Contents

2.1 von Neumann Model 2

2.3 Parallel hardware 12

2.4 Parallel Software 18

2.5 INPUT AND OUTPUT 22

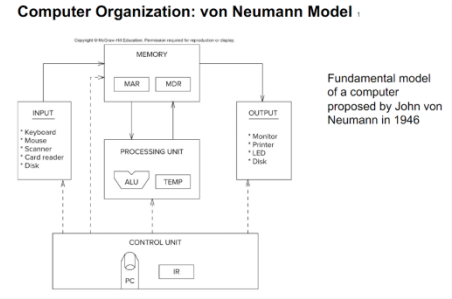
2.6 PERFORMANCE 23

2.6.1 Speedup and efficiency 23

Parallel Hardware and Parallel Software

2.1 von Neumann Model

What the von Neumann Model



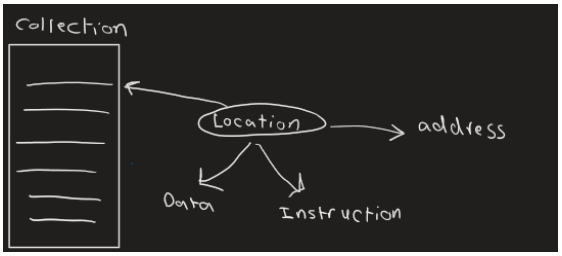
Memory:

Store the data and Instructions during program Execution

Main memory consists of a collection of location

Each of which can store both instructions and data

Every location consists of an address



The Central processing unit CPU

Is divided into:

Processing unit: transforms data, logic circuits, temporary storage

Control Unit: orchestrates fetching of instructions from memory and execution of instructions

It has a special register called the “PC” program counter

Register: Data in the CPU and information about the state of an executing program are stored in a special, very fast storge

What is a program counter?

It stores the address of the next instruction to be executed

Input:

Receives data from external environment

Output:

Send data to external environment

Bus: instructions and data transferred between the CPU and Memory via **Interconnect**

Fetched or read from memory:

Transferred data or instructions from memory to CPU

Write to memory or stored:
Transferred data or instructions from CPU to memory

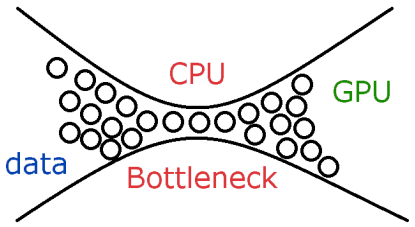
What is the von Neumann architecture problem?

The von Neumann architecture has a problem which is transferring data or instructions from main memory into CPU is much less than it should be

The CPU needs to access more data and instructions and much faster from the main memory to have a better performance

What is bottleneck?

The separation of memory and CPU



2.1.2 Processes. Multitasking and threads:

What is OS (Operating System)?

Major piece of software whose purpose is to manage hardware and software resources on a computer

It determines which programs can run and when they can run
It also controls the allocation of memory to running programs
Access to peripheral devices such as hard disks and network interface cards

Run a program step by step?

OS create a process: an instance of a computer program that is being executed

A process consists of several entities

Executable machine language program

Block of memory, which will include the executable code

Call stack: that keep track of active functions

Descriptors of resources that the OS has allocated to the process, for example File Descriptors

Information about the state of the process, Such as Run, waiting and ready

Multitasking: time Slice:

A small interval of time where a single process runs
Most modern OS are multitasking, because they change a running process many times a minute to allow other processes to run even though changing processes tasks long time, but this allows running many processes at the same time

What are a thread?

Allows dividing a program into other smaller programs called threads

Switching between threads is much faster than switching between processes

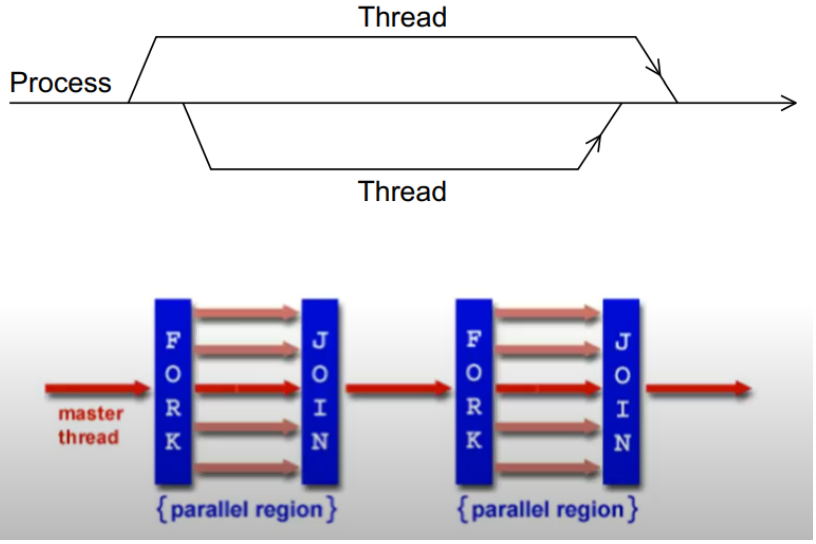
When a thread is block waiting for some resources, other thread can be run

Threads of a process share the same resources as a process does

To make threads execute independently from each other they need the following two most important exceptions

1. A record of their own program counter
2. They'll need their own call stack

When a thread is started, it fork off the process, when a thread terminates, it join the process



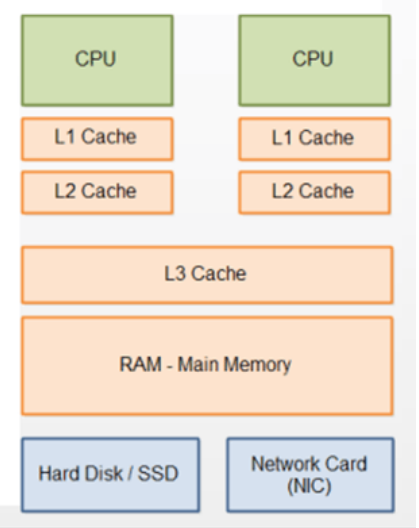
What is Caching?

It is providing two solutions to solve the Von Neumann architecture problem

1. Wider interconnect which allows transporting more data and instructions in a single memory access
2. Rather storing all block of data and instructions in main memory, they can be stored in a special memory closer to the register in the CPU

Another definition:

computer memory with short access time used for the storage of frequently or recently used instruction or data



Cache/ CPU Cache
Is a memory location that can either be located on the same chip as a CPU or it can be located on another chip which makes them be accessed faster by the CPU

Once a program has accessed one memory location, it will continue accessing other memory locations that are physically nearby
Caches are divided into levels and most caches have at least 2-3 levels
The first level is the smallest and the fastest
The level 2 and 3 are larger and slower

Some caches store information in slower and faster levels "Level 1 and 3" for example which means the same information will be stored in two levels of cache.

Other multilevel caches do not store information in more than one level
They instead store information in level 1 and main memory

When the CPU needs to get some data, it goes the whole way from first level in cache, level 1, level 2 and so on till main memory

If the data is available in the cache, it is called "cache hit"
If not, available it is called "cache mis"

Cache hit:
requested value found in cache
Cache miss:
value not found; request passed up to next cache line
Cache line:
cache is spilt into lines each line is multiple bytes (64-256 bytes), a memory request will obtain multiple bytes

So, there might be a cache miss in L1 and cache hit in L2. Data or instructions always get stored down-up which means from slower memory to faster memory.

For example, if the required data are in L2, these data will be stored to L1 then the CPU can get access to it. Otherwise, if the data or instructions are in the main memory, these data or instructions will be stored in the cache first before the CPU can access them.

The CPU will stop executing a program while it is waiting for the required data to be fetched.

When the CPU writes data to a cache, the value in the cache and the value in main memory are different or inconsistent. There are two basic approaches to dealing with the inconsistency.

In write-through caches, the line is written to main memory when it is written to the cache. In write-back caches, the data isn't written immediately.

Rather, the updated data in the cache is marked dirty, and when the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Locality

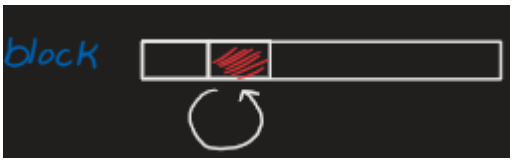
Locality is when for example, the first element of the following array z[0] gets access to a memory location, the next element which is z[1] will be located in the next / nearby memory location. To apply the principle of locality the system uses a wider interconnect to access more data and instructions.

So, Locality: Program tend to use data and instructions with address near or equal to those they have used recently

We have two types of locality:

Temporal locality

Recently referenced items are likely to be referenced again in the near future



Spatial Locality

Items with nearby addresses tend to be referenced close together in time

Example

```
sum = 0;
  for (int i = 0; i < n; i++)
  {
      sum += a[i];
      return sum;
  }
```

Data:

Temporal: `sum` referenced in each iteration

Spatial: `a[i]` assessed in stride -1 pattern

Instructions:

Temporal: cycle through loop repeatedly (for loop)

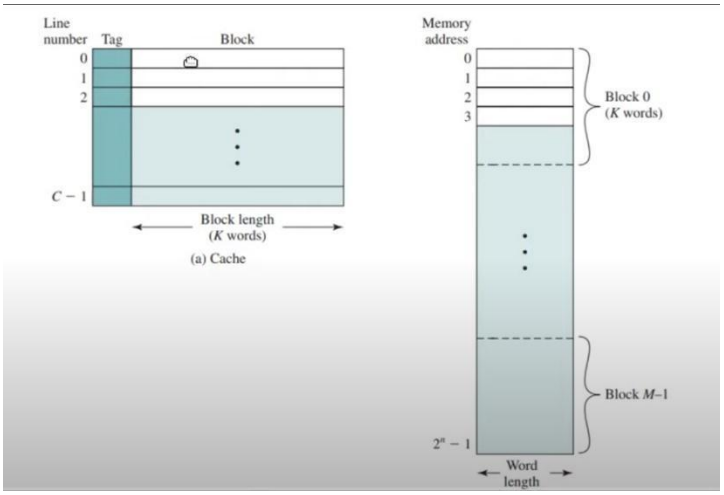
Spatial: reference instructions in sequence

What is cache Problem?

Deciding which data and instructions should be stored in the cache. Data and instructions that are stored in the cache are the most used data and instructions.

Cache block/cache line

Cache blocks are blocks that contain a couple of data and instructions. A cache line stored 8 – 16 times as much as a single memory location. So from the previous example with array when we go to the first element to store some data in z[0], the first 16 elements will be read and to the cache. Then when we want to do something with for example element z[1], all the elements up till 15 are already in the cache.



Cache mapping

There is another problem in caching which to decide where in the cache, cache lines should be placed if they were fetched from main memory

There are three ways to deal with this issue:

Fully associative cache

A cache line can be placed at any location in the cache

Direct mapped cache

Each cache line has a unique location in the cache

n-way set associative
A cache line can be placed in one of n different locations

Example From the book:

Suppose our main memory consists of 16 lines with indexes 0-15

Our cache consists of 4 lines with indexes 0-3

Fully associative cache

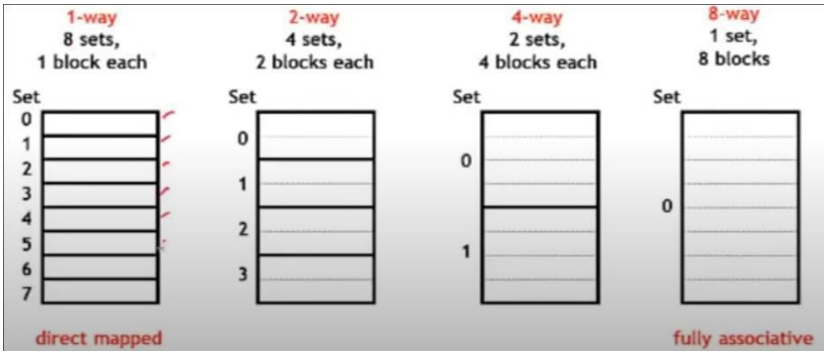
Cache line 0 - 15 from main memory can be placed in any one cache lines 0 – 3. Why? Because cache lines in a fully associative cache will be placed randomly in any location in the cache.

Direct mapped cache

Cache lines from main memory will be placed in cache lines based on their index. That means that we have to do some calculation to find out each and every line from main memory where it has to be placed in caches. To do the calculation, we divide the index of each cache line from main memory with 4. The remaining result will be the number of the index where a cache line from main memory will be placed in cache. So, for example, cache lines from main memory with indexes 0. 4, 8 and 12 will be placed index number 0 in the cache. Why, because
0/4 = 0 and we have 0 left
4/4 = 1 and we have 0 left
8/4 = 2 and we have 0 left and so on.
And cache lines form main memory with index 1, 5, 9 and 13 will be placed in cache index 1 and so on.

n-way set associative
For example, we have a 2-way set associate which means we have to make to sets where each set contains 2 indexes. Set 1 will be called set 0, and set 2 will be called set 1. Each of set 0 and 1 has 2 indexes. Set 0 has indexes 0 and 1. Set 1 has indexes 2 and 3. Each element from cache lines from main memory can be placed in any index of one the above sets. To determine which set, each element from main memory should be stored in, we take the module 2 of each index.
For example
Index 0 / 2 = 0 and we have 0 left which means index 0 of cache lines from main memory has to be placed in set 0 at any of index 0 and 1.
Index 2/2 = 1 and 0 left, again in set 0
Index 3/2 = 1 and 1 left, and it should be placed in set 1 at any index of 2 or 3 and so on.

N-way set associative



b

How do I know where data goes?

Example

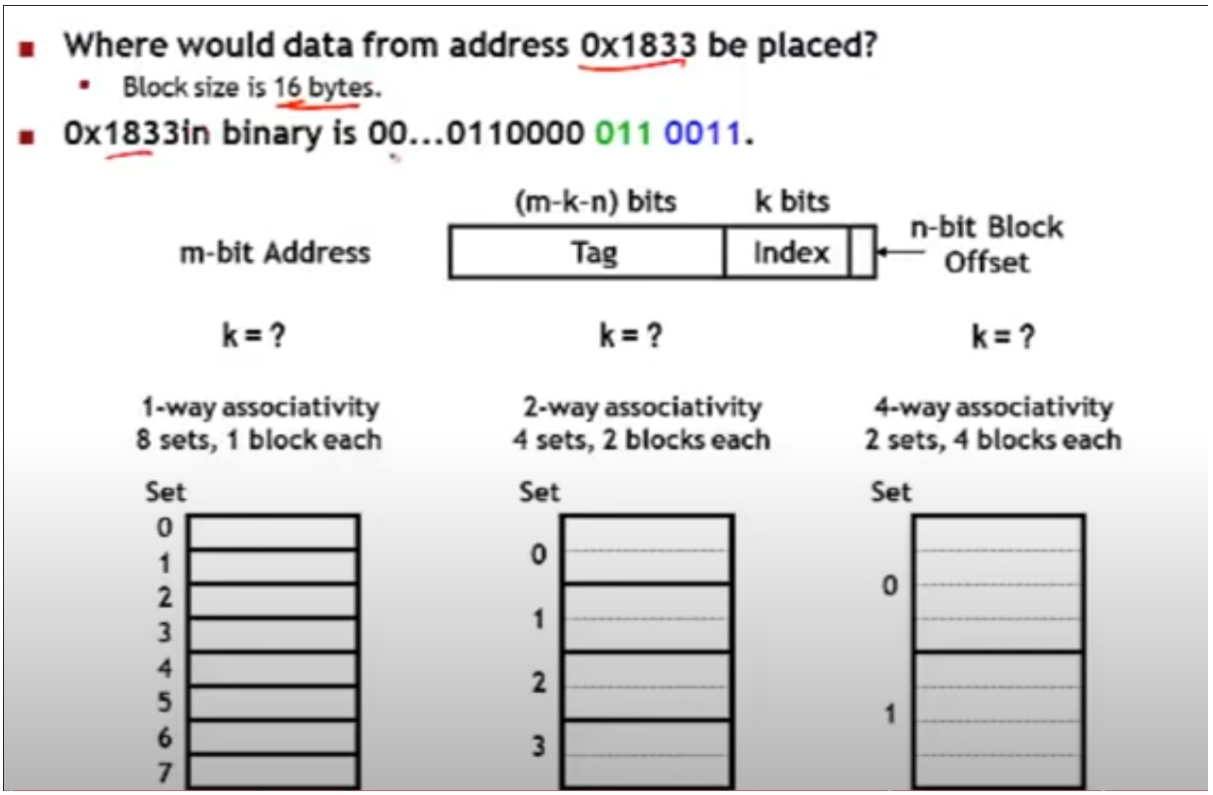
Used a 2^2 -block cache with 2^1 bytes per block, where would 13(1101) be stored?

We need 2 bit for the index because we have 4 blocks 2^2

2 bytes per block, that's mean we need 2^1 bits for offset

The rest for the tag => 1

Example:



Block size is 16 bytes => 2^4

So offset is 4

Index:

It depends on the associtivity if we have 8 diffrents sets

8 = 2^3 that's mean we need 3

if we have 4 diffrents sets

4 = 2^2 that's mean we need 2

The rest for the tag

Even though caches make it very easy and simple to the CPU to access data and instructions, we have a problem which is large programs can't fit in main memory especially when the system must switch between different programs.

But even if the main memory is large enough, there might be another problem which is that programs share the same memory, and their data and instructions must be protected from corruption

virtual memory

was developed so that main memory can function as a cache for secondary storage

Main memory works as a cache where virtual memory works as a secondary memory where all the principle of caching are applied here such as locality

To avoid that one program does not use the same memory as another program is using, there would be created a page table which translates virtual address into physical address.

The time to access a location in main memory can be doubled because when we want to execute a location in main memory and then our program has its instruction in virtual memory, then we need to translate the virtual address into a physical address which means to go from virtual memory "hard disk" into physical memory "main memory"

Swape space:

Is a block of the active and running data and instructions that are kept the main memory

Page

Like in CPU cache we have block that contain data and instructions, we do have in virtual memory

Pages that are blocks, that contain data and instructions

The size of each page ranges between 4 to 16 kilobytes

How virtual memory does work?

Load data from disk

Access the page table in ram

Translate the address

Update the cache

Have the OS update the cache

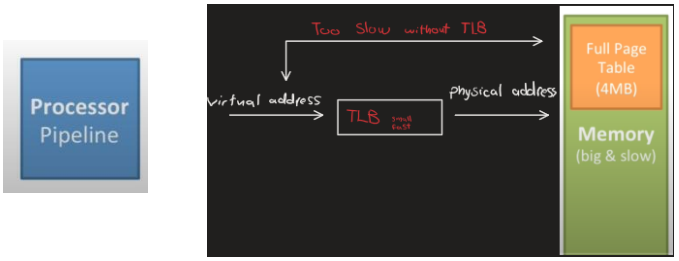
Access the data in RAM

Virtual memory is great, but it comes at high cost every memory operation has to look up in the page table, need to access the page table and the memory address

How can we make a page table look up really fast?

To make the virtual memory fast we need to add a special page table cache

The translation lookaside Buffer (TLB)



Instruction-level parallelism

Is all about having more processers working together to execute more instructions at the same time then we can execute a program much faster. There are two approaches to Instruction-level parallelism "ILP"

Pipelining

Pipelining is about doing more work at the same time independently.

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

Instead of having one processer doing the whole work one step at a time and cannot jump to next step before finishing the first step, we can have multiple processers that can share the tasks and each processer can do a single task then we can compound the work and get the final result.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Multiple issue

Multiple issue means that we halve the work by two.

```
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

While the first adder is computing z[0] the second adder can compute z[1]. While the first adder computes z[2] the other adder computes z[3] and so on.

Static multiple issue

We say the multiple issue system is static if the functional units are scheduled at compile time.

Dynamic multiple issue

We say the multiple issue system is static if the functional units are scheduled at run time.

Superscalar

A processor that supports dynamic multiple issue

Hardware multithreading

It is difficult to execute ILP “Instruction-level parallelism” because many statements are in a sequence and are depended on each other.

Thread-level parallelism

Thread-level parallelism is about instead of in ILP we divide the tasks within a single thread, we divide the work by switching between whole threads as we are switching between tasks. It is much faster to switch between threads than switching between processors. For example, if in a thread, there is waiting for some data to be fetched from memory, we can switch the thread and run another thread to save time. The system mush support rapid switching between threads to make this useful.

Fine-grained

There is a switch between threads after every single instruction that has been executed.

Coarse-grained

There is a switch between threads only when there is a task inside a thread requires long time to be executed such as loading/fetching data from memory.

2.3 Parallel hardware

SIMD systems

MIMD systems

Interconnections networks

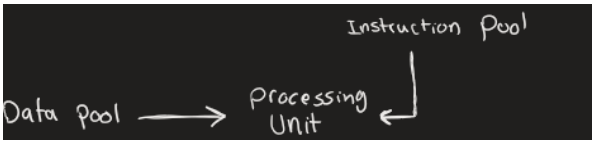
Cache coherence

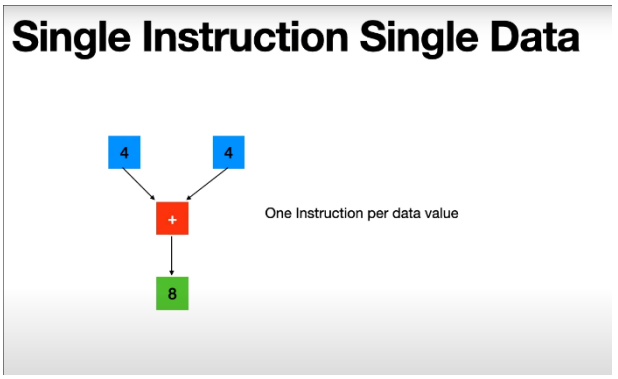
Shared memory vs distributed memory

What are SISD systems?

It executes a single instruction at a time, and it can fetch or store one item of data at a time.

- Single processor takes data from a single address in memory and preforms a single instruction on the data at a time
- Pipelining can be implemented, but only one instruction will be executed at a time
- All single processor systems are SISD





What are SIMD systems?

SIMD systems operate on multiple data streams by applying the same instruction

An abstract SIMD system can be thought of as having a single control unit and multiple ALUs

Another definition:

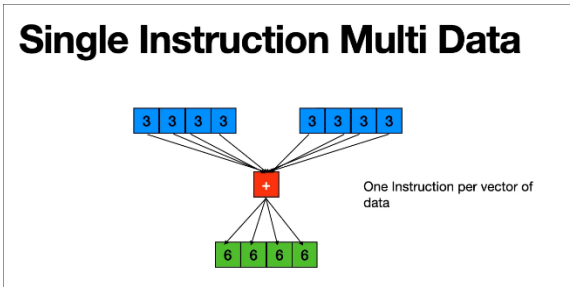
A single instruction is executed on multiple different pieces of data.

These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using

Multiple processors

Example from book:

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



If the system has m ALUs and m<n, for example m = 4 and n = 15

We can simply execute the additions in blocks of m elements at a time

In the last group of elements in our example- element 12 to 14 we are only operating on three elements of x and y, so one of the four ALUs will be idle

---

What are MIMD systems?

The MIMD consist of a collection of fully independent processing units or cores each of which has its own control unit and its own ALU

MIMD systems are usually asynchronous, that is the processors can operate at their own pace

In many MIMD system there is no global clock, and there may be no relation between the system times on two different processors

---

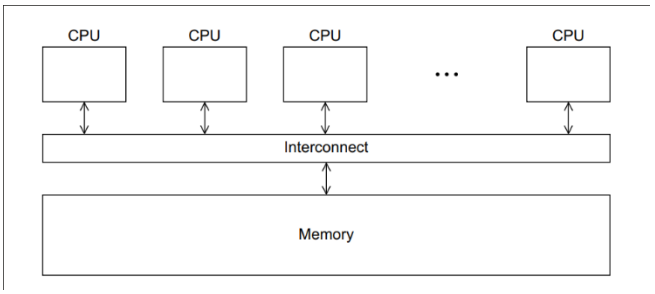
We have two types of MIMD?

Shared-memory systems OpenMP

Distributed memory systems MPI

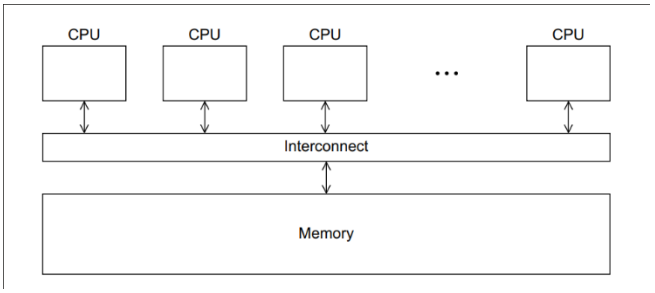
Shared-memory systems OpenMP

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



Distributed-memory systems MPI

In 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 system the processor usually communicates explicitly by sending message or by using special functions that provides access to the memory of another processor



Shared-memory systems

Use one or more multicore processors

What is multicore processor?

Multicore processor has multiple CPUs or cores on a single chip

The cores have private level 1 caches

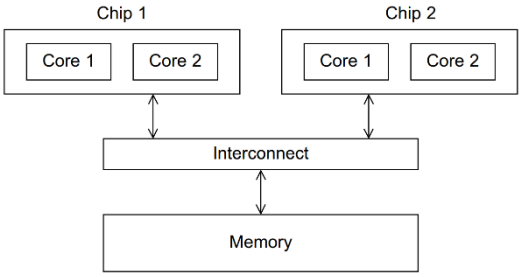
other caches may or may not be shared between the cores.

The interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory

Uniform memory access UMA

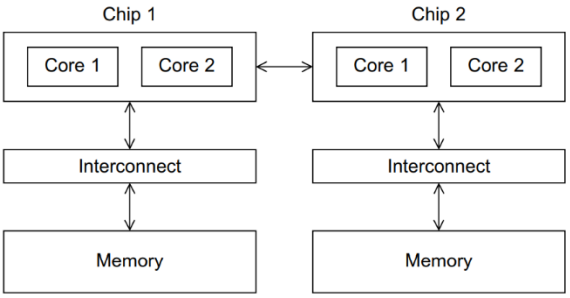
the time to access all the memory locations will be the same for all the cores

UMA systems are usually easier to program since the programmer doesn't need to worry about different access times for different memory locations.



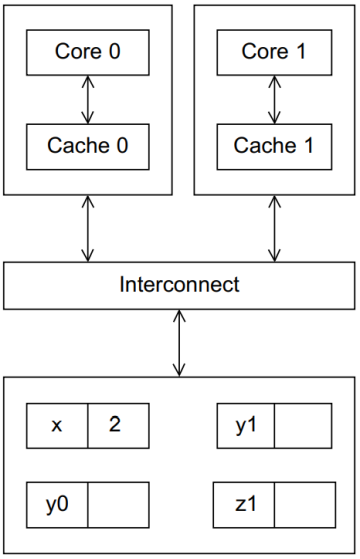
Nonuniform memory access, or NUMA

the time to access all the memory locations will be the same for all the cores



Cache coherence

Suppose we have a shared-memory system with two cores, each of which has its own private data cache. See Figure 2.17. If the two cores only read shared data, there is no problem. For example, suppose that x is a shared variable that has been initialized to 2, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1. Now suppose the following statements are executed at the indicated times:



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;`

2.4 Parallel Software

a problem because we can no longer rely on hardware and compilers to provide a steady increase in application performance. If we're to continue to have routine increases in application performance and application power, software developers must learn to write applications that exploit shared- and distributed-memory architectures

In this section we'll look at some of the issues involved in writing software for parallel systems.

shared-memory programs	distributed-memory programs
we'll start a single process and fork multiple threads we'll talk about threads carrying out tasks	we'll start multiple processes, and we'll talk about processes carrying out tasks

Caveats

Single program, multiple data SPMD

consist of a single executable that can behave as if it were multiple different programs using conditional branches.

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

Observe that SPMD programs can readily implement data-parallelism. For example,

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Recall that a program is task parallel if it obtains its parallelism by dividing tasks among the threads or processes. The first example makes it clear that SPMD programs can also implement task-parallelism.

Coordinating the processes/threads

What is load balancing?

Recall that the process of dividing the work among the processes/threads so that (a) is satisfied

Parallelization

The process of converting a serial program or algorithm into a parallel program

Embarrassingly parallel

simply dividing the work among the processes/threads

Dynamic and static threads

Dynamic threads	Static threads
In many environments shared-memory programs use dynamic 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. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.	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 clean-up (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.
However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It also has the virtue that it's closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other. Hence, we'll often use the static thread paradigm	

Nondeterminism

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

A computation is nondeterministic if a given input can result in different outputs

hence the results of the program may be different from run to run. As a very simple example, suppose we have two threads, one with id or rank 0 and the other with id or rank 1. Suppose also that each is storing a private variable `my_x`, thread 0's value for `my_x` is 7, and thread 1's is 19. Further, suppose both threads execute the following code:

```
. . .
printf("Thread %d > my_val = %d\n", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

but it could also be

Suppose each thread computes an `int`, which it stores in a private variable `my_val`. Suppose also that we want to add the values stored in `my_val` into a shared-memory location `x` that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

Now recall that an addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, we'll assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Clearly this is not what we want, and it's easy to imagine other sequences of events that result in an incorrect value for `x`. The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location `x`. When threads or processes attempt to simultaneously access a

Race condition:

because the threads or processes are in a "horse race." That is, the outcome of the computation depends on which thread wins the race.

A critical section

A block of code that can only be executed by one thread at a time

Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a **mutex function**

when it's done executing the code in the critical section, it should "relinquish" the mutex by calling an **unlock function**

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

This insures that only one thread at a time can execute the statement `x += my_val`. Note that the code does *not* impose any predetermined order on the threads. Either thread 0 or thread 1 can execute `x += my_val` first.

There are alternatives to mutexes. In **busy-waiting**, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable `ok_for_1` that has been initialized to false. Then something like the following code can insure that thread 1 won't update `x` until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);  /* Busy-wait loop */
x += my_val;           /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;    /* Let thread 1 update x */
```

So until thread 0 executes `ok_for_1 = true`, thread 1 will be stuck in the loop `while (!ok_for_1)`. This loop is called a "busy-wait" because the thread can be

Thread safety

Distributed memory

In distributed-memory programs, the cores can directly access only their own, private memories

2.5 INPUT AND OUTPUT

In distributed-memory programs, only process 0 will access stdin. In sharedmemory programs, only the master thread or thread 0 will access stdin.

In both distributed-memory and shared-memory programs, all the processes/ threads can access stdout and stderr

However, because of the nondeterministic order of output to stdout, in most cases only a single process/thread will be used for all output to stdout. The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to stdout.

Only a single process/thread will attempt to access any single file other than stdin, stdout, or stderr. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file

Debug output should always include the rank or id of the process/thread that's generating the output.

2.6 PERFORMANCE

2.6.1 Speedup and efficiency

p cores: one thread or process on each core

T_{serial} = the serial run-time

$T_{parallel}$ =parallel run-time

$T_{parallel} = T_{serial} / p$ parallel program has linear speedup.

the speedup of a parallel program to be

$$S = \frac{T_{serial}}{T_{parallel}}$$

then linear speedup has $S = p$, which is unusual. Furthermore, as p increases, we expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that S/p will probably get smaller and smaller as p increases. Table 2.4 shows an example of the changes in S and S/p as p increases.[1]

This value, S/p, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.[1] Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in

the same system. A server host runs one or more server programs, which share their resources with clients. A client usually does not share any of its resources, but it requests content or service from a server. Clients, therefore, initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client-server model are email, network printing, and the World Wide Web.