Module 1

Introduction to parallel programming, Parallel hardware and parallel software

Why we need ever-increasing performance?

- Vast increases in computational power

- Decoding the human genome, ever more accurate medical imaging, astonishingly fast and accurate Web searches, and ever more realistic and responsive computer games

Other example

- *Climate modelling, Protein folding, Drug discovery, Energy research*.

Why we're building parallel systems?

- Much of the tremendous increase in single-processor performance was driven by the ever-increasing density of transistors.
- As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased. However, as the speed of transistors increases, their power consumption also increases.
- Integrated circuits are called **multicore** processors, and **core** has become synonymous with central processing unit, or CPU. In this setting a conventional processor with one CPU is often called a **single-core** system.

Why we need to write parallel programs

- Rewrite our serial programs so that they're *parallel*, so that they can make use of multiple cores, or write translation programs, that is, programs that will automatically convert serial programs into parallel programs

- As an example, suppose that we need to compute *n* values and add them together.

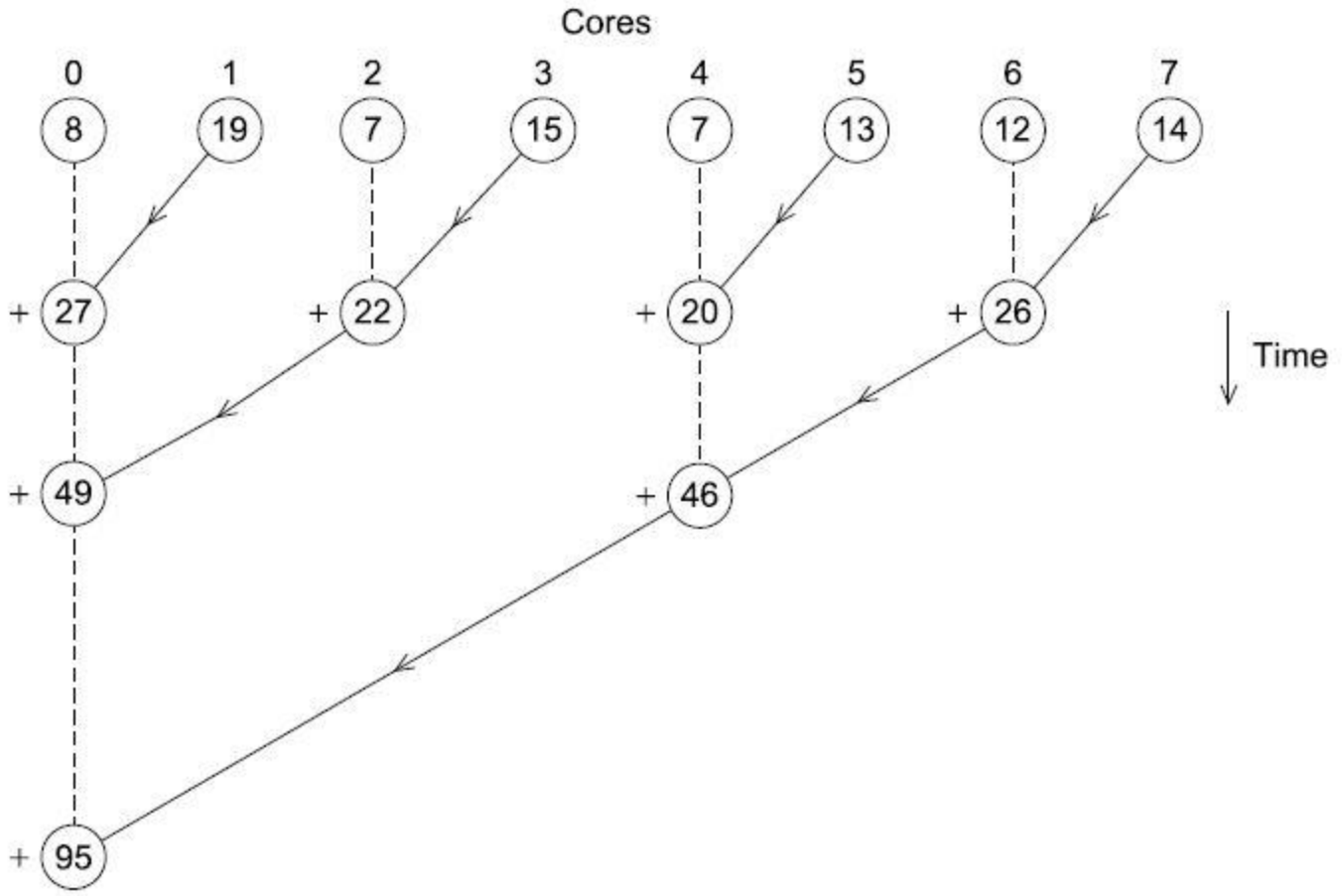
We know that this can be done with the following serial code:

```
sum = 0;
for ( i = 0; i < n ; i ++) {
x = Compute_next_value ( . . . ) ;
sum += x ;
}
```

Single core …

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for ( my_i = my_first_i ; my_i < my_last_i ; my_i ++) {
my_x = Compute_next_value ( . . . ) ;
my_sum += my_x ;
}
```

Multiple cores forming a global sum.



```
f ( I'm the master core) {
sum = my_sum ;
for each core other than myself {
receive value from core ;
sum += value ;
}
} else {
send my_sum to the master ;
}
```

How do we do parallelism?

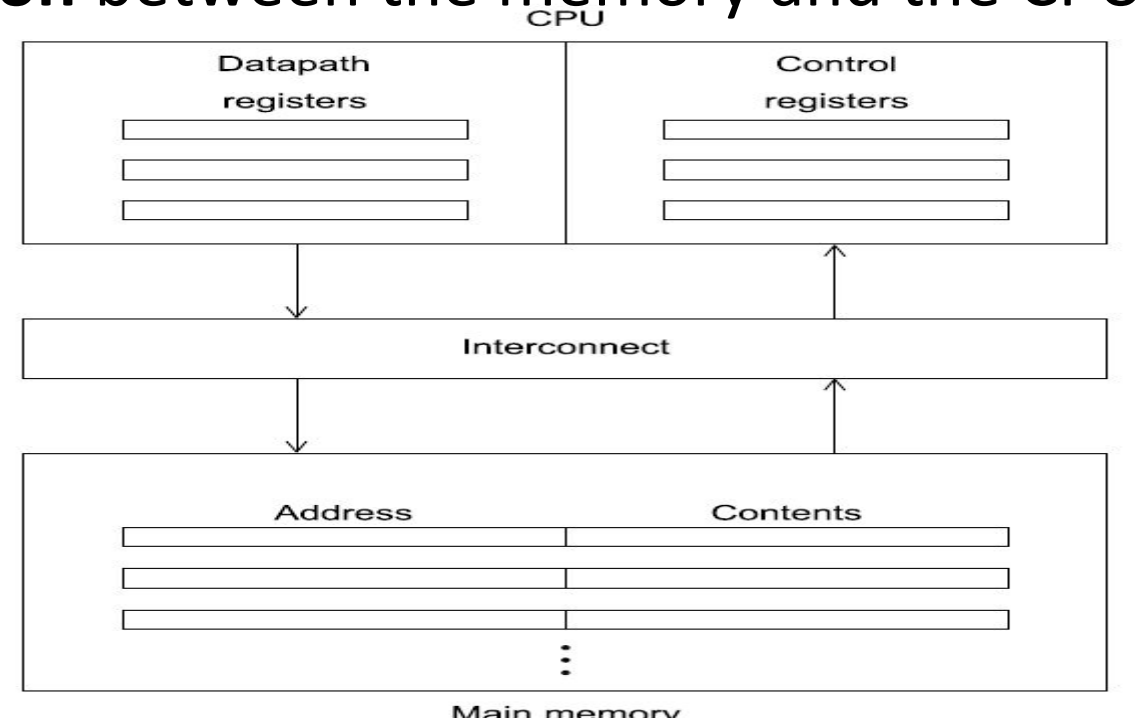
- There are two widely used approaches: **task-parallelism** and **data-parallelism**.

What we do in course?

- Our purpose is to learn the basics of programming parallel computers using the C language and four different **APIs** or **application program interfaces**: the **Message-Passing Interface** or **MPI**, **POSIX threads** or **Pthreads**, **OpenMP**, and **CUDA**.

The von Neumann architecture

- The "classical" **von Neumann architecture** consists of **main memory**, a **central processing unit** (CPU) or **processor** or **core**, and an **interconnection** between the memory and the CPU.



Important Terminology

- Process , Thread
- Cache, cache mappings

Classifications of parallel computers

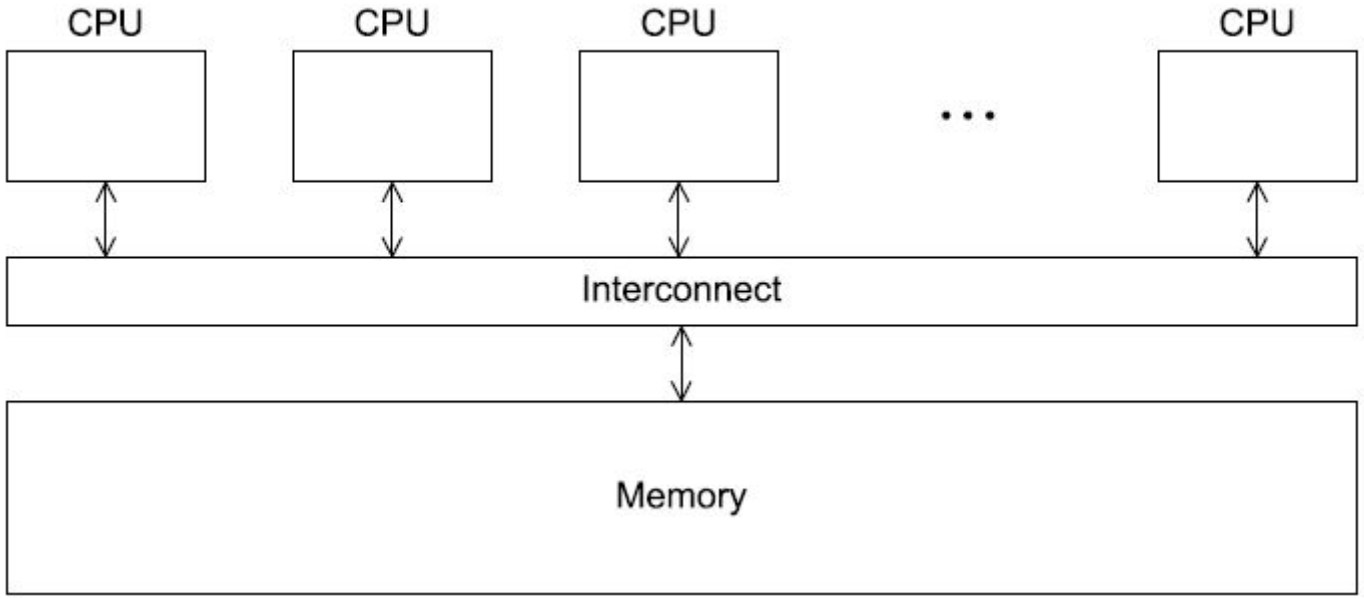
- The number of instruction streams and the number of data streams (or datapaths)
- A classical von Neumann system is therefore a **single instruction stream, single data stream**, or SISD
- The parallel systems in the classification can always manage multiple data streams, and we differentiate between systems that support only
- a single instruction stream (SIMD)
- multiple instruction streams (MIMD)

Memory Systems

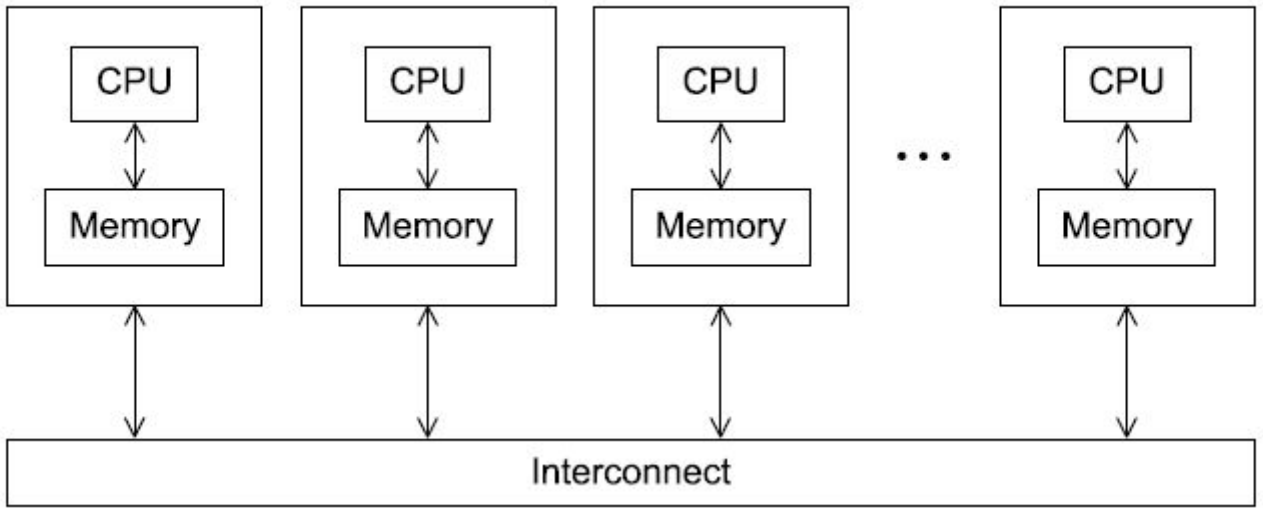
- In **shared memory** systems, the cores can share access to memory locations, and the cores coordinate their work by modifying shared memory locations.

- In **distributed memory** systems, each core has its own, private memory, and the cores coordinate their work by communicating across a network.

Shared-memory systems



A distributed-memory system



SIMD systems

- **Single instruction, multiple data**, or SIMD, systems are parallel systems.
- SIMD systems operate on multiple data streams by applying the same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple datapaths.

Vector addition

- An instruction is broadcast from the control unit to the datapaths, and each Datapath either applies the instruction to the current data item, or it is idle.
- Example:

two arrays x and y, each with *n* elements are added

```
for ( i = 0; i < n ; i ++)
    x [ i] += y [ i ] ;
```

Data-parallelism

- SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data.
- Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called **data-parallelism**.

Vector Processors

- Operates on Vectors of Data or arrays of data
- Vector Registers

These are registers capable of storing a vector of operands and operating simultaneously on their contents.

- *Vectorized and pipelined functional units*
- to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors Thus vector operations are SIMD

Vector Processors

- *Vector instructions.* These are instructions that operate on vectors rather than scalars.
- *Interleaved memory.* The memory system consists of multiple “banks” that can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed much sooner.
- In *strided memory access*, the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on would be strided access with a stride of four

MIMD systems

- **Multiple instruction, multiple data**, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams.
- MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own datapath
- MIMD systems are usually **asynchronous**
- MIMD systems, there is no global clock

Two typical MIMD system

- **Shared-memory system**

collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location

- **Distributed-memory system**

each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. So in distributed-memory systems, the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor.

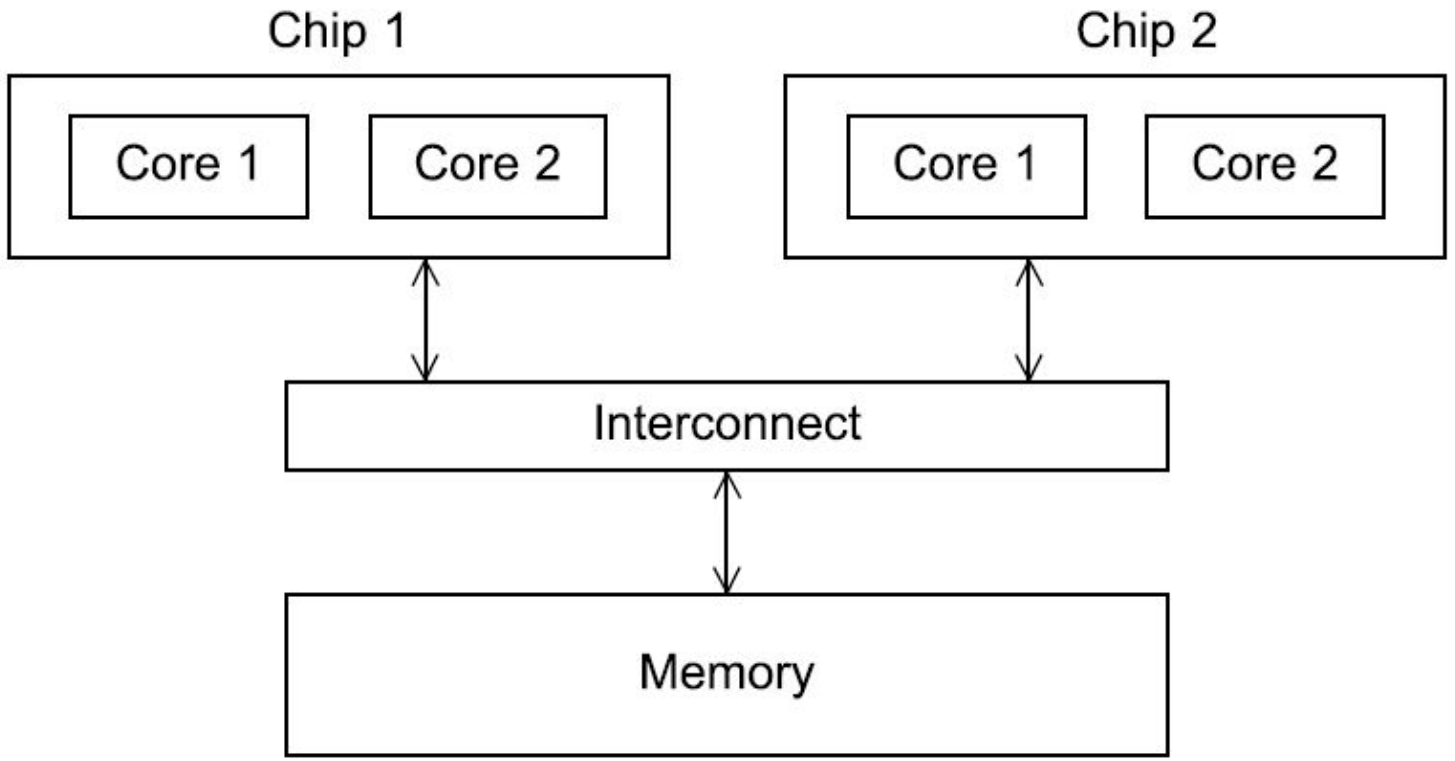
Shared Memory System

- Two Types

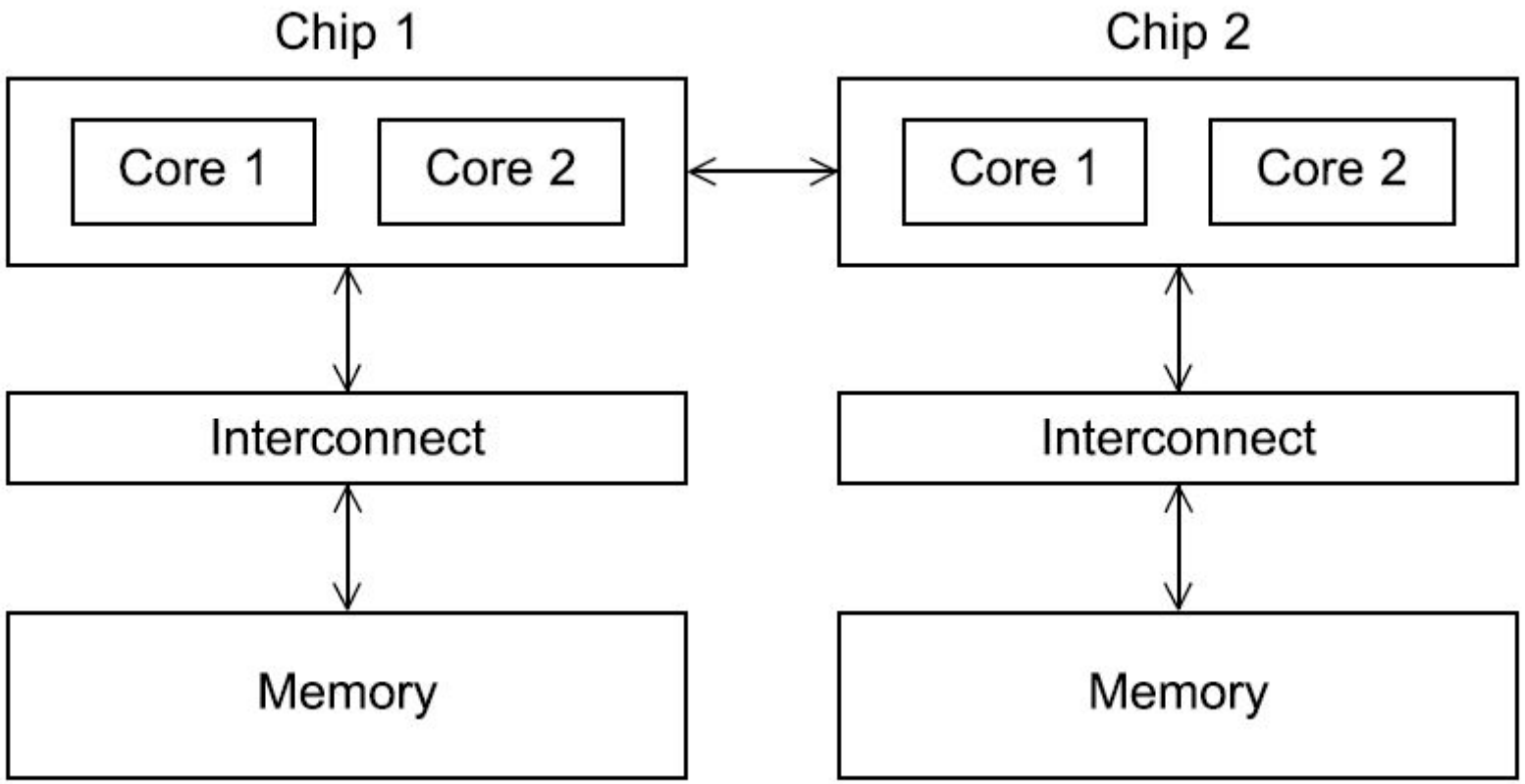
1.UMA - the time to access all the memory locations will be the same for all the cores,

2. NUMA- A memory location to which a core is directly connected, can be accessed more quickly than a memory location that must be accessed through another chip.

A UMA multicore system.



A NUMA multicore system.



Distributed-memory systems

- The most widely available distributed-memory systems are called **clusters**.
- They are composed of a collection of commodity systems—for example, PCs—connected by a commodity interconnection network—for example, Ethernet.
- The **grid** provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system.

Interconnection networks

- The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems
- the processors and memory have virtually unlimited performance
- But we have a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program
- Examples of Interconnection networks: Bus

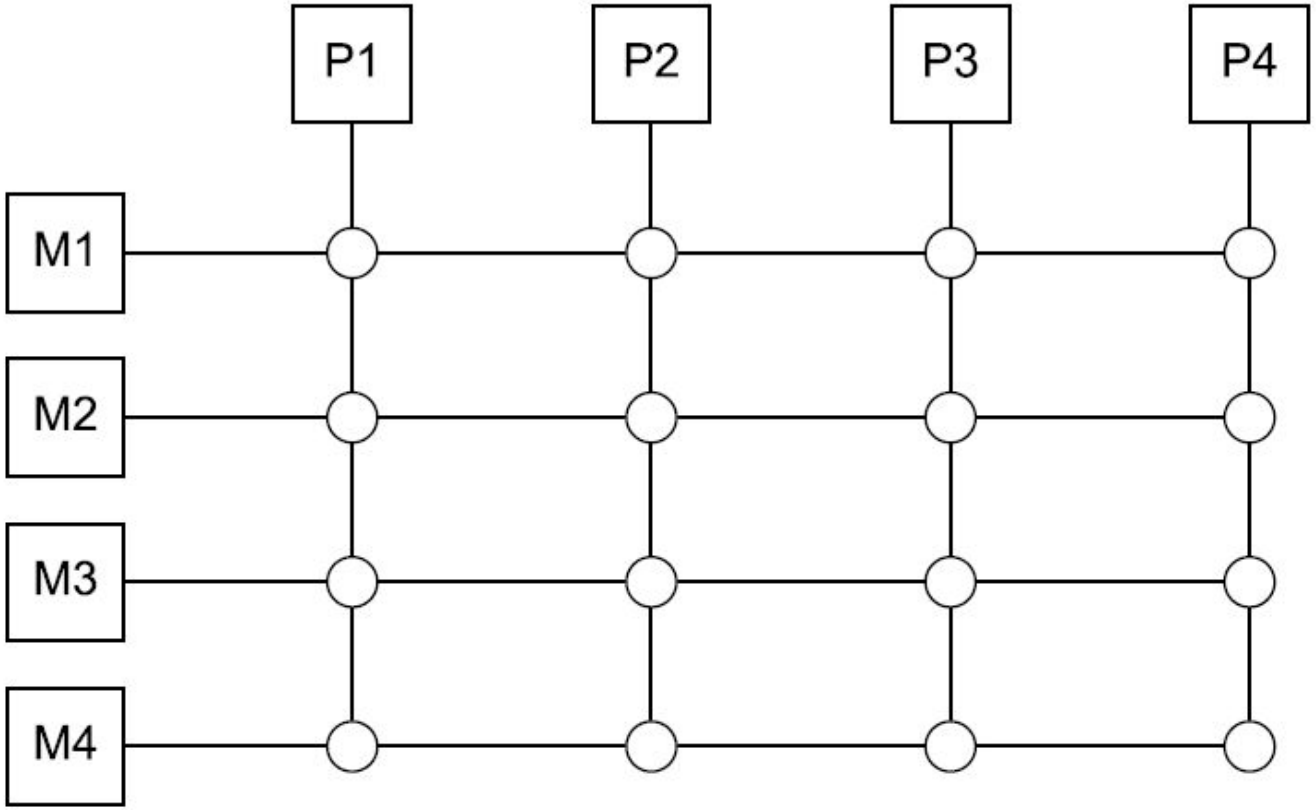
Shared-memory interconnects

- shared memory systems to use a **bus** to connect processors and memory.
- Buses have the virtue of low cost and flexibility
- as the size of shared-memory systems has increased, buses are being replaced by *switched* interconnects.
- **switched** interconnects use switches to control the routing of data among the connected devices. A **crossbar** is a relatively simple and powerful switched interconnect

- Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high.

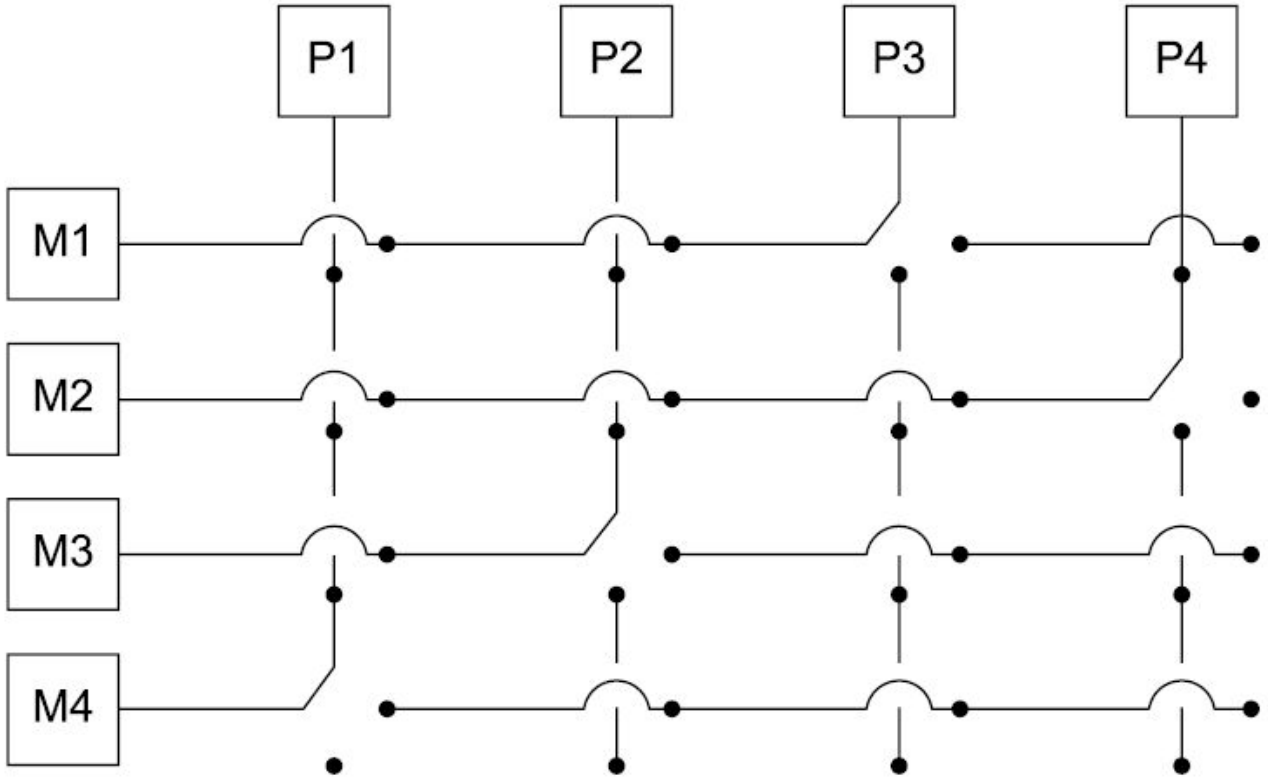
- A small bus-based system will be much less expensive than a crossbar-based system of the same size.

A crossbar switch connecting four processors (*Pi*) and four memory modules (*Mj*)



Simultaneous memory accesses by the processors

(b)



Distributed-memory interconnects

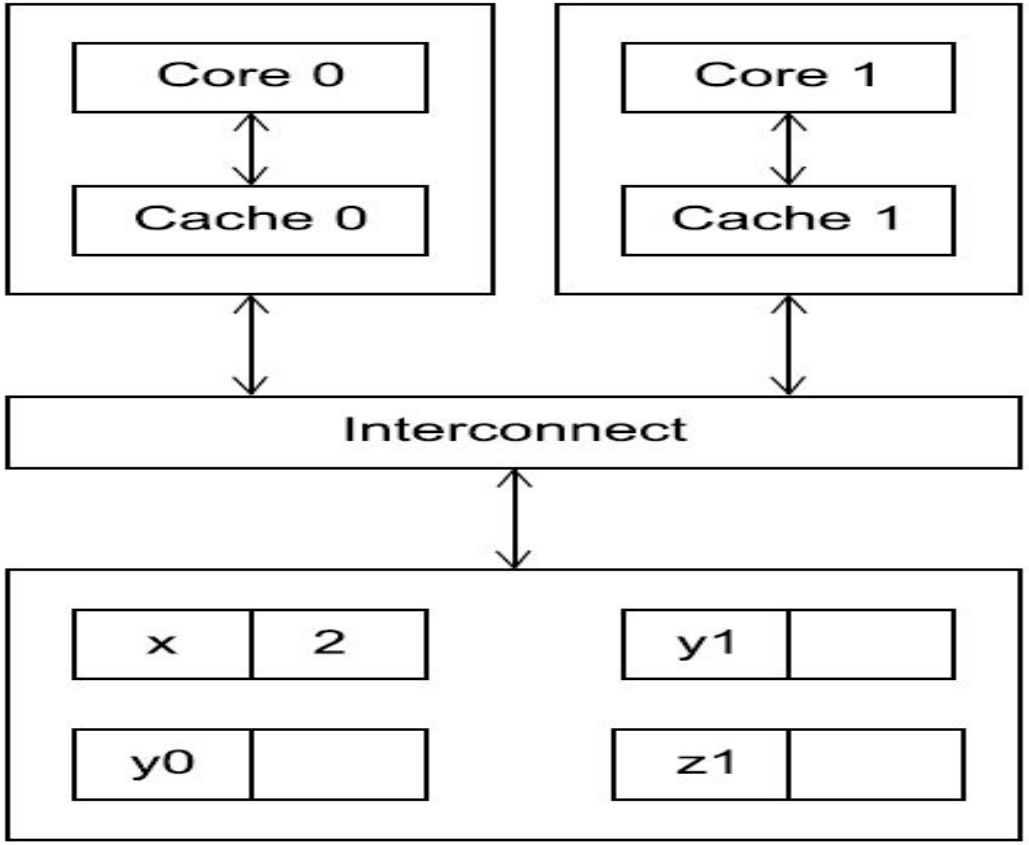
1. Direct interconnects

In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other.

2. Indirect interconnects

provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor.

Cache coherence



Cache coherence

- Cache coherence is a mechanism that ensures data consistency across multiple caches in a shared memory system, such as a multi-core processor.
- It addresses the issue where different processors might have inconsistent copies of the same data in their caches, potentially leading to incorrect program execution.

Directory-based

- In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches
- The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache
- When an entry is changed, the directory either updates or invalidates the other caches with that entry.

Cache updating policy

Write Through Policy
Write Back Policy

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

Snooping

Is a process where the individual caches monitor address lines for accesses to memory locations that they have cached.

It is called a write invalidate protocol.

When a write operation is observed to a location that a cache has a copy of and the cache controller invalidates its own copy of the snooped memory location.

Coordinating the processes/threads

1. Divide the work among the processes/threads in such a way that

a. each process/thread gets roughly the same amount of work, and

b. the amount of communication required is minimized.

2. Arrange for the processes/threads to synchronize.

3. Arrange for communication among the processes/threads.

Coordinating the processes/threads

- **double** x [n] , y [n] ;
- **for** (**i n t** i = 0; i < n ; i ++)

x [i] += y [i] ;

- To parallelize this, we only need to assign elements of the arrays to the processes/ threads. For example, if we have *p* processes/threads, we might make process/ thread 0 responsible for elements 0*, . . . , n/p* − 1, process/thread 1 would be responsible for elements *n/p, . . . ,* 2*n/p* − 1, and so on.

Shared Memory Programs

- Shared-memory programs, variables can be **shared** or **private**.
- Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread.
- Communication among the threads is usually done through shared variables, so communication is implicit rather than explicit.

Dynamic thread

- Shared-memory programs use **dynamic threads**.
- There is often a master thread and at any given instant a collection of worker threads. The master thread typically waits for work requests
- For example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread.
- Efficient use of system resources, since the resources required by a thread are only being used while the thread is actually running.

Static Thread

- All of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed.
- After the threads join the master thread, the master thread may do some cleanup (e.g., free memory), and then it also terminates. In terms of resource usage, this may be less efficient.
- If a thread is idle, its resources (e.g., stack, program counter, and so on) can't be freed

Nondeterminism

- MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism.**

- Computation is nondeterministic if a given input can result in different outputs.

- If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run.

Thread Safety

- A function such as strtok is not **thread safe**. This means that if it is used in a multithreaded program, there may be errors or unexpected results.
- As in the multi-threaded context where a program executes several threads simultaneously in a shared address space and each of those threads has access to every other thread's memory,
- thread-safe functions need to ensure that all those threads behave properly and fulfill their design specifications without unintended interaction

Distributed Memory

- In distributed-memory programs, the cores can directly access only their own, private memories.
- There are several APIs that are used. However, by far the most widely used is message-passing.

- A message-passing API provides (at a minimum) a send and a receive function. Processes typically identify each other by ranks in the range 0, 1, . . . , $p - 1$, where p is the number of processes. So, for example, process 1 might send a message to process 0

Message Passing API pseudocode

```
char message [ 1 0 0 ] ;
my_rank = Get_rank ( ) ;
i f ( my_rank == 1) {
sprintf ( message , " Greetings from process 1" ) ;
Send ( message , MSG_CHAR , 100 , 0 ) ;
} e l s e i f ( my_rank == 0) {
Receive ( message , MSG_CHAR , 100 , 1 ) ;
printf ( " Process 0 > Received : %s\n" , message ) ;
}
```

One-sided communication

- In **one-sided communication**, or **remote memory access**, a single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process.
- Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes.
- It can also reduce overhead by eliminating the overhead of one of the functioncalls (send or receive).