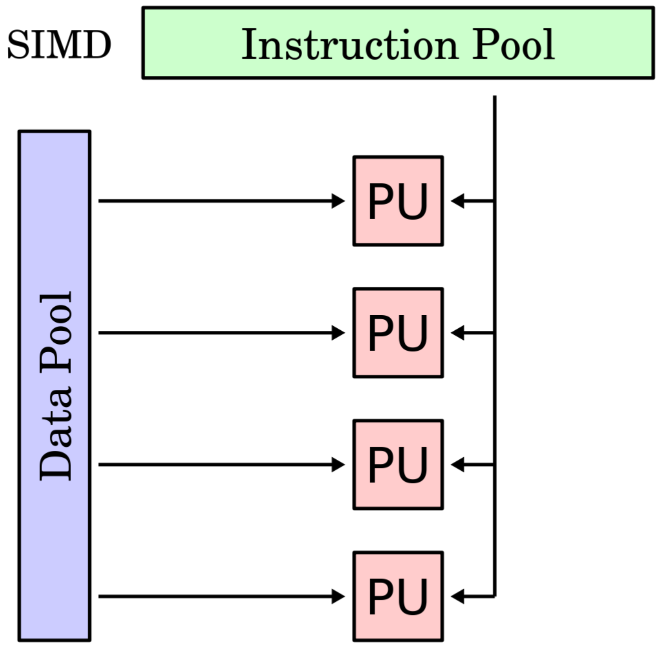


Implementation

- 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

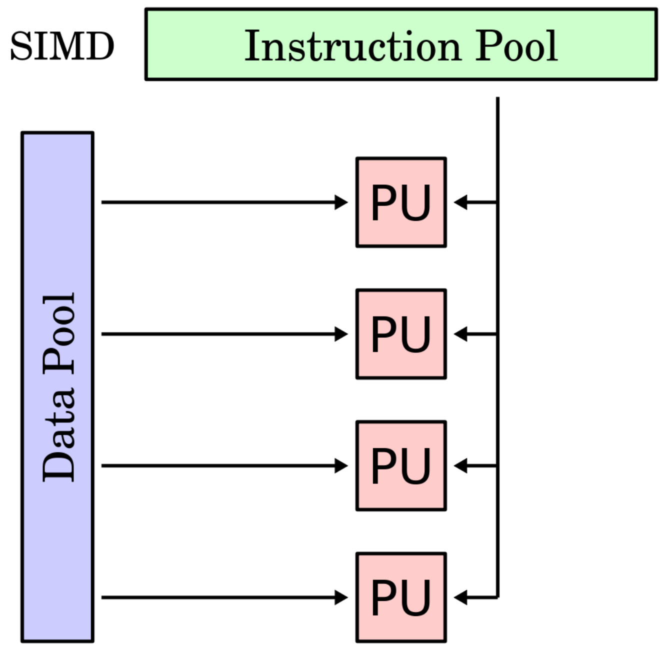
Single-Instruction, Multiple-Data

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



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

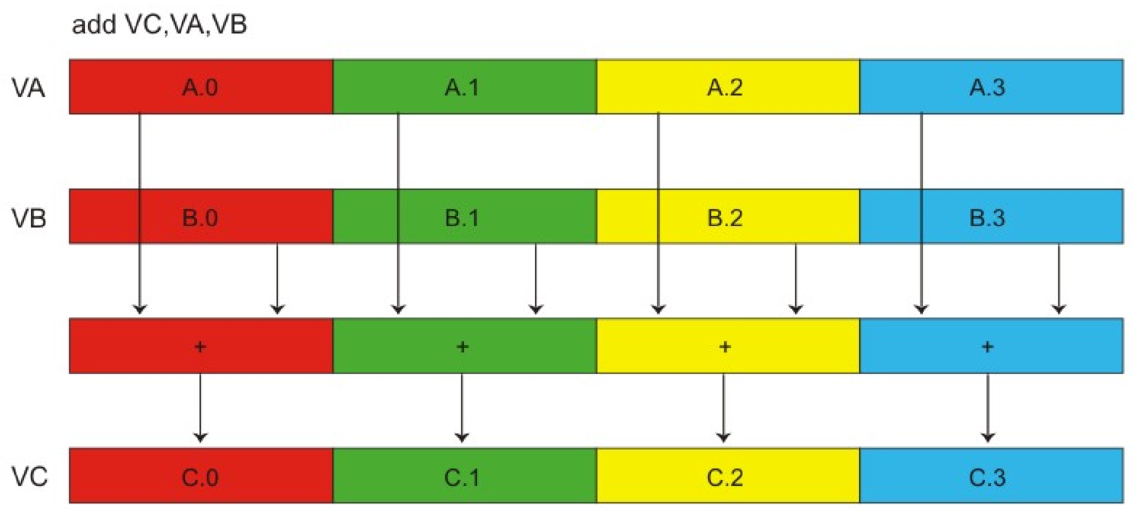


Implementation

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

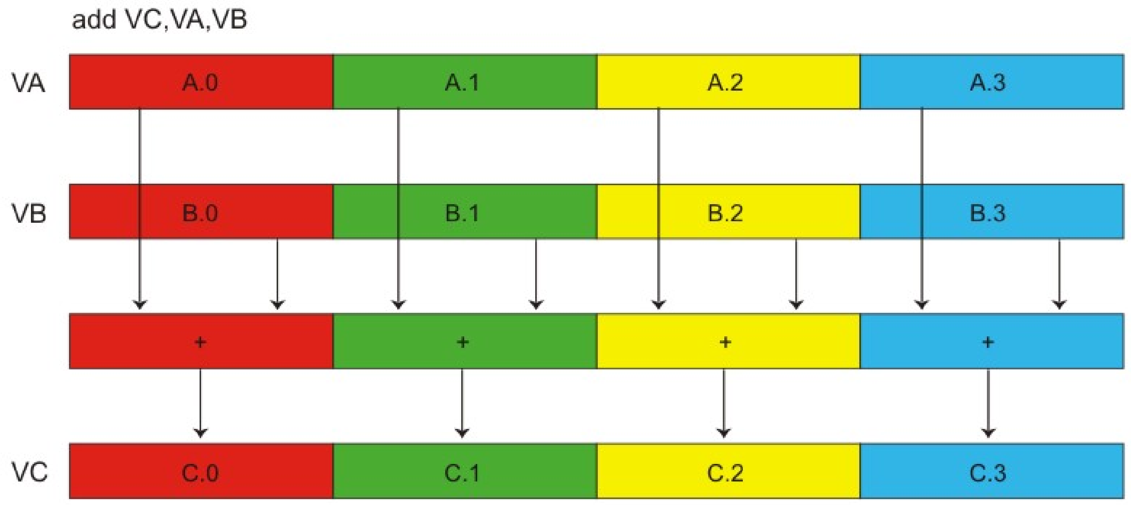
Single-Instruction, Multiple-Data

About vector operations



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

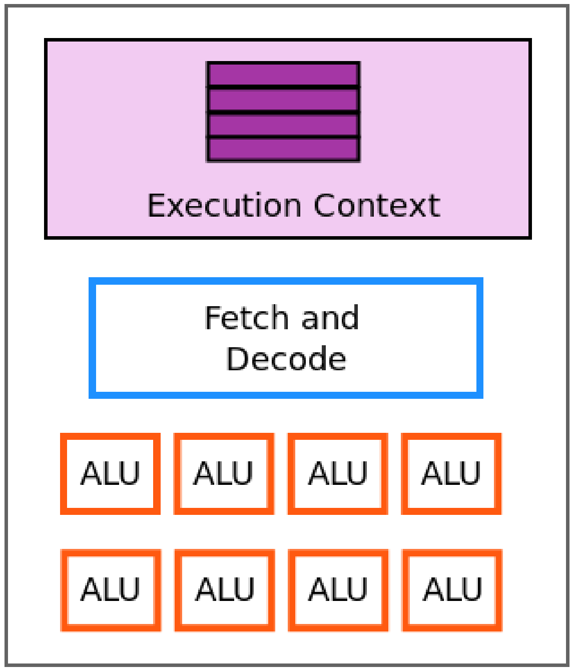
Performance of SIMD

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

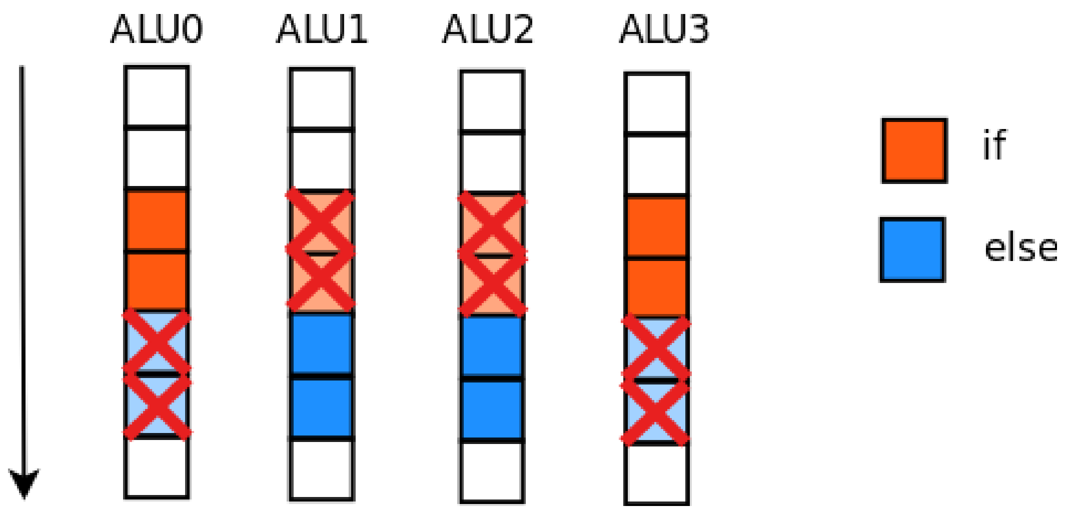
Limitations

- 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

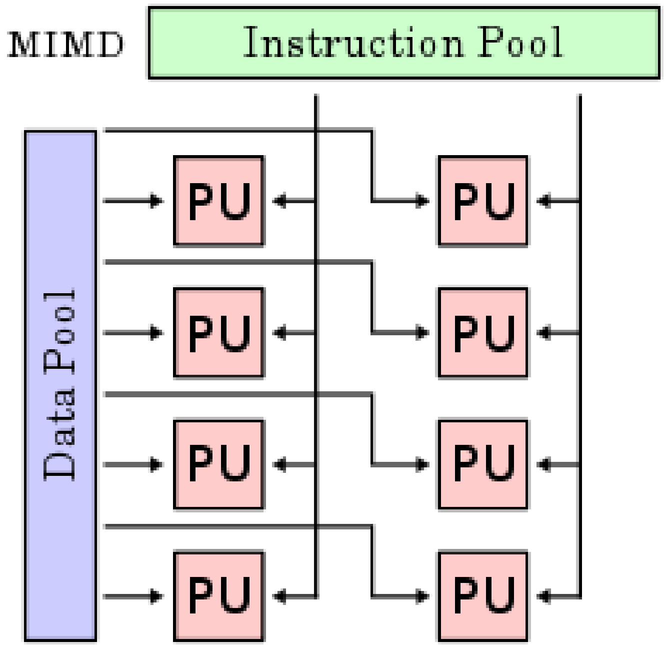
- The compiler can find parallelism automatically by analyzing the dependencies between loops
  - Difficult problem

- The programmer can explicit ask for SIMD parallelism
  - Using compiler intrinsics

- The programmer can give hints through a parallel programming language
  - *Parallel for* loops

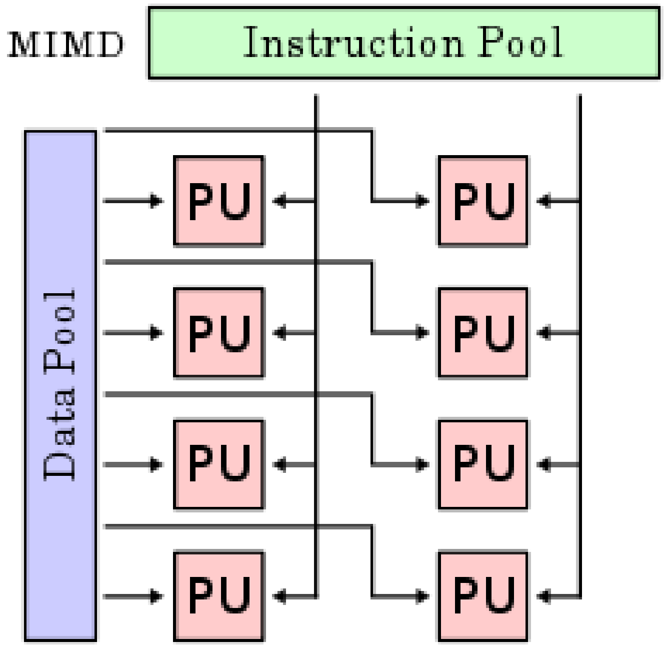
Multiple-Instruction, Multiple-Data

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



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 processor
  - 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.

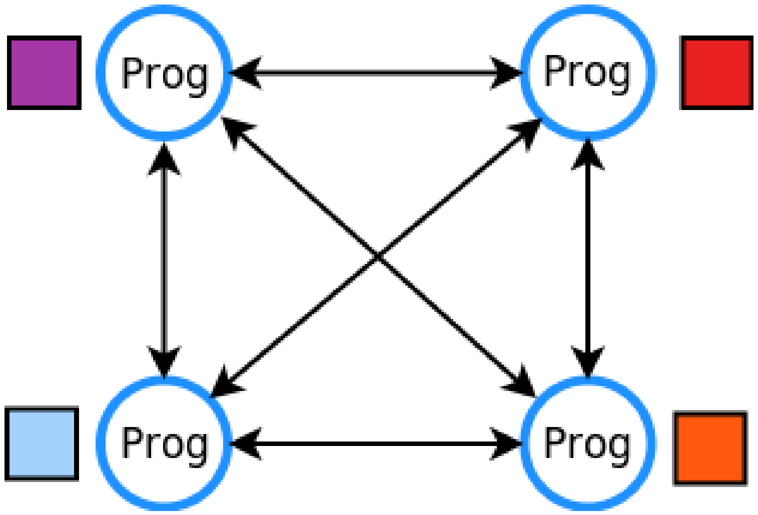
```
*SPMD*: Single program, multiple data

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

Note that here, we are discussing from the **point of view of the programmer**:

- The programmer sees her application as composed of multiple instruction streams:
  - Processes/Threads/Tasks
- Under the hood, the application could (theoretically) be translated into a single stream of SIMD instructions.
  - This comment only applies to the SPMD model
  - Most often, we will execute our programs on MIMD architectures

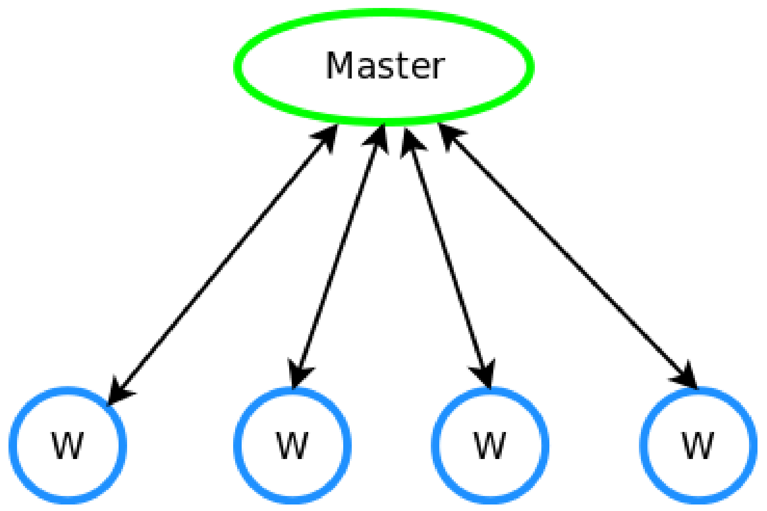
SPMD

- 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
  - Client-server
  - Master-worker (also called master-slave)
- Role of the master
  - Assigning work to worker
  - Coordinating the workers



Programming models

Programming models

3 programming models

What abstraction is offered to the programmer?

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

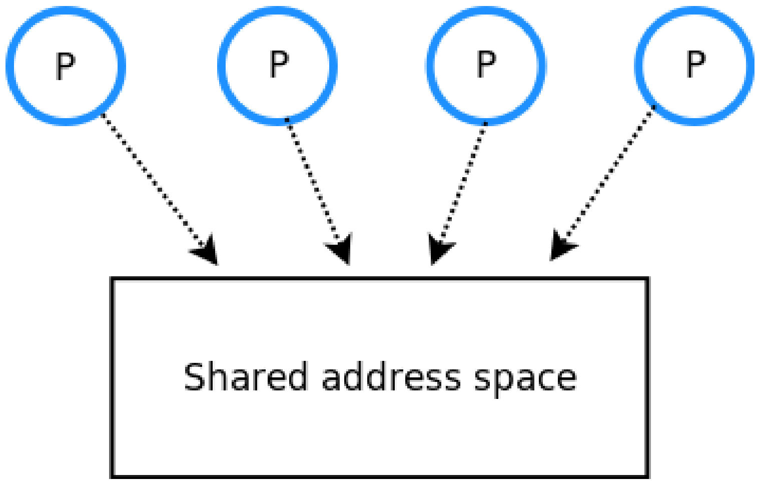
Two communication architecture

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

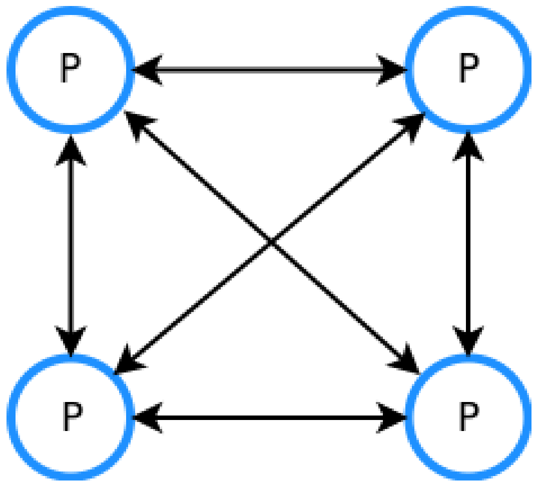
- *Simplest* way of parallelizing a sequential program
    - All the communications are implicit
    - Reaching high performance can be challenging

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

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.

Message passing over 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

Distributed shared memory

- A shared memory abstraction is implemented in software
- A protocol is implemented between the nodes to keep the shared address space consistent
- **Problems**:

Shared memory over message passing

Distributed shared memory

- A shared memory abstraction is implemented in software
- A protocol is implemented between the nodes to keep the shared address space consistent
- **Problems**:
    - No control on the locality of the data
    - Might be very inefficient

Shared memory over message passing

Distributed shared memory

- A shared memory abstraction is implemented in software
- A protocol is implemented between the nodes to keep the shared address space consistent
- **Problems**:
    - No control on the locality of the data
    - Might be very inefficient

Partitioned Global Address Space (PGAS)

- Adds the notion of locality to a distributed shared memory model
    - SHMEM, Global Arrays, Coarray Fortran, Chapel, etc.
    - No wide adoption yet

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 is an input line *
    foreach word in value.split():
        emit(word, 1)



reduce(key, values):        * {word, list nb occurences} *
    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

Conclusion

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

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

About message passing and shared memory

The case of manycore processors

Manycore processors

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

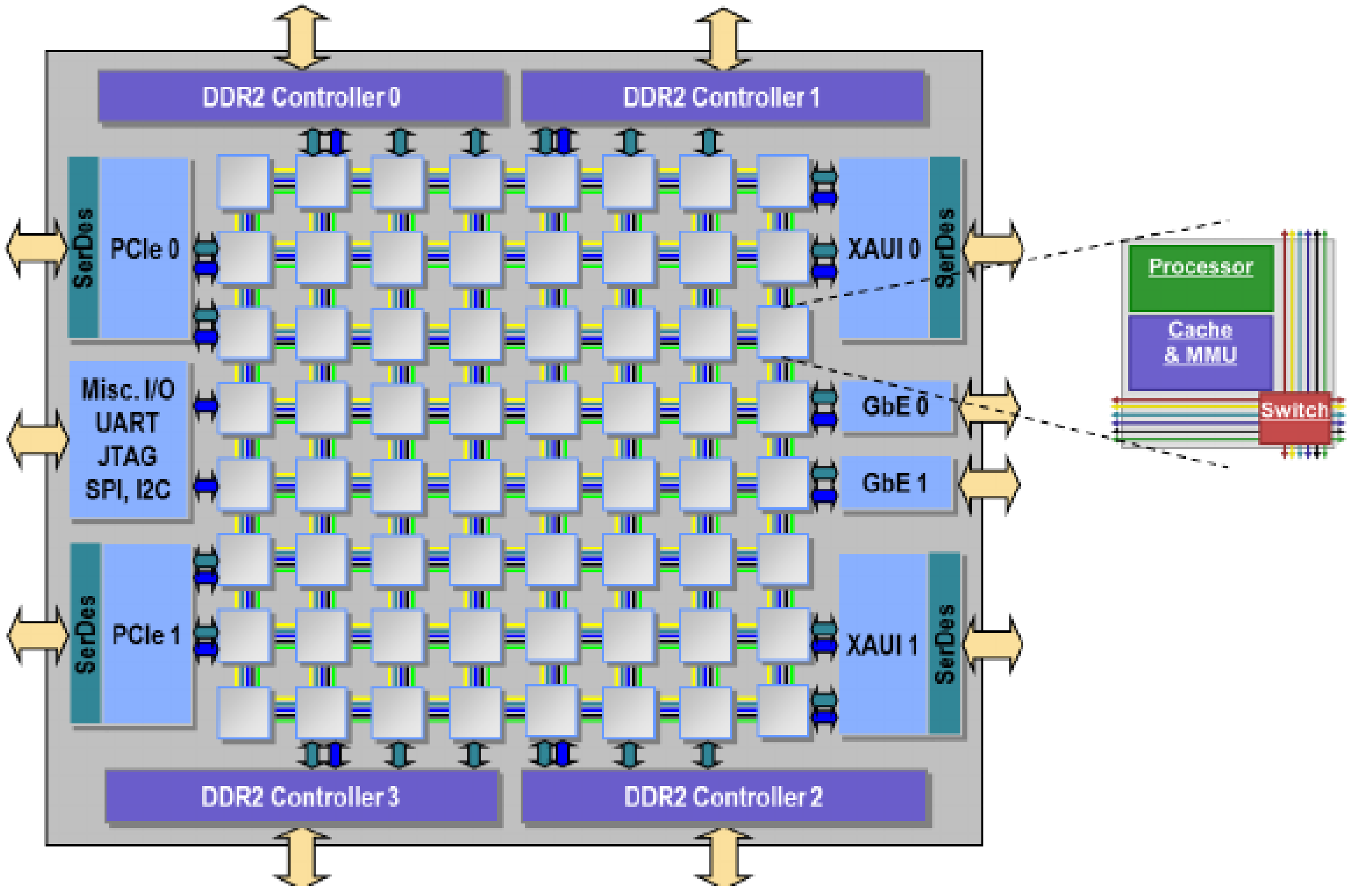
- 72-core tilera processor
- 260-core Sunway processor
- 80-core Kalray processor
- etc.

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

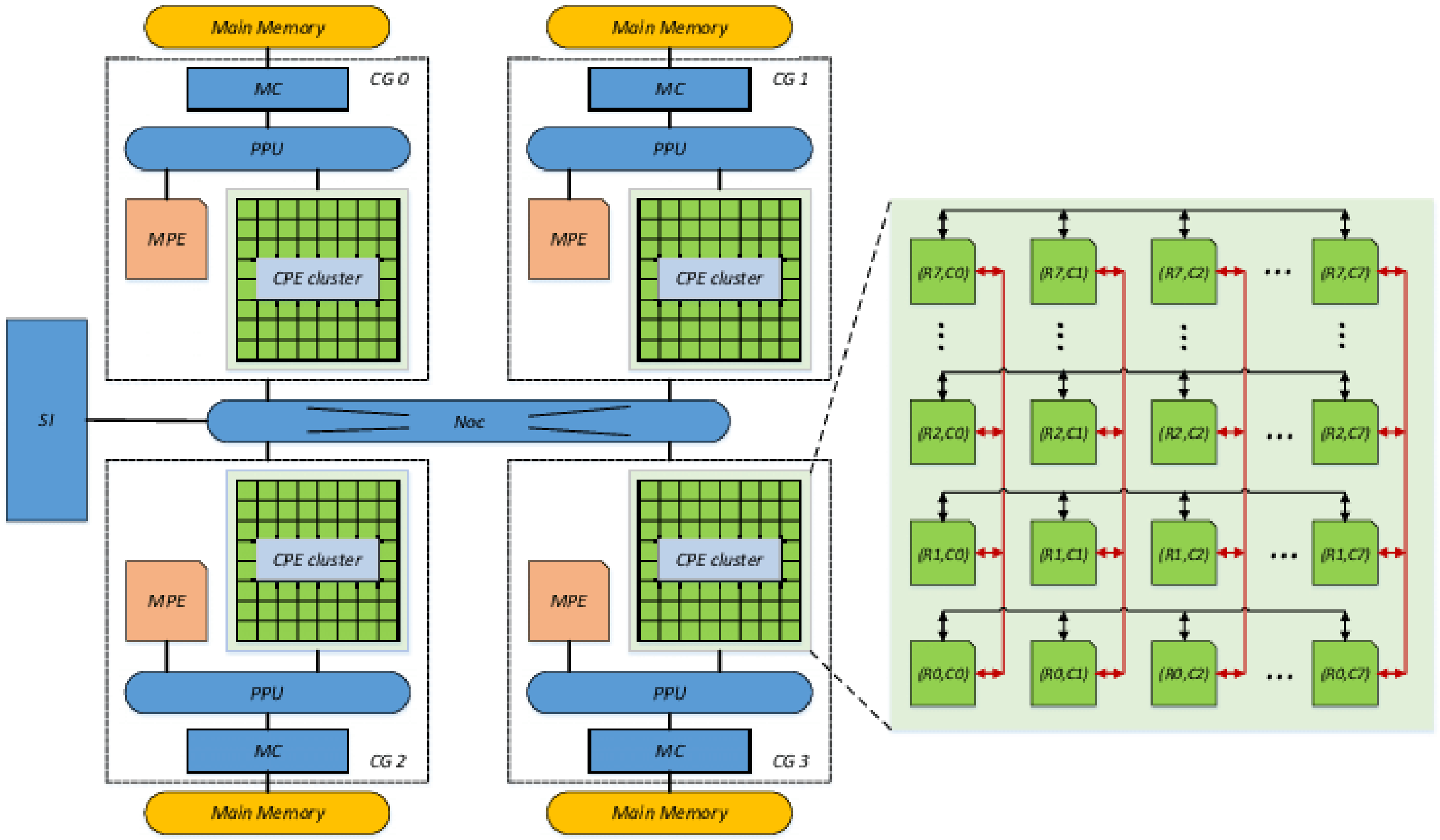
Manycore processor

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.