February 03, 2022

A process can run on one single core.

You can't improve performance by creating more instances of a process

Modify/write new code so you can leverage more processors.

Parallel programming → Performance programming

Programmer must divide application so that

- each processor has roughly equal at same time
- overhead of scheduling & coordination does not fritter away the potential performance benefits of parallelism.

ex:

- one task runs faster than other & one CPU waits for other completion then the CPU is also losing performance

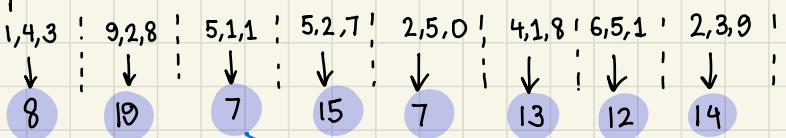
A general conversion from a serial program to a parallel program does not always perform well. → it is sometimes better to devise a new algorithm altogether.

Example: serial to parallel

each of the cores runs the code, taking n/p values

each core will have different values of my_sum

Ex: p = 8 cores, n = 24



Values of my_sum across cores

main core to do final my_sum

this step can be added to the code

runs the "if"

each core is programmed to send to main/"master" core w/ a if/else

runs the "else"

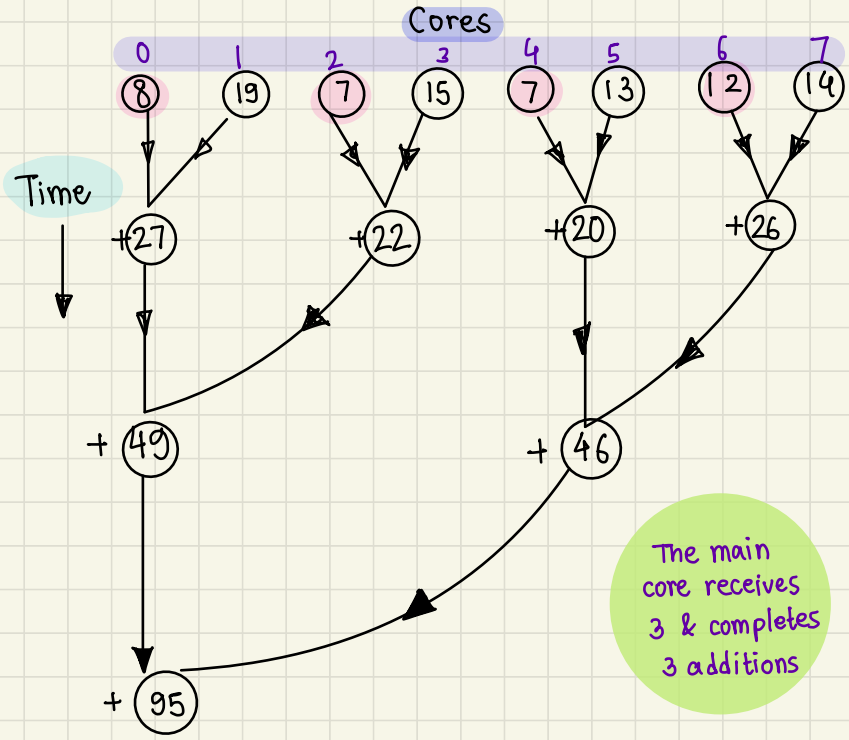
we can use a master core to add the final result

The main core does 7 receives & 7 additions

This is very similar to the idea of divide & conquer.

Can we do better?

The master code does not do all the work - the computation is shared among the other cores. - the cores are paired to add results.



The main core receives 3 & completes 3 additions

- We reduced from 8 numbers to 4 numbers
- we split the 7 additions to different cores. Each core has a small number of additions.
- we might not split evenly but we can still reduce the number of additions

- If you divide the size of the problem by 2, you get $\log_2 x$

Improvement by a factor of 100

The effect/difference is more dramatic w/ a larger number of cores.

How to write parallel programs?

There are two ways:

① Task parallelism - partition different tasks among cores

② Data parallelism - divide input data among cores

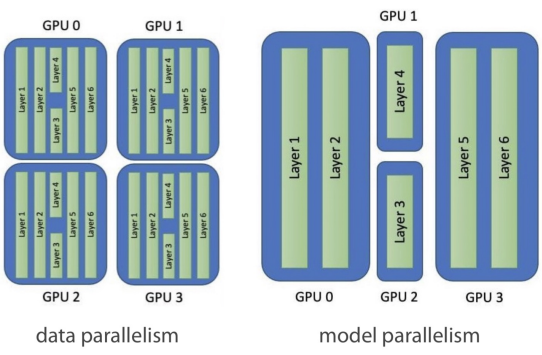
Ex: 15 Q's 300 Exams & 3 TA's

- Data parallelism
  - TA#1: 100 exams
  - TA#2: 100 exams
  - TA#3: 100 exams
- Task parallelism:
  - TA#1: Q 1-5 in all exams
  - TA#2: Q 11-15 in all exams
  - TA#3: Q 6-10 in all exams

Parallelism in GPU → take the slide Data parallelism & Model parallelism

Division of work – data parallelism

```
1 my_sum = 0;
2 my_first_i = . . . ;
3 my_last_i = . . . ;
4 for (my_i = my_first_i; my_i < my_last_i; my_i++) {
5     my_x = Compute_next_value( . . .);
6     my_sum += my_x;
7 }
```



data parallelism model parallelism

Division of work – task parallelism

```
1 if (I'm the master core) {
2     sum = my_X;
3     for each core other than myself {
4         receive value from core;
5         sum += value;
6     }
7 } else {
8     send my_x to the master;
9 }
```

Tasks
1) Receiving
2) Addition

Cores need to coordinate their work

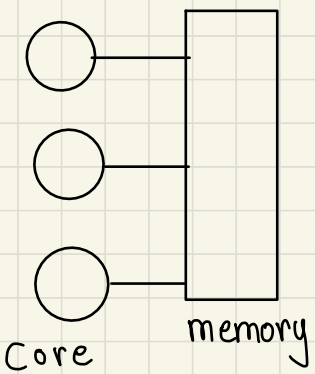
① Communication
② Load balancing
③ Synchronization

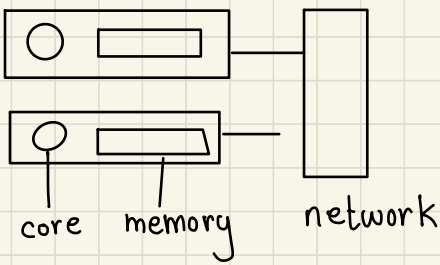
Types of Parallel Systems

February 10, 2022

① Shared memory
- the cores can share access to the computer's memory
- coordinate the cores by having them examine & update shared memory locations

② Distributed Memory
- each core has its own private memory
- cores must communicate explicitly over a network





- **Concurrent computing** – a program is one in which multiple tasks can be in progress at any instant.
- **Parallel computing** – a program is one in which multiple tasks cooperate closely to solve a problem
- **Distributed computing** – a program may need to cooperate with other programs to solve a problem.

- Concurrent Computing: a program is one in which multiple tasks can be in progress at any instant

- Distributed computing
  ex: internet
- Parallel computing
  - more narrow & closely related tasks
  ex: multiple cores on a computer with multiple threads

MPI is the model for distributed memory system

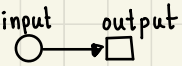
Concluding Remarks

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.

- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually very complex and therefore, require sound program techniques and development.

Parallel Hardware & Parallel Software

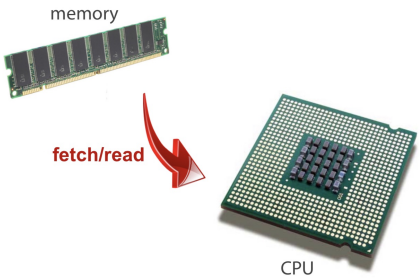
Some background

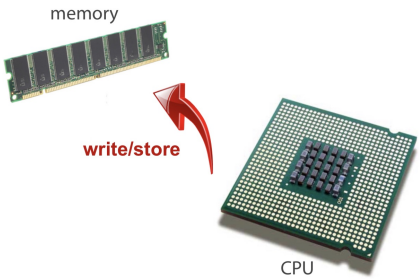


- computer runs one program at a time
  - the architecture behind it is the Von Neumann Architecture
    - CPU & Memory connected by bus
    - CPU has
      - ALU: calculations
      - Control registers: decoding the instructions & instruct the CPU for corresponding functionality
    - Registers store input & output data
    - Load instructions can move data from & between CPU registers & main memo
- Main memory is a collection of locations, each of which is capable of storing both instructions & data in von neumann architecture
- CPU: (see before)
  - Register: very fast storage
  - Program counter: find instruction for next step
  - Bus: hardware & wires connecting CPU & memory
- Von Neumann Bottleneck: the bottleneck can become the bus
- A operating system process: An instance of a computer program being executed.

  Components of a process:
  - The executable machine language program
  - A block of memory
  - Descriptors of resources OS allocated to the process
  - Security information
  - Information of the state of process

    ex: kernel state
- Multitasking: running multiple processes virtually at the same time. Time on the CPU is split into time slices used by processes. One process gets one time slice, & once it's up it waits for another time slice. If a process needs to wait for a resource, it will block & the OS moves to another process.





An operating system "process"

An instance of a computer program that is being executed.

Components of a process:

- The executable machine language program.
- A block of memory.
- Descriptors of resources the OS has allocated to the process.
- Security information.
- Information about the state of the process.

Multitasking

Gives the illusion that a single processor system is running multiple programs simultaneously.

Each process takes turns running. (time slice)

- After its time is up, it waits until it has a turn again.

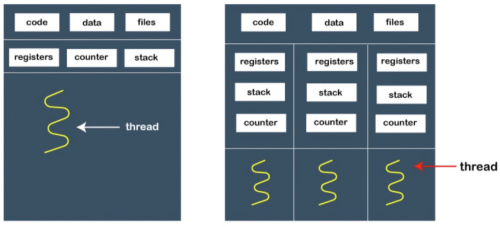
If a process needs to wait for a resource, it will stop executing and the operating system can run another process. (block)

Threading

Threads are contained within processes.

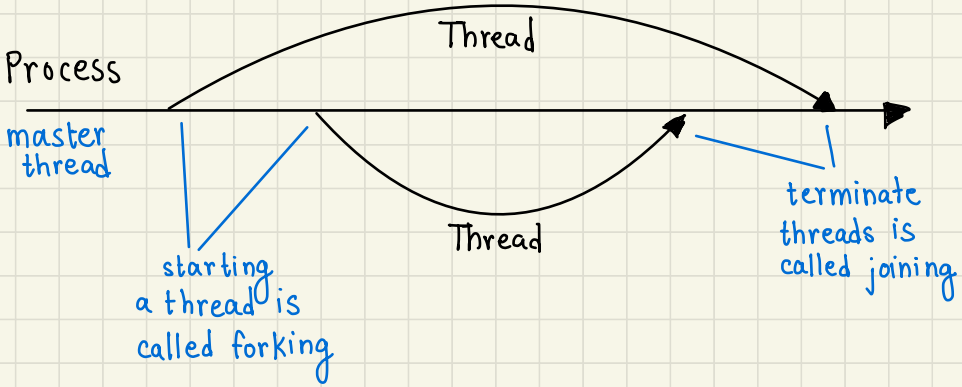
They allow programmers to divide their programs into (more or less) independent tasks.

The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.



Single-threaded process Multi-threaded process

- Threading : They allow programmers to divide their programs into (more or less) independent tasks.
  When one thread blocks, another can run.
  Threads are contained within processes.



- Caching : A collection of memory locations that can be accessed in less time than some other memory locations.
  A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.
- Principle of locality : Accessing one location is followed by an access of a nearby location
  Spatial locality : accessing a nearby location
  Temporal location: accessing in the near future

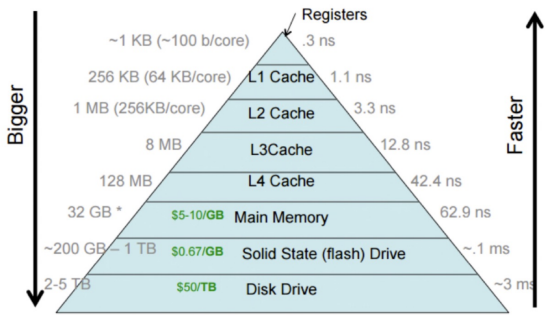
Von Neumann Model - Modifications to it

Basics of caching

A collection of memory locations that can be accessed in less time than some other memory locations.

A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

Memory hierarchy



Different levels of cache between local & main memory

The bigger the cache, the slower the access time

In general a **cache** is a collection of memory locations that can be accessed in less time than some other memory locations. In our setting, when we talk about caches we'll usually mean a **CPU cache**, which is a collection of memory locations that the CPU can access more quickly than it can access main memory. A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache. The universally used principle is based on the idea that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs typically execute the next instruction; branching tends to be relatively rare. Similarly, after a program has accessed one memory location, it often accesses a memory location that is physically nearby. An extreme example of this is in the use of arrays. Consider the loop

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

Accessing one location is followed by an access of a nearby location.

Spatial locality

accessing a nearby location

Temporal locality

accessing in the near future

Principle of locality

```
1 float z[1000];
2 …
3 sum = 0.0;
4 for (i = 0; i < 1000; i++)
5    sum += z[i];
```

The principle that an access of one location is followed by an access of a nearby location is often called **locality**. After accessing one memory location (instruction or data), a program will typically access a nearby location (**spatial** locality) in the near future (**temporal** locality).

In order to exploit the principle of locality, the system uses an effectively *wider* interconnect to access data and instructions. That is, a memory access will effectively operate on blocks of data and instructions instead of individual instructions and individual data items. These blocks are called **cache blocks** or **cache lines**.

Level of Cache

L1 - fastest & smallest
L2
L3 - largest & slowest

In computer networking, latency is **an expression of how much time it takes for a data packet to travel from one designated point to another**. Ideally, latency will be as close to zero as possible.

Cache can provide higher access latency
Cache saves a copy of data from main memory

Cache hit

- if we have cache we look through levels of cache for the data.
- all values in cache are also in memory
- you find the value

Cache miss → example: we want to fetch x in cache but it's only in main memory

- value not available in L1, L2, L3 & eventually we go back to main memory to load the data
- if we always have cache miss, it is not helpful

However, the value of cache may be inconsistent with main memory.
There are different policies to handle this.

1. Write-through cache: update the data in memory at the time it is written to cache. This is more reliable as you have less chance to lose data.

2. Write-back cache: mark data in cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.
   - this can provide higher performance but we might use write through is not always available, & you have less chance to lose data.

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.

Cache Mappings

1. Full associative: a new line can be placed at any location in the cache
2. Direct mapped: each cache has a unique location that it is mapped to However, this way the excess space might not be offerable even if accessible as it's uniquely mapped.
3. n-way set associative: each cache line can be placed in n-different locations of cache.

When more than one line in memory can be mapped to several different locations in cache so we need to be able to decide which line should be replaced or evicted.

When more than one line in memory can be mapped to several different locations in a cache (fully associative and n-way set associative), we also need to be able to decide which line should be replaced or **evicted**. In our preceding example, if, for example, line 0 is in location 0 and line 2 is in location 1, where would we store line 4? The most commonly used scheme is called **least recently used**. As the name

example of caching methods:

	Cache Location		
Memory Index	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Cache & Programs ?

If we load column by column instead of row by row in the cache, we still have space in the cache line.

If we visit by columns we have a better cache rate

Caches and programs

```
double A[MAX][MAX], x[MAX], y[MAX];
/* Initialize A and x, assign y = 0 */
/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j< MAX; j++)
        y[i] *= A[i][j] * x[j];
/* Assign y = 0 */
/* Second pair of loops */
for (j = 0; < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j] * x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

example, C stores two-dimensional arrays in "row-major" order. That is, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. So in row-major storage, we store row 0 first, then row 1, and so on. In the following two code segments, we would expect the first pair of nested loops to have much better performance than the second, since it's accessing the data in the two-dimensional array in contiguous blocks.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

first pair is faster than next pair since there are less cache misses since it accesses the loaded cache more often, there are less misses.

more ahead →

Solution:

Suppose the matrix has order 8 or 64 elements; the cache line size is still 4; the cache can store 4 lines; and the cache is direct-mapped. Then the following table shows how A is stored in cache lines.

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[0][4]	A[0][5]	A[0][6]	A[0][7]
2	A[1][0]	A[1][1]	A[1][2]	A[1][3]
3	A[1][4]	A[1][5]	A[1][6]	A[1][7]
4	A[2][0]	A[2][1]	A[2][2]	A[2][3]
5	A[2][4]	A[2][5]	A[2][6]	A[2][7]
6	A[3][0]	A[3][1]	A[3][2]	A[3][3]
7	A[3][4]	A[3][5]	A[3][6]	A[3][7]
8	A[4][0]	A[4][1]	A[4][2]	A[4][3]
9	A[4][4]	A[4][5]	A[4][6]	A[4][7]
10	A[5][0]	A[5][1]	A[5][2]	A[5][3]
11	A[5][4]	A[5][5]	A[5][6]	A[5][7]
12	A[6][0]	A[6][1]	A[6][2]	A[6][3]
13	A[6][4]	A[6][5]	A[6][6]	A[6][7]
14	A[7][0]	A[7][1]	A[7][2]	A[7][3]
15	A[7][4]	A[7][5]	A[7][6]	A[7][7]

Assuming that no lines of A are in the cache when the first pair of loops begins, we see that there will be two misses for each row of A. Also, after the first two rows have been read, the cache will be full, and each miss will evict a line. As in the example in the text, an evicted line won't need to be read again. So we see that the total number of misses for the first pair of loops is 16, or

$$\frac{\text{number of elements in A}}{\text{cache line size}}.$$

More generally, then, for the first pair of loops, the number of misses is only affected by the size of A, not the size of the cache.

For the second pair of nested loops, let's also assume that no lines of A are in the cache when the loops begin. When we're working with column 0 ($j = 0$), each time we multiply `A[i][0]*x[0]` we'll need to first load a new cache line, since the previously read lines will only contain elements from rows $< i$. So executing the loop with $j = 0$, will result in 8 misses. After reading in the four lines containing `A[0][0]`, `A[1][0]`, `A[2][0]`, and `A[3][0]`, respectively, subsequent reads of elements of column 0 will evict these four lines. So after completing the multiplications involving column 0 of A, no elements in the first 4 rows of A will be stored in the cache. So every read of an element of A in rows 0–3 of column 1 will result in a miss. In fact, we see that every multiplication will result in a miss and there will be 64 misses.

Observe, however, in the second pair of loops if we have an 8 line cache, then the multiplications involving columns 1–3 of A won't result in misses, and we won't have additional misses until we start the multiplications by elements in column 4. Once the lines containing elements of column 4 are loaded, there won't be any additional misses, and we see that with an 8 line cache, the total number of misses is reduced to 16.

Virtual Memory

We may not have memory for a large program in main memory.
Virtual memory functions as a cache for secondary storage.
It exploits the principle of spatial & temporal locality
It only keeps the active parts of running programs in main memory
Main memory serves as cache for execution of the program.

Virtual memory (1)

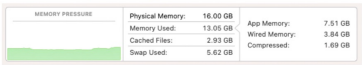
If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.

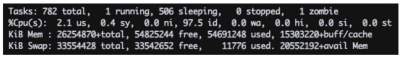
Virtual memory functions as a cache for secondary storage.

- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.

Virtual memory (2)

Swap space - those parts that are idle are kept in a block of secondary storage.

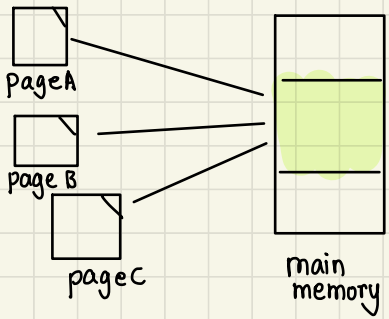




Pages – blocks of data and instructions.

- Usually these are relatively large
- Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.

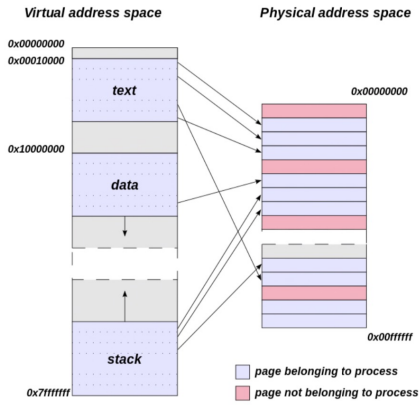
Swap space - those parts that are idle are kept in a block of secondary storage.
Pages - blocks of data & instructions.
Usually these are relatively large. Most systems have a fixed page size that currently ranges from 4-16 kilobytes.



Page table :
we have a virtual page number & the offset.
A virtual page number is mapped to a physical page number

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	···	13	12	11	10	···	1	0
1	0	···	1	1	0	0	···	1	1

Page table (2)



Translation-lookaside buffer (TLB)

TLB is a cache for the page table.

Using a page table may increase the runtime.

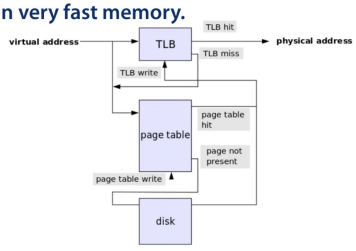
It caches a small number of entries (typically 16-512) from the page table in very fast memory.

Translation-lookaside buffer (TLB)

Using a page table has the potential to significantly increase each program's overall run-time.

A special address translation cache in the processor.

It caches a small number of entries (typically 16–512) from the page table in very fast memory.



Page fault – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

TLB hit : the page number found

TLB miss: the page number not in TLB. A new entry is then created from main memory to the TLB.

Instruction Level Parallelism (ILP)

Attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions.

Pipelining - functional units are arranged in stages.

Multiple issue - multiple instructions can be simultaneously initiated.

Pipelining example (1)

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

Add the floating point numbers
9.87×10^4 and 6.54×10^3

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

Pipelined Addition.
Numbers in the table are subscripts of operands/results.

Assume each operation takes one nanosecond (10^{-9} seconds).

```
1 float x[1000], y[1000], z[1000];
2 for (i = 0; i < 1000; i++) {
3     z[i] = x[i] + y[i];
4 }
```

This for loop takes about 7000 nanoseconds.

One floating point addition still takes 7 nanoseconds.

But 1000 floating point additions now takes 1006 nanoseconds!

Divide the floating point adder into 7 separate pieces of hardware or functional units.

First unit fetches two operands, second unit compares exponents, etc.

Output of one functional unit is input to the next.

Multiple Issue - Part of Instruction Level Parallelism

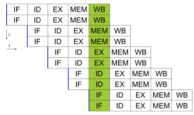
February 17, 2022

Multiple Issue (1)

Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

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

static multiple issue - functional units are scheduled at compile time.



dynamic multiple issue – functional units are scheduled at run-time.

superscalar

1. Static Multiple Issue
2. Dynamic multiple issue: superscalar

Speculation

- CPU can guess what instruction will be executed for the next few steps
- if it can guess correctly, we can improve performance.

In speculation, it might make a guess & make an independent similar instruction later.

```
z = x + y
if (z > 0)
    w = x
```

in speculation, the computer runs these instructions together simultaneously

Speculation (1)

In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.

In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

```
1 z = x + y;
2 if (z > 0)
3       w = x;
4 else
5       w = y;
```

If the system speculates incorrectly, it must go back and recalculate w = y.

Hardware Multithreading

If speculation isn't a good opportunity.
Ex- one result is dependent on the other closely like Fibonacci

There aren't always good opportunities for simultaneous execution of different instructions.

Ex. Fibonacci number with dynamic programming

Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.

Ex., the current task has to wait for data to be loaded from memory.

Then we can do multithreading

1. Fine grained multithreading (FMT)
   - CPU knows how many threads
   - the processor switches between threads after each instruction, skipping threads that are stalled
   - sequential instructions have to wait after every instruction.

Pros: potential to avoid wasted machine time due to stalls.

Cons: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

2. Course-grained Multithreading (CMT)
   - the processor switches threads only when threads are stalled
   - switching does not need to be instantaneous
   - con: there are switches in shorter stalls which can cause delays

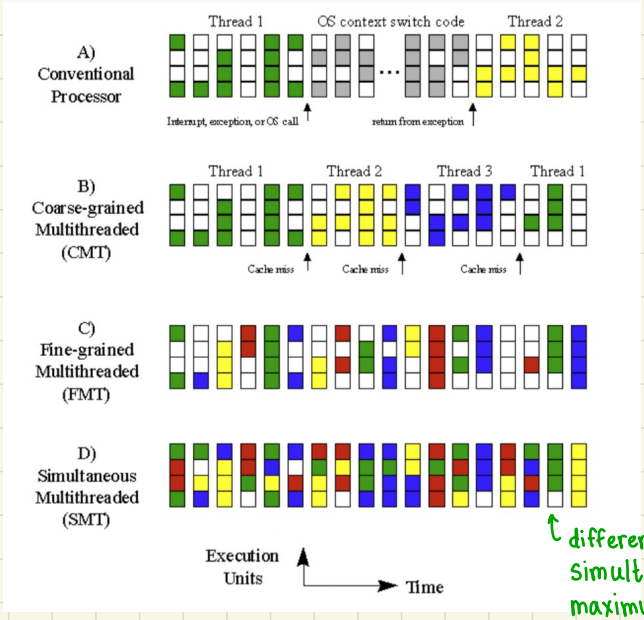
Pros: switching threads doesn't need to be nearly instantaneous.

Cons: the processor can be idled on shorter stalls, and thread switching will also cause delays.

3. Simultaneous Multithreaded
   - a thread carries out multiple instructions & allocates based on available resource

Simultaneous multithreading (SMT) - a variation on fine-grained multithreading.

Allows multiple threads to make use of the multiple functional units.



↑ different threads working simultaneously to ensure maximum utilization of resources

A. Conventional Processor

- switches happen through exception, interrupt or OS call
- there is an OS context switch call

Parallel I

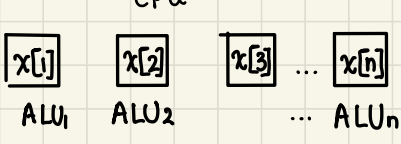
Flyn's Taxonomy

1. SISD
2. SIMD
3. MISD
4. MIMD

1. SIMD

- multiple data stream - data divided into multiple parts - data parallelism

ex




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

with multiple data units, we divide the data & make multiple rounds to carry out the task with different data points

- cons: All ALUs are required to execute the same instruction or remain idle
  Only effecient for large data parallel problems, not other types of more complex parallel programs

What if we don't have as many ALUs as data items?

Divide the work and process iteratively.

Ex. m = 4 ALUs and n = 15 data items.

Round3	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

SIMD drawbacks

All ALUs are required to execute the same instruction, or remain idle.

In classic design, they must also operate synchronously.

The ALUs have no instruction storage.

Efficient for large data parallel problems, but not other types of more complex parallel problems.

Vector processors (1)

Operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars.

Vector registers.

Capable of storing a vector of operands and operating simultaneously on their contents.

Vectorized and pipelined functional units.

The same operation is applied to each element in the vector (or pairs of elements).

Vector instructions.

Operate on vectors rather than scalars.

Interleaved memory.

Multiple "banks" of memory, which can be accessed more or less independently.

Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.

Strided memory access and hardware scatter/gather.

The program accesses elements of a vector located at fixed intervals.

Vector processors - Pros

Fast.

Easy to use.

Vectorizing compilers are good at identifying code to exploit.

Compilers also can provide information about code that cannot be vectorized.

Helps the programmer re-evaluate code.

High memory bandwidth.

Uses every item in a cache line.

Vector processors - Cons

They don't handle irregular data structures as well as other parallel architectures.

A very finite limit to their ability to handle ever larger problems. (scalability)

Bandwidth is **the data transfer capacity of a computer network in bits per second (Bps)**. The term may also be used colloquially to indicate a person's capacity for tasks or deep thoughts at a point in time.

ex: Vector Processor:

operate on arrays or vectors of data while conventional CPU's operate on individual data elements or scalars

- Vectorized & pipelined functional units
  the same operation is applied to each element in vector (or pairs of elems
  The operation is performed on a vector rather than scalar

- Interleaved memory
  - multiple "banks" of memory which can be accessed independently
    As they're accessed independentl there is less delay
  - Strided memory access & hardware scatter gather
    - the program accesses elements of a vector at fixed intervals

Vector processors Pros:

1. Uses every item in a cache line: as when data is carried to cache, the entire array of data is copied

Cons

1. Hard to scale: the number of functional components is fixed. A finite ability to handle larger problems
2. Can't handle irregular data.

Ex: GPU

Convert graphical data to pixels

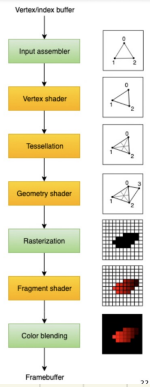
Graphics Processing Units (GPU)

Real time graphics application programming interfaces or API's use points, lines, and triangles to internally represent the surface of an object.

A graphics processing pipeline converts the internal representation into an array of pixels that can be sent to a computer screen.

Several stages of this pipeline (called shader functions) are programmable.

Typically just a few lines of C code.



Shader functions are also implicitly parallel, since they can be applied to multiple elements in the graphics stream.

GPU's can often optimize performance by using SIMD parallelism.

The current generation of GPU's use SIMD parallelism.

Although they are not pure SIMD systems.

2. MIMD

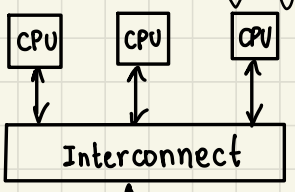
- processor can receive multiple streams of instructions & those instructions can receive multiple streams of data
- this is what we'll use in parallel computing

MIMD

Supports multiple simultaneous instruction streams operating on multiple data streams.

Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

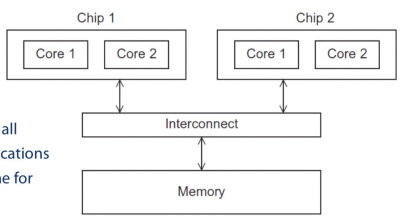
ex: Shared Memory System



ex: UMA multicore system
NUMA multicore system

UMA multicore system

shared Memory System



Time to access all the memory locations will be the same for all the cores.

Shared Memory System (1)

A collection of autonomous processors is connected to a memory system via an interconnection network.

Each processor can access each memory location.

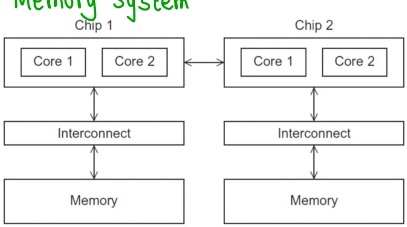
The processors usually communicate implicitly by accessing shared data structures.

Most widely available shared memory systems use one or more multicore processors.

(multiple CPU's or cores on a single chip)

NUMA multicore system

Shared Memory System



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

- Distributed Memory System
  - Cluster
    - A collection of commodity systems
    - each node of a cluster indicates a unit

Distributed Memory System

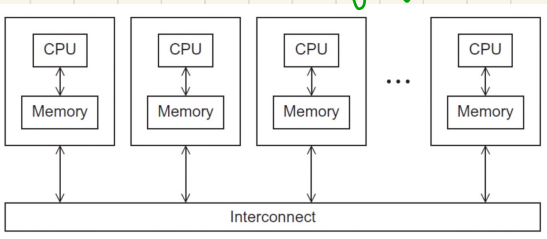
Clusters (most popular)

A collection of commodity systems.

Connected by a commodity interconnection network.

Nodes of a cluster are individual computations units joined by a communication network.

Distributed Memory System



Interconnection networks

Affects performance of both distributed and shared memory systems.

Two categories:

Shared memory interconnects

Distributed memory interconnects

Shared Memory Interconnects

Shared memory interconnects

Bus interconnect

A collection of parallel communication wires together with some hardware that controls access to the bus.

Communication wires are shared by the devices that are connected to it.

As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.

Switched interconnect

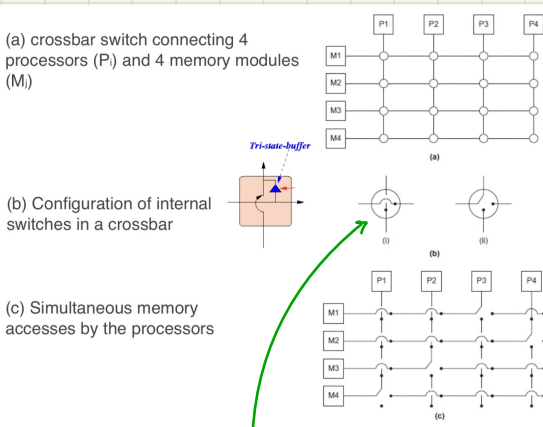
Uses switches to control the routing of data among the connected devices.

Crossbar

Allows simultaneous communication among different devices.

Faster than buses.

But the cost of the switches and links is relatively high. - eventually more expensive



internal of crossbar

- there are internal switches that can decide which memory visits which processor

Distributed Memory Interconnects

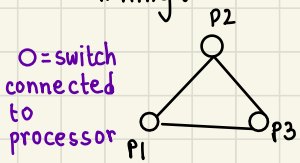
February 22, 2022

Two groups

① Direct interconnect: each switch is connected to a processor memory pair, & the switches are connected to each other.

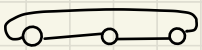
There can be different topology of switches

1. Ring:

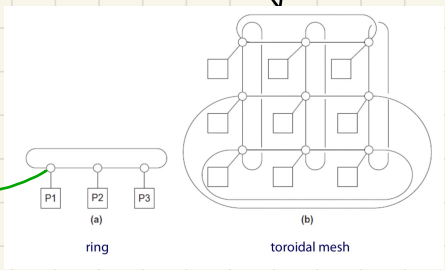


O = switch connected to processor

or

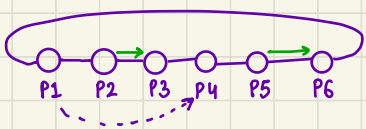


switch



There may be a case that there is contention in a link

Here, P2 cannot access P4 since there is no path available



2. Toroidal mesh: we have rings in higher dimensions.
   It will have more links & be more expensive
   Allows more concurrent communication at the same time.

② Indirect Interconnects
- each processor sends data to switching network
- switches may not be directly connected to a processor

Bisection Width: or "connectivity" / how many simultaneous connections between halves
- used to measure performance.
- a measure of number of "simultaneous connections" or connectivity

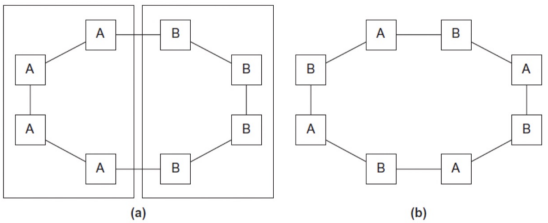
Bandwidth
- the rate at which a link can transmit data
- usually given in megabits or megabytes per second

Bisection bandwidth
- combination of throughput - it shows the bandwidth

A measure of network quality.

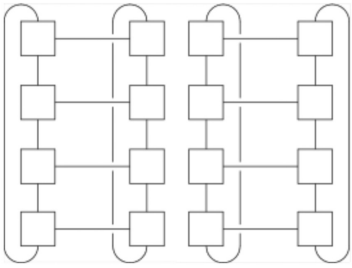
Instead of counting the number of links joining the halves, it sums the bandwidth of the links.

Two bisections of a ring



A bisection of a toroidal mesh

remove the minimum number of links needed to split the set of nodes into two equal halves



p : the number of processors

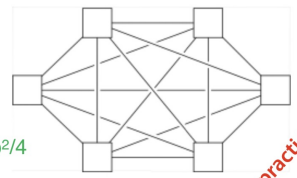
bisection width : $2\sqrt{p}$

Fully connected network

- each switch is directly connected to every other switch.
- bisection width = $p^2/4$ (total number of links)

Fully connected network

Each switch is directly connected to every other switch.

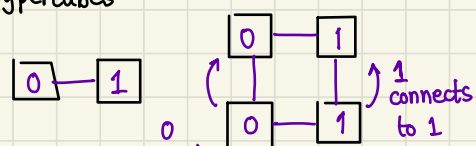


bisection width = $p^2/4$

impractical

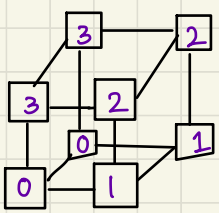
total link = p(p-1)/2

Hypercubes



2 switches

8 switches



32 switches

Number of switches increases 2^3

$p = 2^d$

bisection width = $\frac{p}{2} = 2^{d-1}$

Hypercube

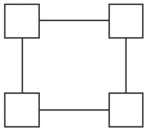
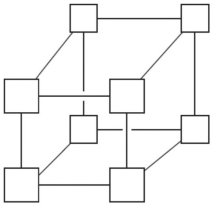
Highly connected direct interconnect.

Built inductively:

A one-dimensional hypercube is a fully-connected system with two processors.

A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches.

Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.

	d=1	d=2	d=3
	 one-dimensional	 two-dimensional	 three-dimensional
BW	1	2	4

$p = 2^d$ bisection width : $\frac{p}{2} = 2^{d-1}$

② Indirect Interconnects

- each processor sends data to switching network

Omega network

- total number of regular switches cross bar switches x 4

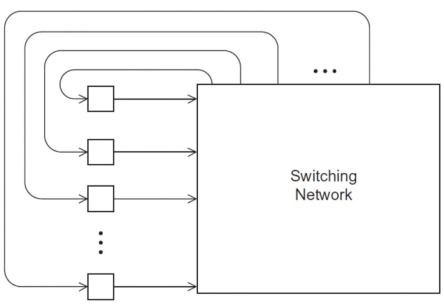
Indirect interconnects

Simple examples of indirect networks:

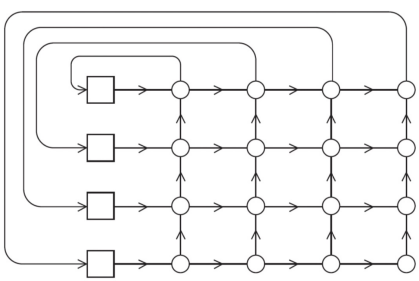
- Crossbar
- Omega network

Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.

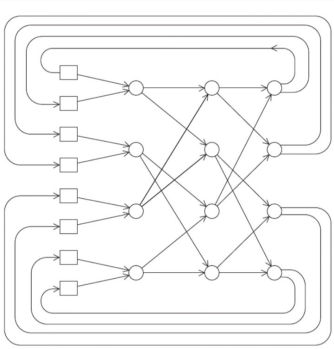
A generic indirect network



Crossbar interconnect for distributed memory

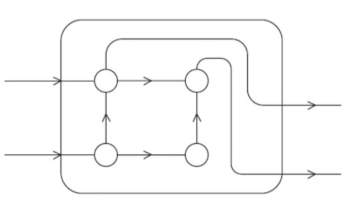


An omega network



$\frac{1}{2}p \log_2 p$ crossbar switches

A switch in an omega network



More definitions

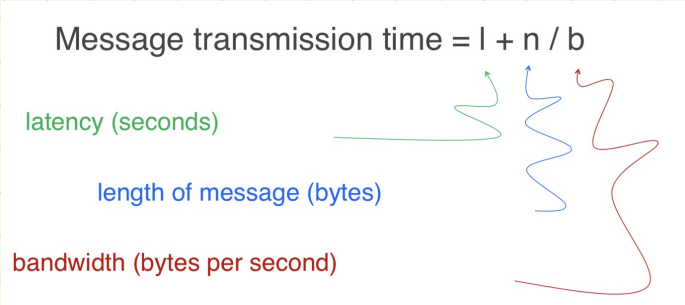
Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.

Latency

The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.

Bandwidth

The rate at which the destination receives data after it has started to receive the first byte.



Cache Coherence

As programmers have no control when cache gets updated, we have issue w/ parallel processing.

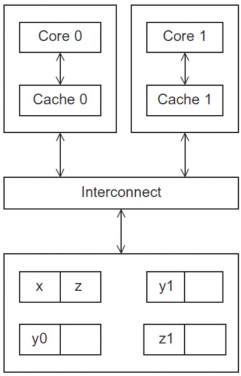
To enforce cache coherence, the cores share a bus.

When we have any write of data in the memory, it is placed on the bus so it can be seen by all the cores.

→ more on next page

Cache coherence

Programmers have no control over caches and when they get updated.



A shared memory system with two cores and two caches

Cache coherence

y0 privately owned by Core 0
y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

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;

y0 eventually ends up = 2
y1 eventually ends up = 6
z1 = ???

Snooping Cache Coherence

The cores share a bus .

Any signal transmitted on the bus can be "seen" by all cores connected to the bus.

When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.

If core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid.

Directory Based Cache Coherence

Uses a data structure called a directory that stores the status of each cache line.

When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated and it can mark its copy of x as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing x has been updated, not that x has been updated.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable x stored in core 1's memory, by simply executing a statement such as y = x.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

Answer to 2.15

(a) Core 0: x = 5
Core 1: y = x
Core 1 does not have x in its cache.
Core 0 uses snooping cache coherence but it uses write back cache which means that extra communication is necessary since the cache does not get immediately sent to memory. If the cache for x = 5 is sent to memory by the time Core 1 executes y = x, then y receives the value 5 otherwise it receives an old value of x which was already in memory.

(b) If it uses a directory based protocol, so when x = 5 is read into Core 0's cache, the directory entry corresponding to that line is updated to show that Core 0 has a copy of the line. However when a cache variable is updated only the cores storing that variable is updated. As Core 1 does not already have the value in its cache. Similarly from before if x is not updated in main memory, Core 1 will have an old copy of x, else if it is updated Core 1 will receive the updated value for x.

(c) In the last two problems, there is guaranteed cache coherence only if both cores already have the variable in cache. It should be the case that a core which does not have a value in it's cache should have the updated value. We can do this by employing a critical section, so that a core can only get the value of x once it's been updated in main memory.

Parallel Software

① SPMD

- single program multiple data

SPMD – single program multiple data

A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

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

From now on…

In shared memory programs:

- Start a single process and fork threads.
- Threads carry out tasks.

In distributed memory programs:

- Start multiple processes.
- Processes carry out tasks.

→ the burden is on software

Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.

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

2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

These last two problems are often interrelated.

Shared Memory

- Dynamic threads : real-time thread creation & deletion.
- Static threads
  - use case: network service : when we have a new incoming connection we create a new thread. We want to keep a pool of threads for less work & a lower overhead

Shared Memory

Dynamic threads

Master thread waits for work, forks new threads, and when threads are done, they terminate

Efficient use of resources, but thread creation and termination is time consuming.

Static threads

Pool of threads created and are allocated work, but do not terminate until cleanup.

Better performance, but potential waste of system resources.

Nondeterminism

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

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

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

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

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

Race condition

Critical section

Mutually exclusive

Mutual exclusion lock (mutex, or simply lock)

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

When we launch multiple threads, the OS has control over scheduling
Execution is non-deterministic in parallel programming. Threads can run at different times. x is a shared variable in the example

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

In order to avoid race condition we have a critical section. We can make the code mutually exclusive. We can place mutual exclusion lock (mutex, or simply lock)

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

→

Busywait : we can use busywaiting to specify sequence in the non-deterministic model

busy-waiting

```
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 */
```

Distributed Memory: each core has its own corresponding memory
In order to exchange data, we need specific computation core.
• You usually run distributed memory in multiple servers. Since it is not possible to run multiple threads in multiple servers, we use multiple processes

Distributed Memory

Message-passing

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

Message passing : passing messages from one core to another.

Input & Output : I/O operations lose performance

- we want to use conventions to not lose too much performance
- in most cases, only a single process/thread will be used for all output to stdout rather than debugging output. Debug output should always include the rank or id of the process/thread that's generating the output

Input and Output

In distributed memory programs, only process 0 will access *stdin*. In shared memory 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 indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

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

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.

Performance

Speedup

Number of cores = p

Serial run-time = T_{serial}

Parallel run-time = $T_{parallel}$

linear speedup

$$T_{parallel} = T_{serial} / p$$

$$S, \text{speedup} = \frac{T_{serial}}{T_{parallel}}$$

p = num of processors

$$E, \text{effecien cy of a parallel program} = \frac{S}{p} = \frac{T_{serial}}{p * T_{parallel}}$$

when we increase p, effeciency is not necessarily increasing.
As with more p, there is usually more overhead so there is not a linear increase.

Effeciency decreases with p increase when
- less data in each core - more core communication

more data will give you a higher effeciency.

$$T_{parallel} = T_{serial} / p + T_{overhead}$$

T_{serial}/p: can be optimized

$T_{overhead}$: cannot be optimized so it remains unchanged

ex: $E = \frac{20}{\frac{18/p + 2}{p}} = \frac{20}{18 + 2p}$ As p increases, effeciency decreases

as $p \to \infty$, $\frac{18}{p} \to 0$, so we have $\frac{20}{2} = 10$ speed up

you can only go 10x faster. You can't go 10x faster

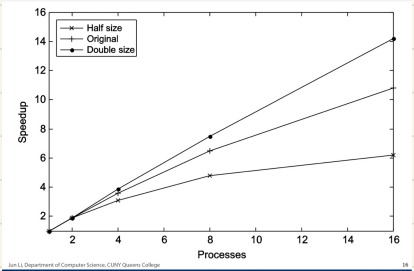
Speedups and efficiencies of a parallel program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

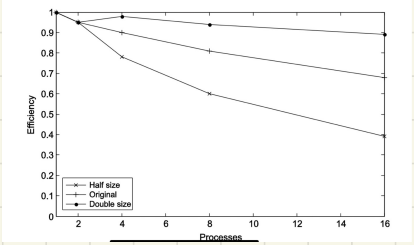
Speedups and efficiencies of parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speed-up



Efficiency



Amdahl's Law

Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.

Scalability

If we can increase the problem size & it maintains effeciency E then the problem is scalable.

① Strongly scalable:
- keep problem size same
- increase threads & processes

② Weakly scalable
- increase problem size in proportion to thread/process increase

Scalability

Suppose that we increase the number of processes/threads, if we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is scalable.

If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is strongly scalable.

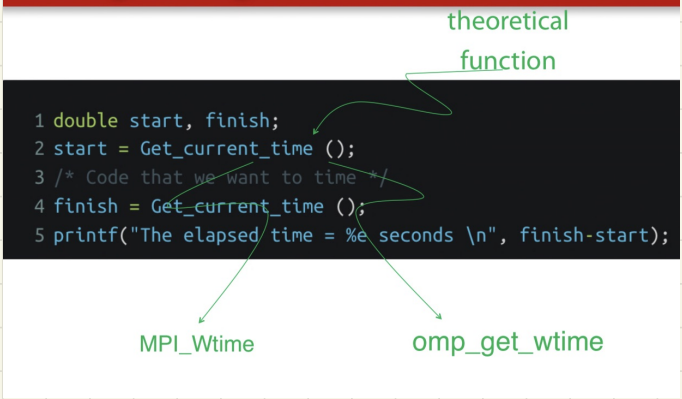
If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/ threads, the problem is weakly scalable.

Taking timing

- wall clock time
  - we may use it for tracking performance

we can have problem tracking time for distributed computing as each can have their own variable for tracking time

Taking Timings




```
1 private double start, finish;
2 start = Get_current_time ();
3 /* Code that we want to time */
4 finish = Get_current_time ();
5 printf ("The elapsed time = %e seconds \n", finish-start);
```

private is a part of the process

```
1 shared double global_elapsed;
2 private double my_start, my_finish, my_elapsed;
3 ...
4 /* Synchronize all processes/threads */
5 Barrier();
6 my_start = Get_current_time();
7
8 /* Code that we want to time */
9 ...
10
11 my_finish = Get_current_time();
12 my_elapsed = my_finish - my_start;
13
14 /* Find the max across all processes/threads */
15 global_elapsed = Global_max(my_elapsed);
16 if (my_rank == 0)
17     printf ("The elapsed time = %e seconds \n", global_elapsed);
```

global time elasped

Parallel Program Design

March 1, 2022

There is no one single way to have the best performance

Foster's methodology

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

We are going to partition it into small tasks

Foster's methodology

2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.

ex: input & output between different cores

Foster's methodology

3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.

   For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

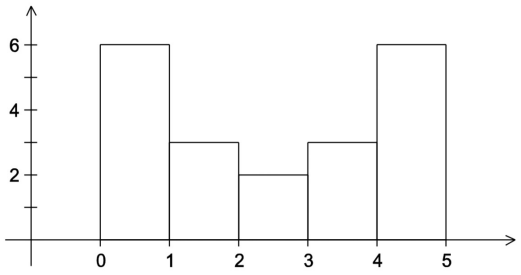
Foster's methodology

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Example - histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



we can try to make a serialized program to a parallelized program