

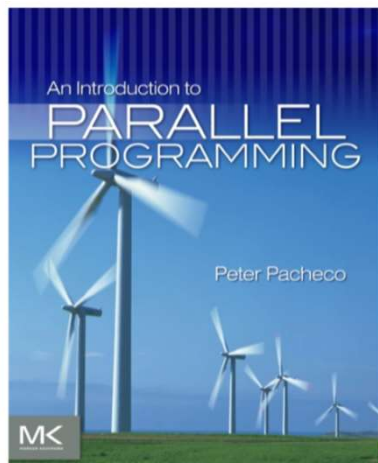
An Introduction to Parallel Computing/ Programming

Vicky Papadopoulou Lesta

Astrophysics and High Performance Computing Research Group

<http://ahpc.euc.ac.cy>

**Dep. of Computer Science and Engineering
European University**



The presentation is based on material of the book:

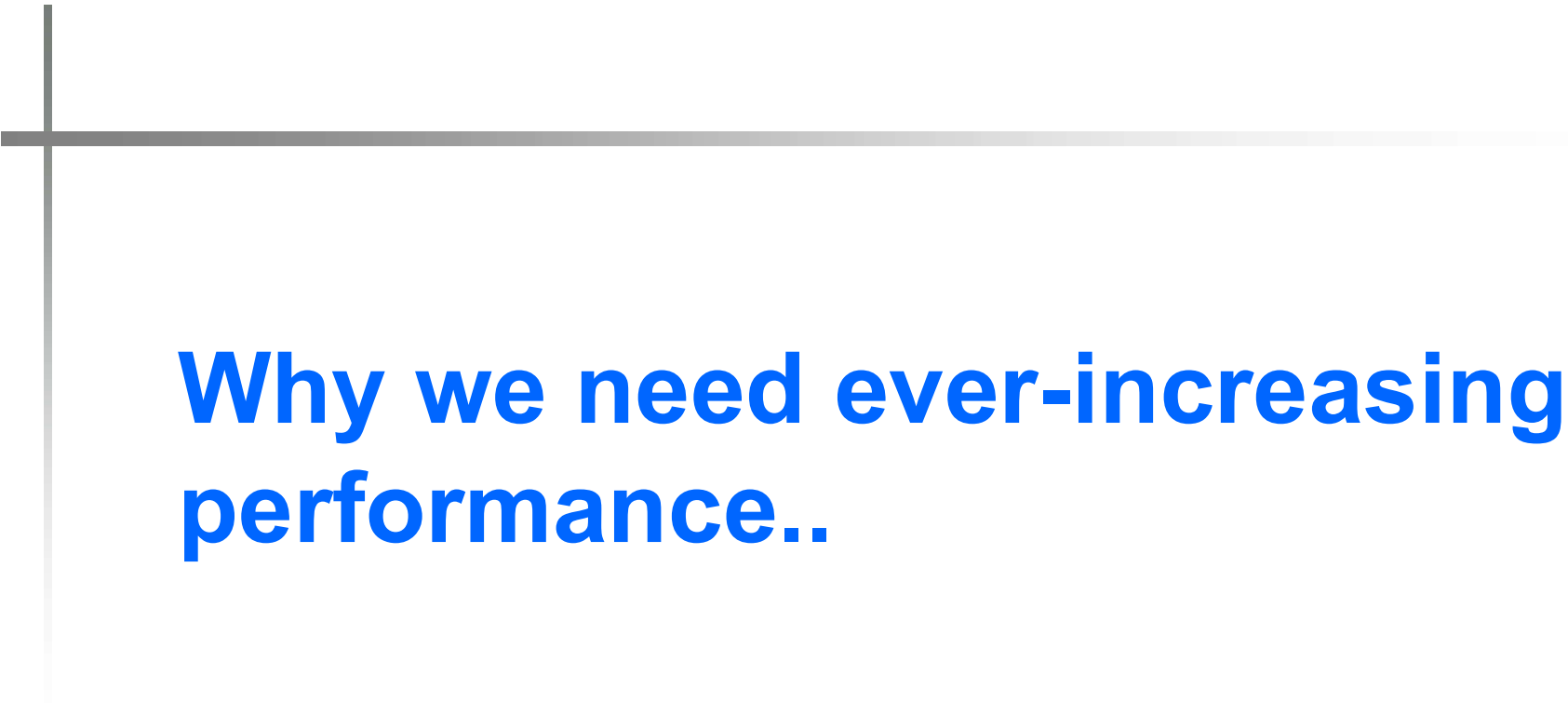
An Introduction to Parallel Programming

Peter Pacheco



- **Topic 1:**

Why Parallel Computing?



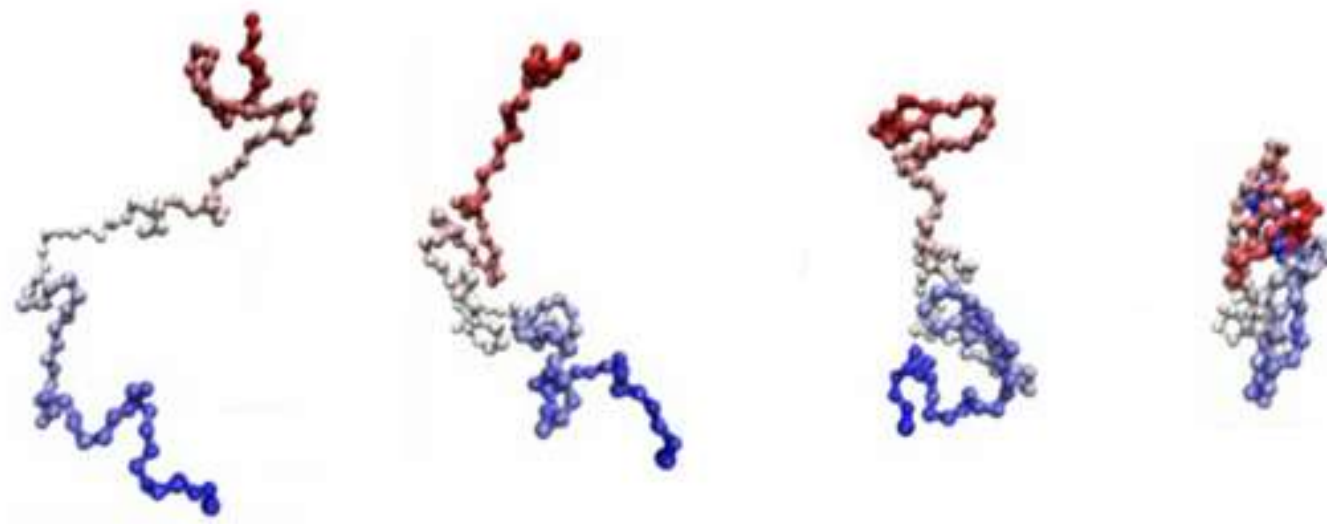
**Why we need ever-increasing
performance..**

Climate modeling



Protein folding

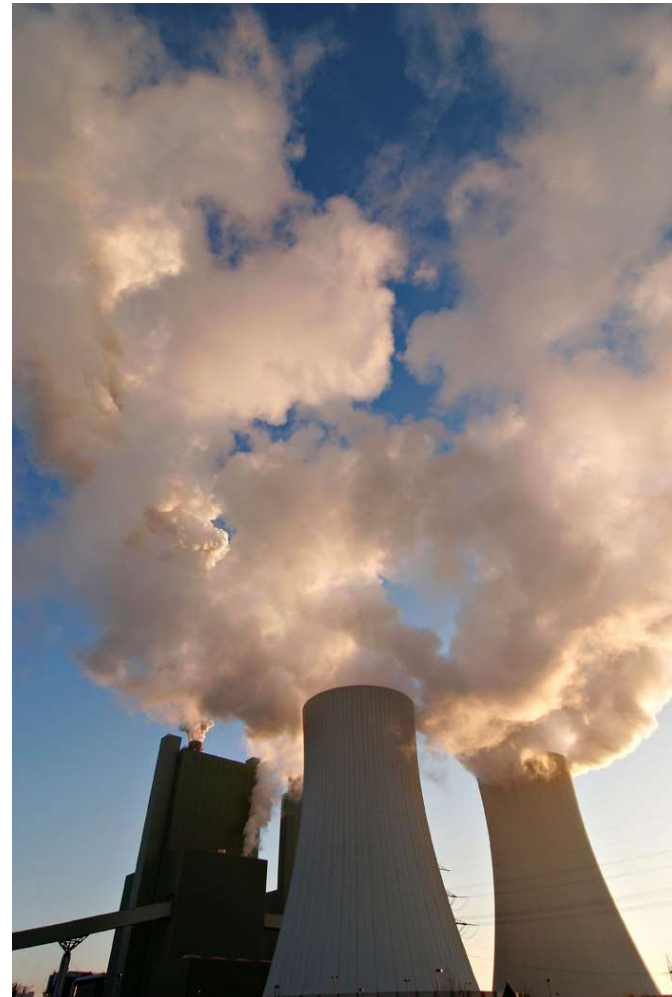
Misfolded proteins related to diseases like Parkinson, Alzheimer.



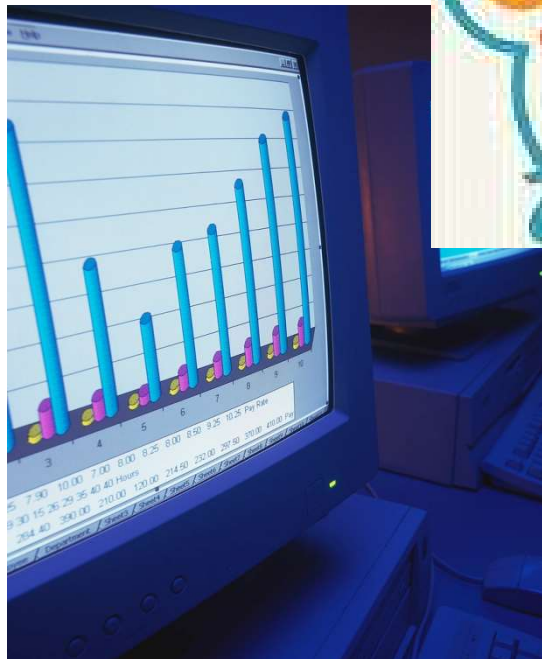
Drug discovery



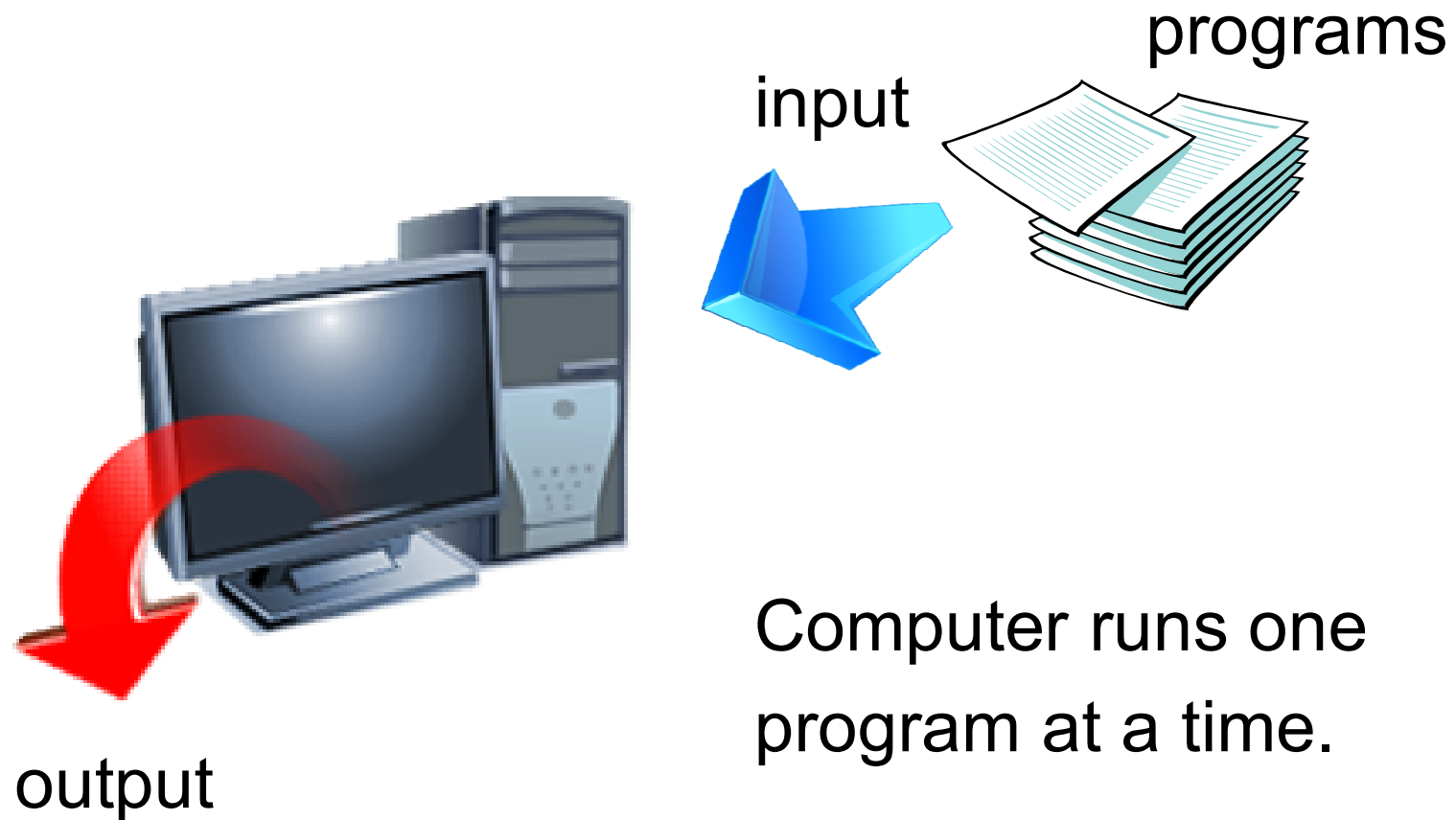
Energy research



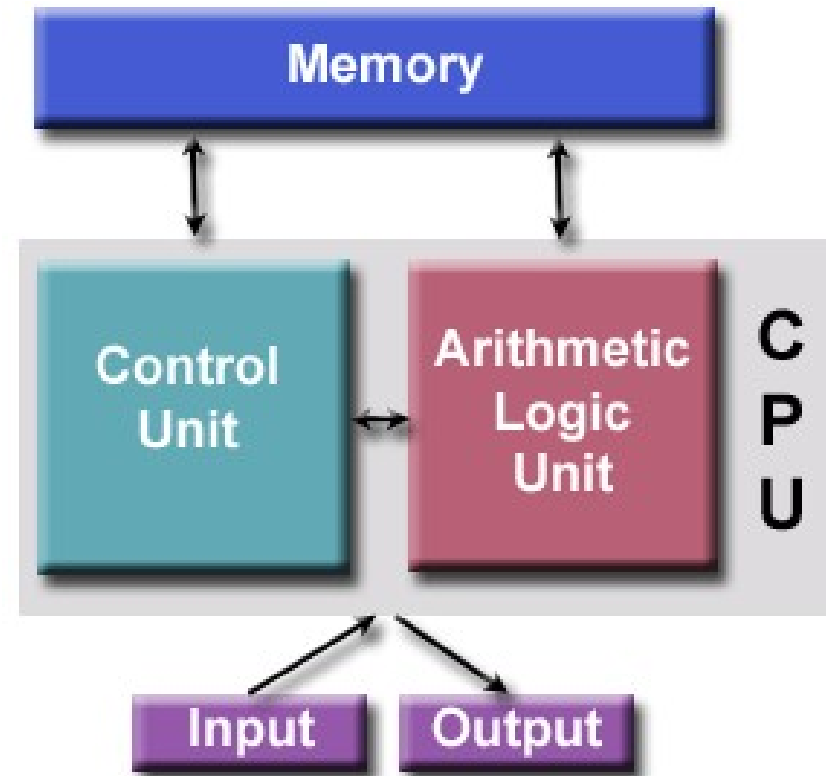
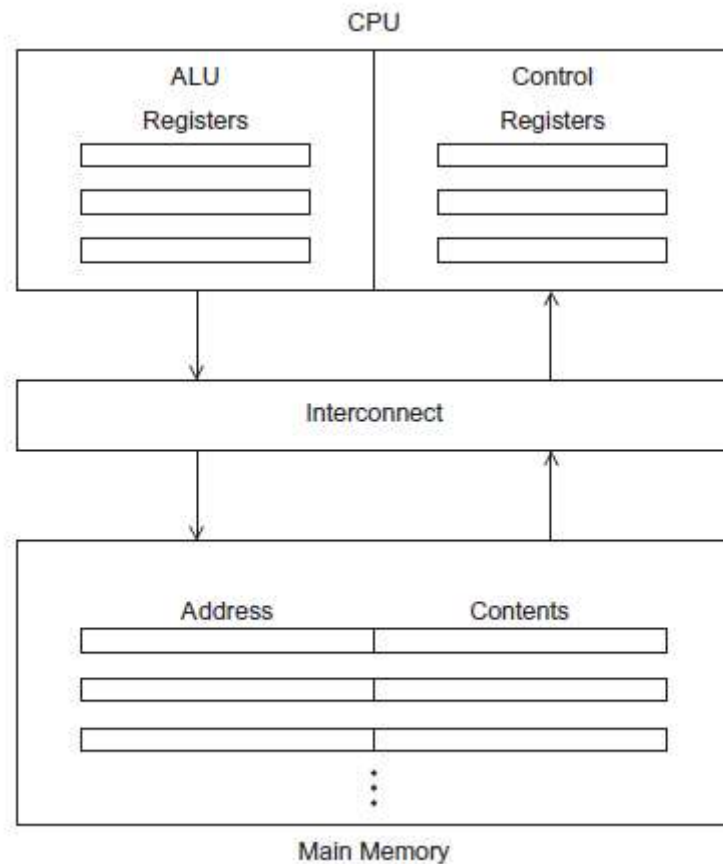
Big Data processing!



Past: Serial hardware and software



Past : Computer architecture: The von Neumann Architecture



Increasing single processor performance

- From 1986 – 2002, microprocessors were increasing in performance an average of 50% per year
 - by increasing density of transistors.



- Since then, it's dropped to about 20% increase per year..

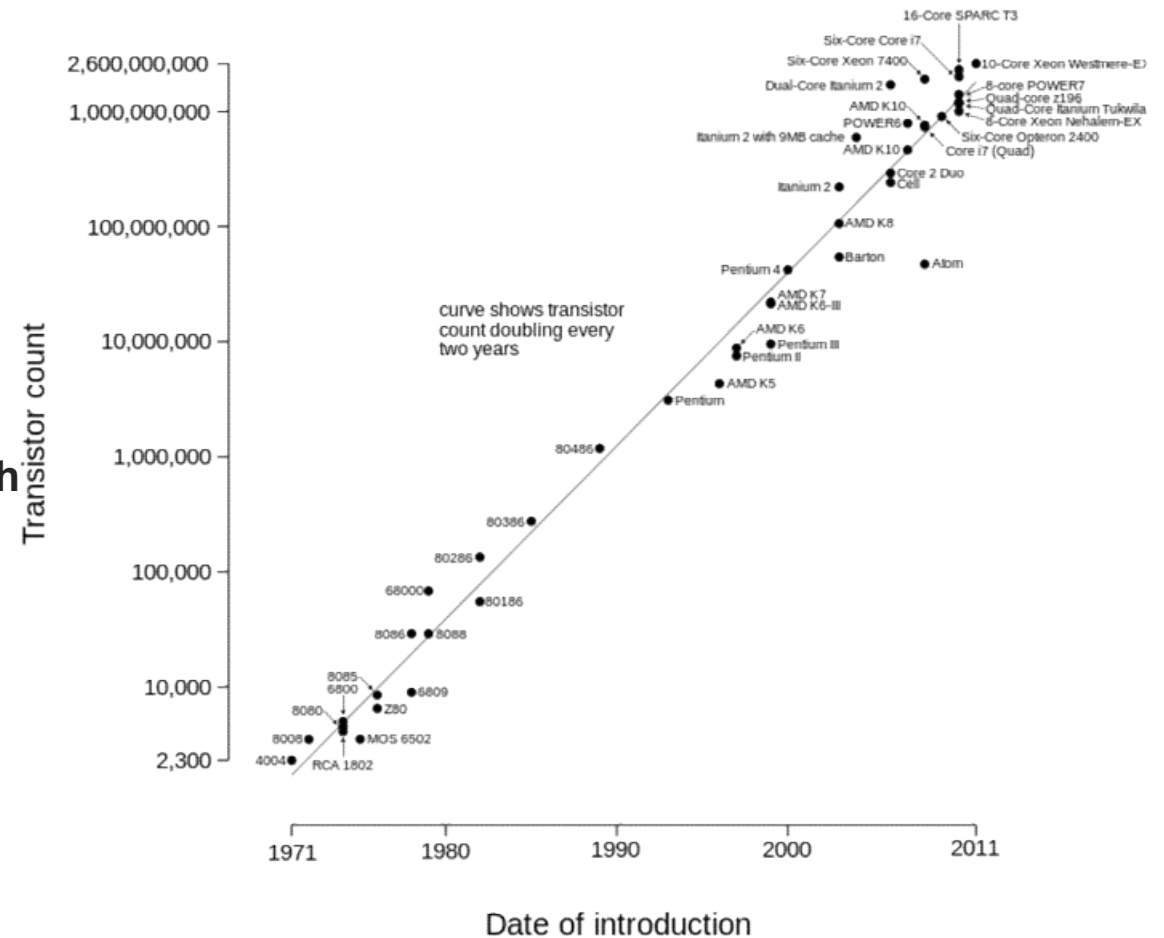


Increasing density of transistors

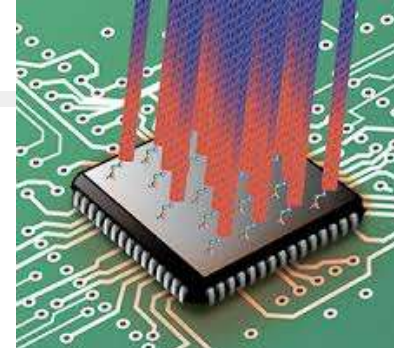


circuit board with a 4×4 array of SyNAPSE-developed chips, **each** chip using **5.4 billion** transistors.

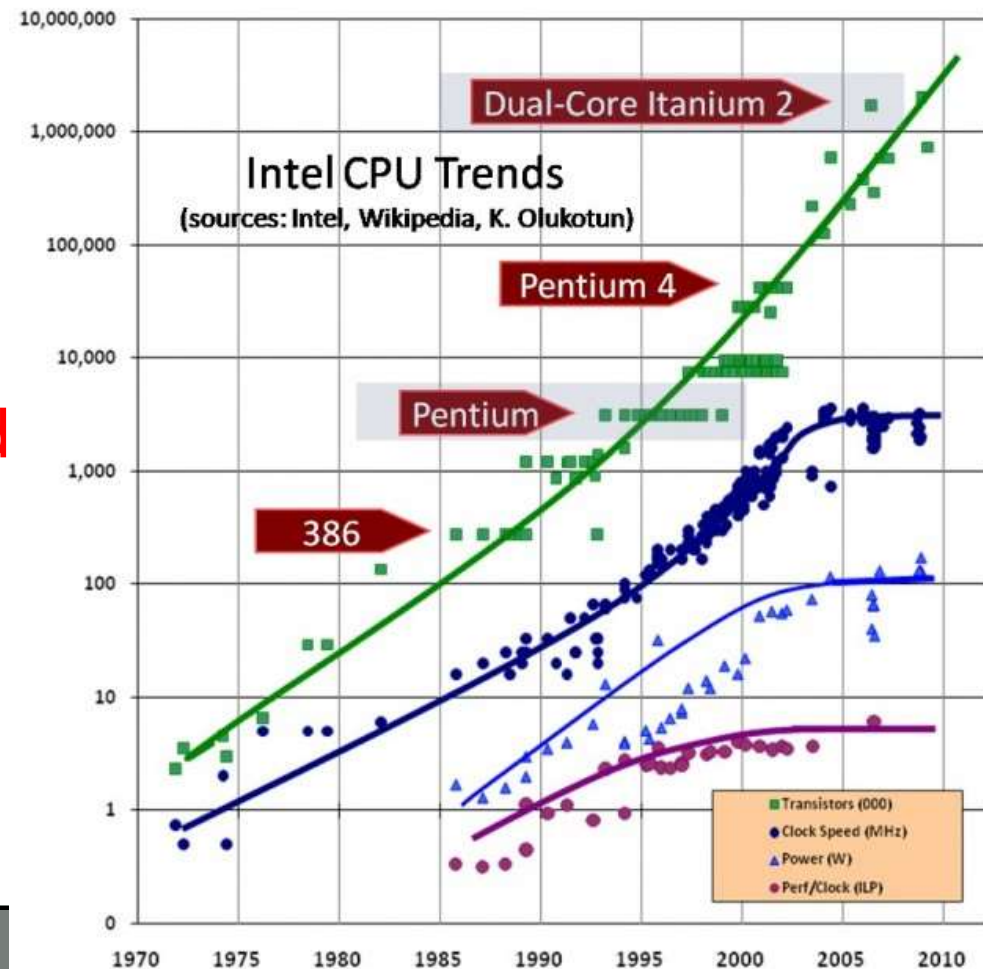
Microprocessor Transistor Counts 1971-2011 & Moore's Law



Heating Problems..



- Smaller transistors = faster processors.
- Faster processors = increased power consumption.
- Increased power consumption = increased heat.
- Increased heat = unreliable processors.



Solution

- ➔ Use **multicore** processors (**CPUs**) on a single chip
- ➔ called **cores**



An Intel Core 2 Duo E6750 dual-core processor.

- ➔ Introducing parallelism!!!

Now it's up to the programmers

- Adding more processors doesn't help much if programmers aren't aware of them...
- ... or don't know how to use them.
- Serial programs don't benefit from this approach (in most cases).



How to write parallel programs

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to run faster.



Approaches to the serial problem

- Rewrite serial programs so that they're parallel.
- Write translation programs that automatically convert serial programs into parallel programs.
 - This is very difficult to do.
 - Success has been limited.


Example

- Compute n values and add them together.
- Serial solution:


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

Example (cont.)

- We have p cores, p much smaller than n .
- Each core performs a partial sum of approximately n/p values.



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



Each core uses its own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, a **private** variable `my_sum` contains the sum of the values computed by its calls to `Compute_next_value`.
- Ex., 8 cores, $n = 24$, then the calls to `Compute_next_value` return:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Example (cont.)

- Once all the cores are done computing their private `my_sum`,
 - they form a **global sum** by **sending** results to a designated “**master**” core
 - which **adds** the final result...

Example (cont.)

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

Example (cont.)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

$$8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$$

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

But wait!

There's a much better way
to compute the global sum.



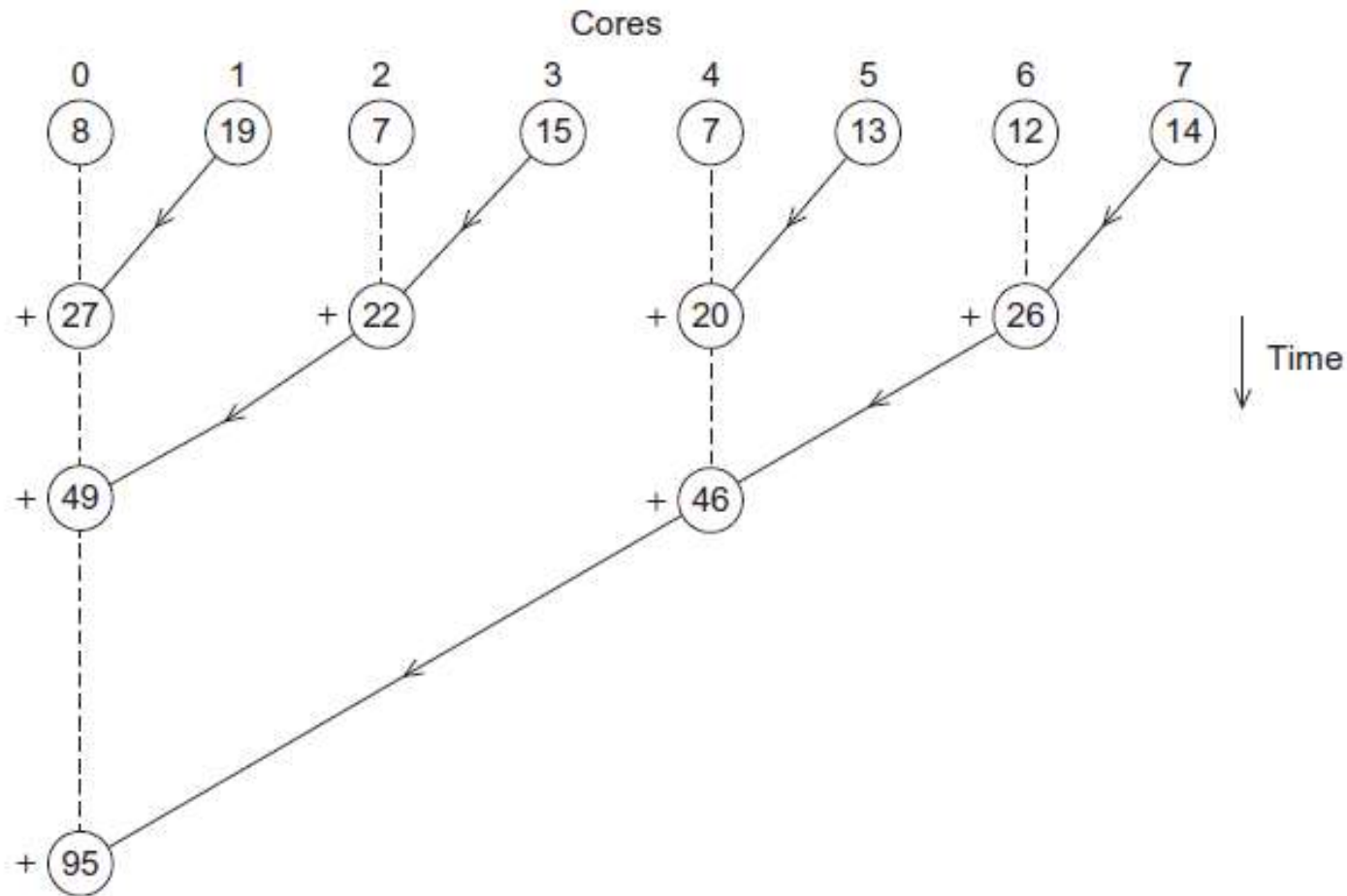
Better parallel algorithm

- Don't make the master core do all the work.
- Share it among the other cores.
- Pair the cores so that core 0 adds its result with core 1's result.
- Core 2 adds its result with core 3's result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.
 - Core 0 adds result from core 2.
 - Core 4 adds the result from core 6, etc.
- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2!

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
 - The first example would require the master to perform 999 receives and 999 additions.
 - The second example would only require 10 receives and 10 additions.
- That's an improvement of almost a factor of 100!

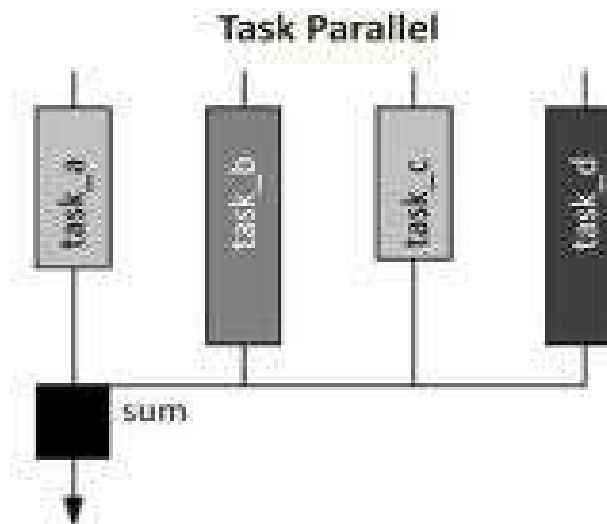
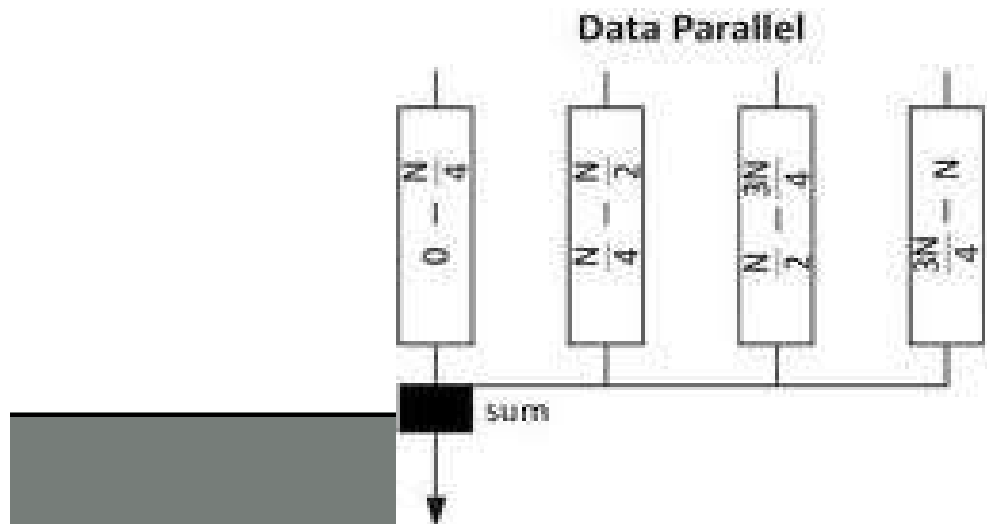
How do we write parallel programs?

- Data parallelism

- Partition the **data** used in solving the problem among the cores.
- Each core carries out similar operations on its **part of the data**.

- Task parallelism

- Partition various **tasks** carried out solving the problem among the cores.

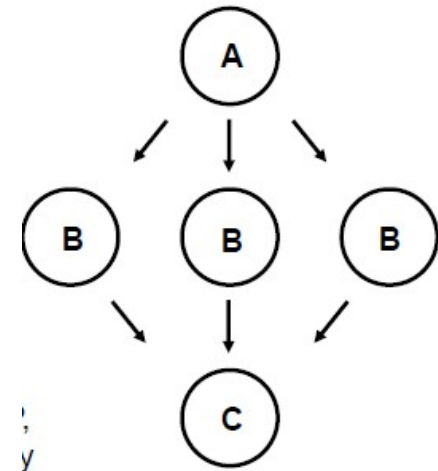


Data Parallelism

- Definition: each process does the same work on **unique** and independent pieces of **data**

- Examples:

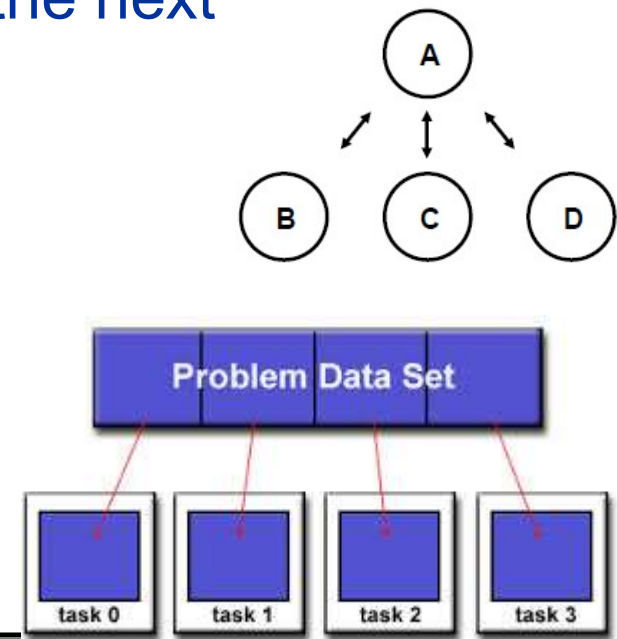
- 8 farmers mow a lawn
- 2 farmers paint a storage area



- Usually more **scalable** than functional parallelism
- Can be programmed at a high level with **OpenMP**

Task Parallelism

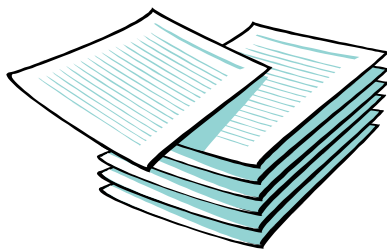
- Definition: each processor executes a **different process/thread** (same or different code) on the same or different data.
- Usually each processor executes a different process or an independent program
- **Processes communicate** with one another as they work by passing data from one process/thread to the next
- More suitable for distributed computation
- **Examples:**
 - Independent Monte Carlo Simulations
 - ATM Transactions



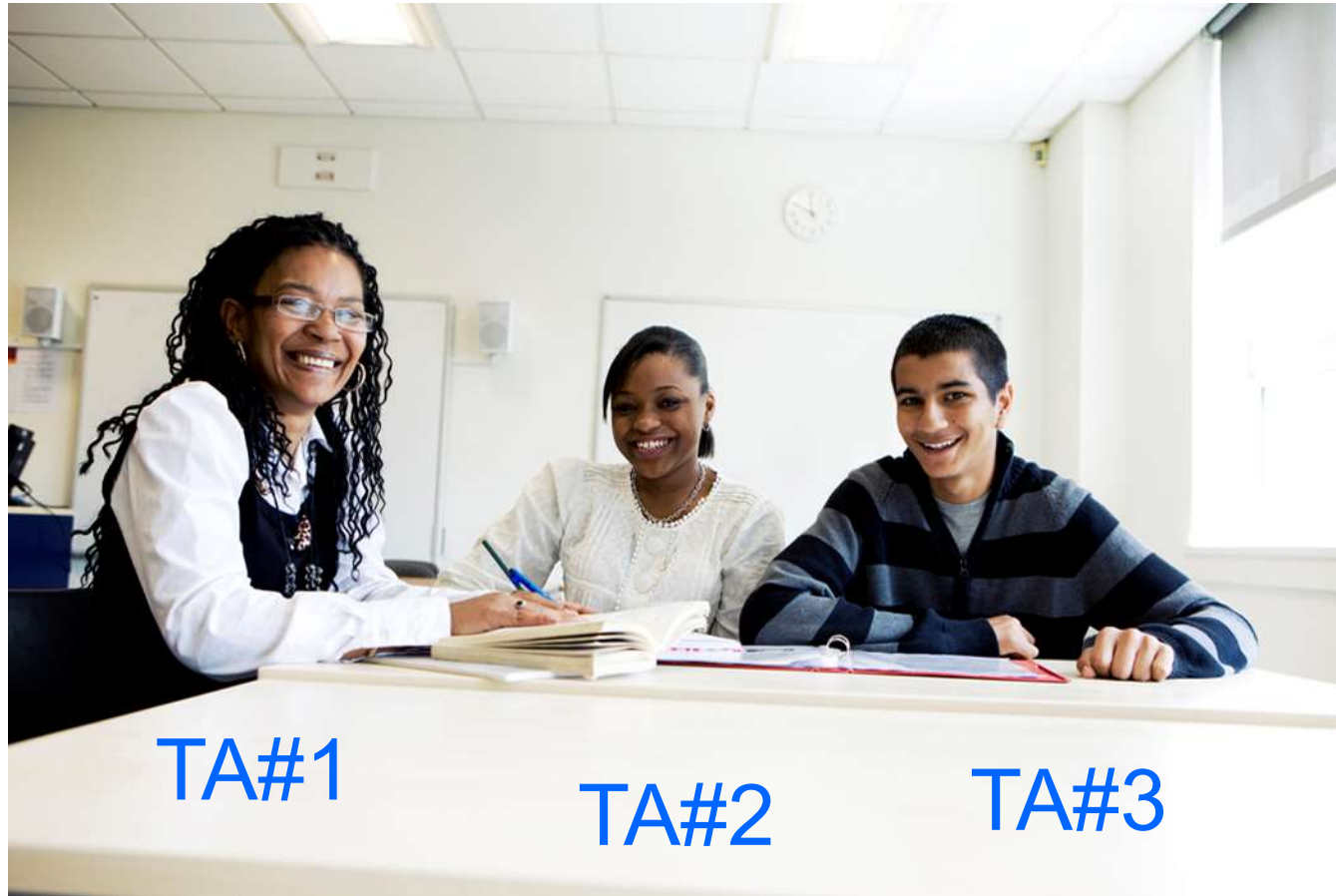
- Example follows..

Professor P

15 questions
300 exams

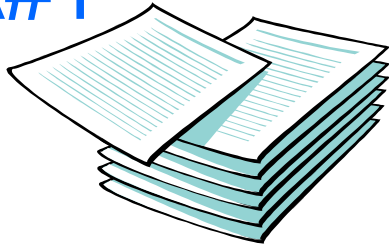


Professor P's grading assistants

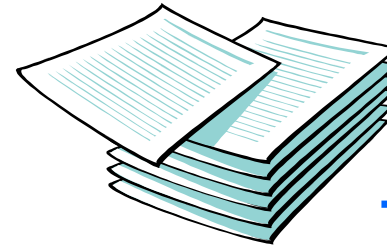


Division of work – data parallelism

TA#1

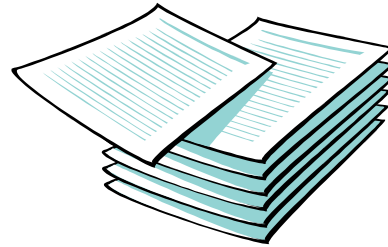


100 exams



100 exams

TA#3

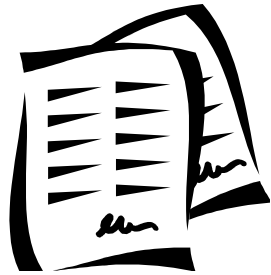


100 exams

TA#2

Division of work – task parallelism

TA#1



Questions 1 - 5



TA#3

Questions 11 - 15



Questions 6 - 10

TA#2

Division of work –data parallelism

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

Division of work –task parallelism

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

Tasks

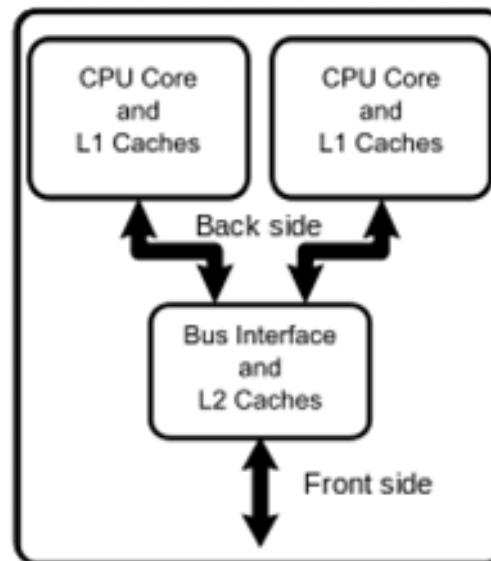
- 1) Receiving
- 2) Addition

Terminology (1/3)

- **Serial** code is a single thread of execution working on a single data item at any one time
- **Parallel** code has more than one thing happening at a time. This could be
 - Multiple **threads** of execution in a single executable on different data
 - Multiple **executables (processes)** all working on the same problem (in on or more programs)
 - Any combination of the above
- **Task** is a program or a function.
 - Each task has its own virtual address space and may have multiple threads

Terminology (2/3)

- Traditional CPU: a single central processing unit (CPU) on a chip.
- Multi-core processor/ socket : A single chip containing two or more CPUs called "cores"
- A node may have multiple cores...



Terminology (3/3)

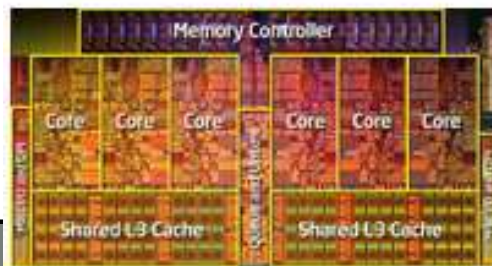
- **Node:** a actual physical self contained computer unit that has its own processors, memory, I/O bus and storage.



Supercomputer - each blue light is a node

Node - standalone
Von Neumann computer

CPU / Processor / Socket - each
has multiple cores / processors.



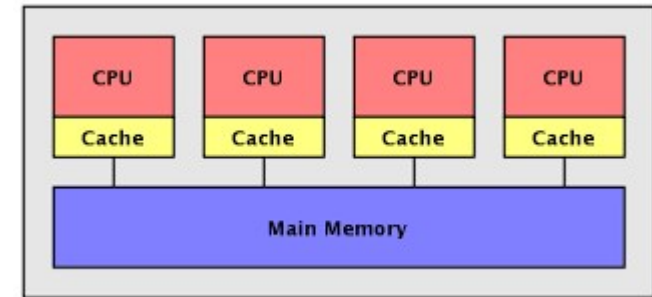
Coordination

- Cores usually need to coordinate their work.
- **Communication** – one or more cores send their current partial sums to another core.
- **Load balancing** – share the work evenly among the cores so that one is not heavily loaded.
- **Synchronization** – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

Type of parallel systems

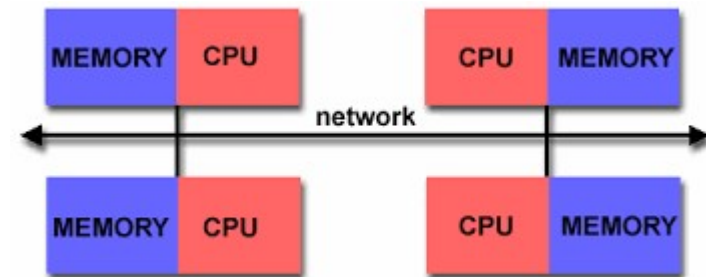
■ Shared-memory

- The cores can **share** access to the computer's memory.
- Coordinate the cores by having them **examine and update shared memory locations**.

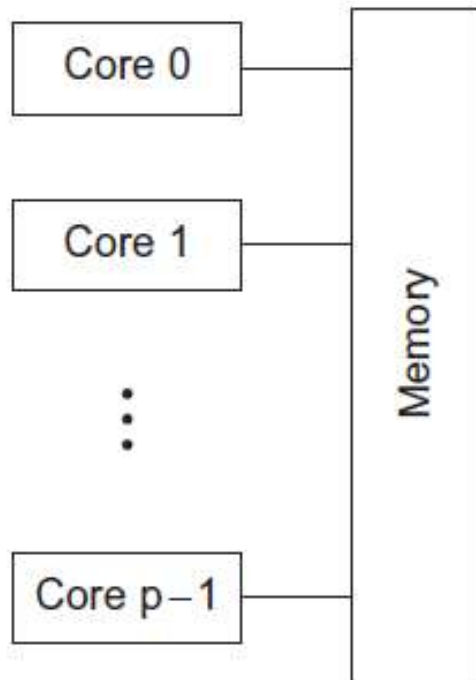


■ Distributed-memory

- Each core has its **own**, private memory.
- The cores must **communicate** explicitly by sending messages across a network.

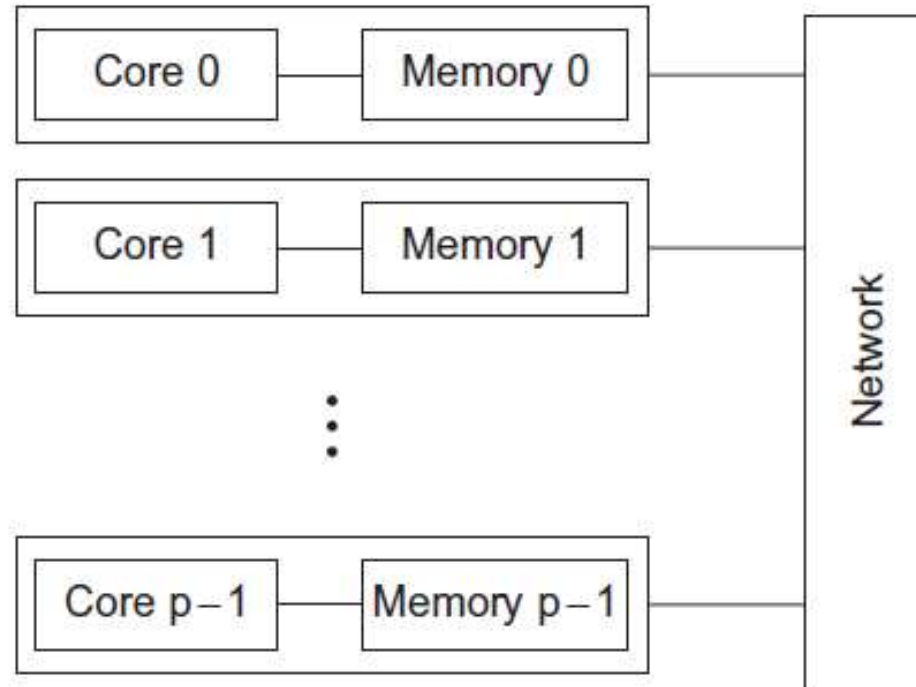


Type of parallel systems



(a)

Shared-memory



(b)

Distributed-memory

Terminology

- **Parallel computing** – a single program in which multiple tasks *cooperate closely* to solve a problem
- **Distributed computing** – many programs cooperate with each other to solve a problem.

Time for a Break?

Or for some comments/ discussion up to now?!....

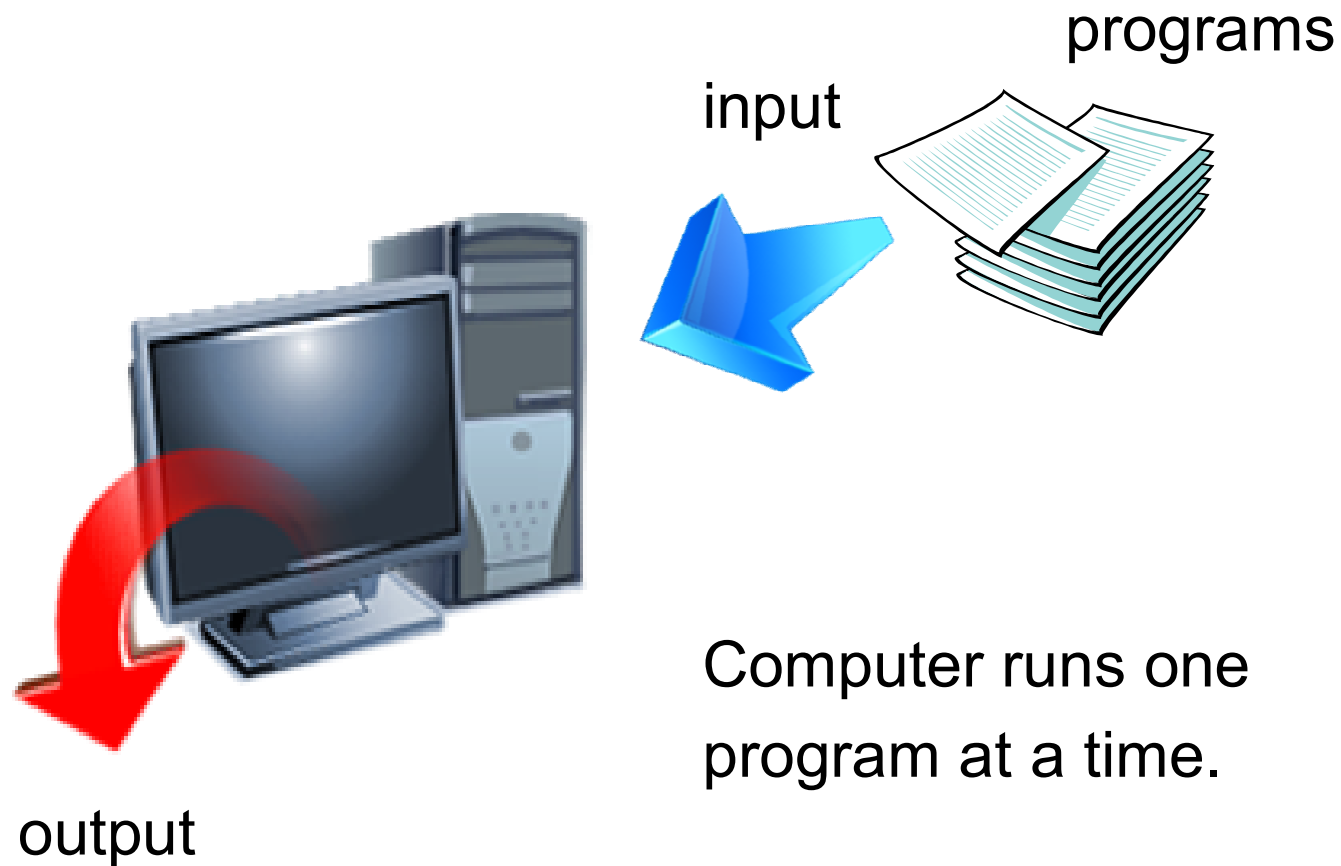




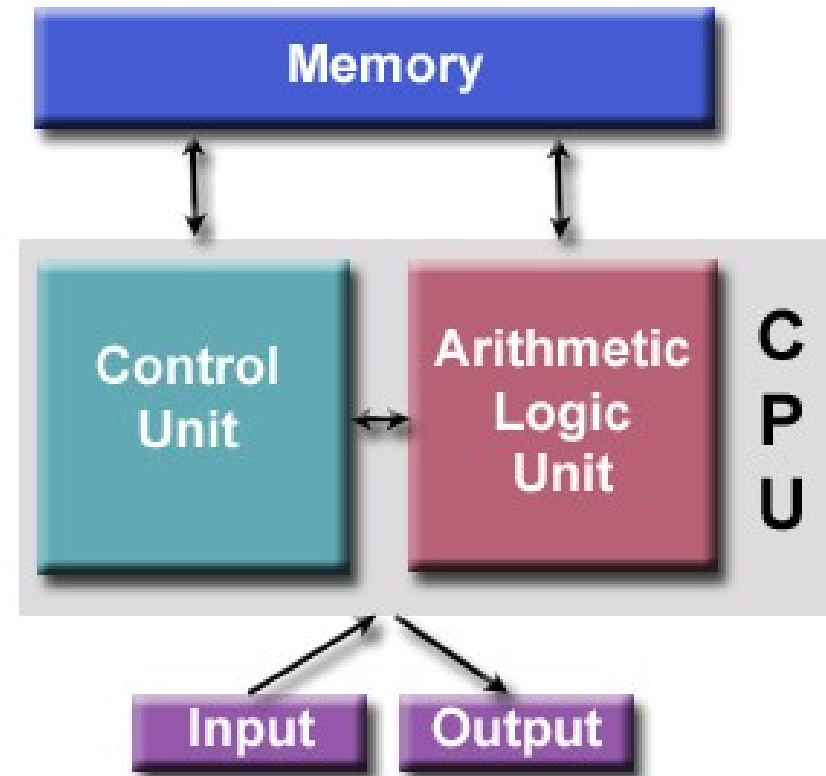
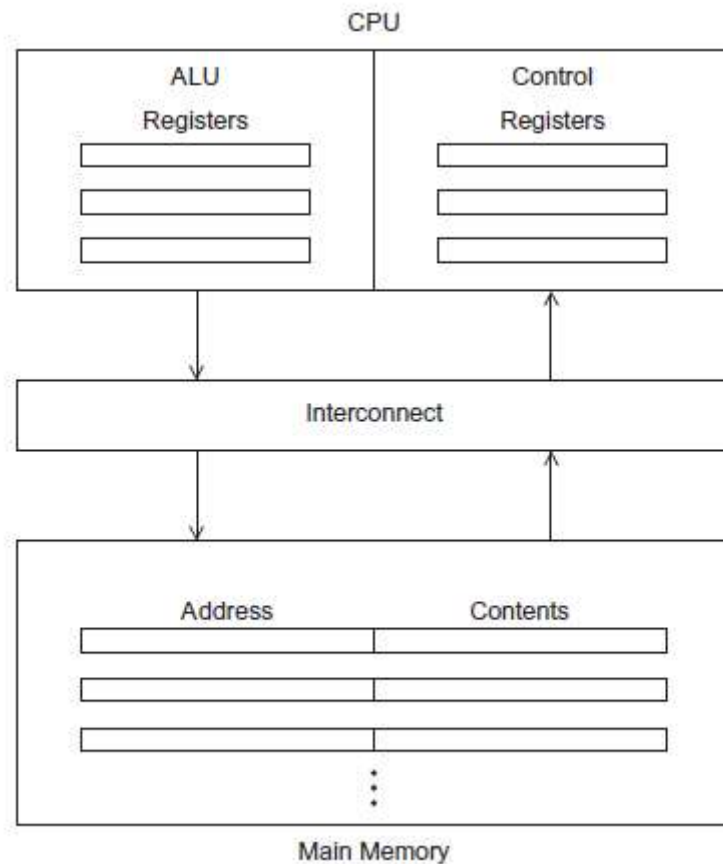
Part 2

Parallel Hardware and Parallel Software

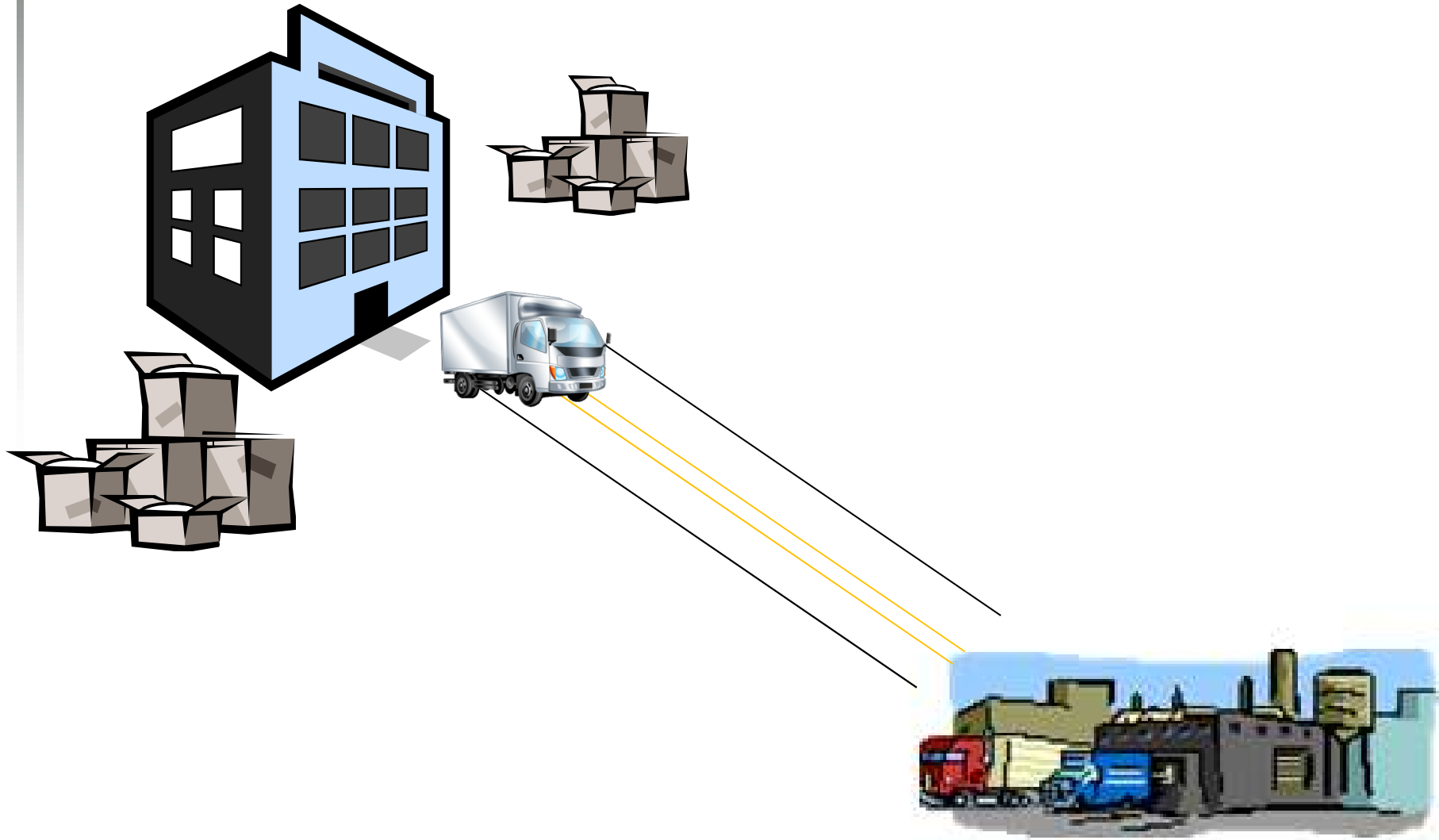
Past: Serial hardware and software

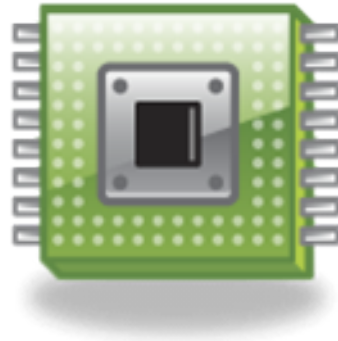


The von Neumann Architecture

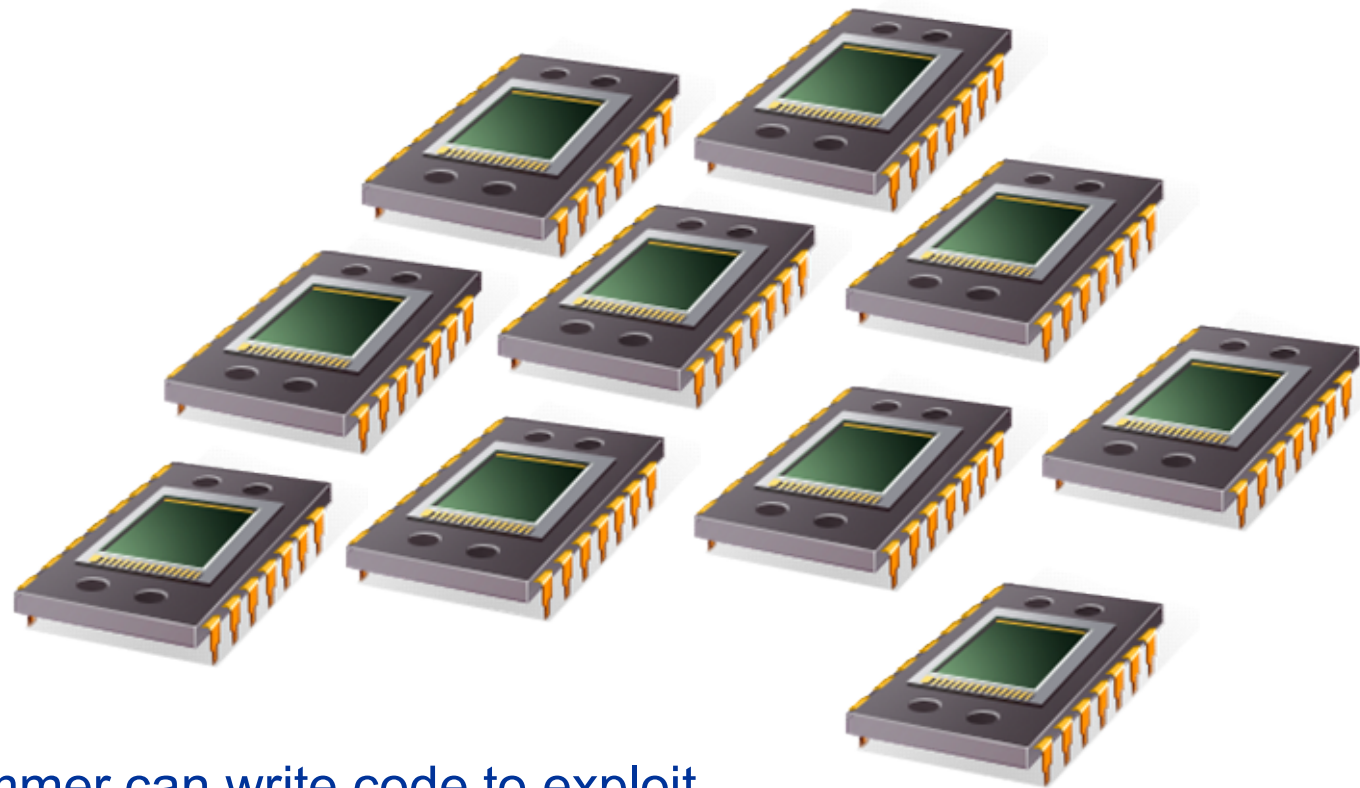


von Neumann bottleneck





MODIFICATIONS TO THE VON NEUMANN MODEL



A programmer can write code to exploit.

PARALLEL HARDWARE

Flynn's Taxonomy

classic von Neumann

<p>SISD</p> <p>Single instruction stream Single data stream</p>	<p>(SIMD)</p> <p>Single instruction stream Multiple data stream</p>
<p>MISD</p> <p>Multiple instruction stream Single data stream</p>	<p>(MIMD)</p> <p>Multiple instruction stream Multiple data stream</p>

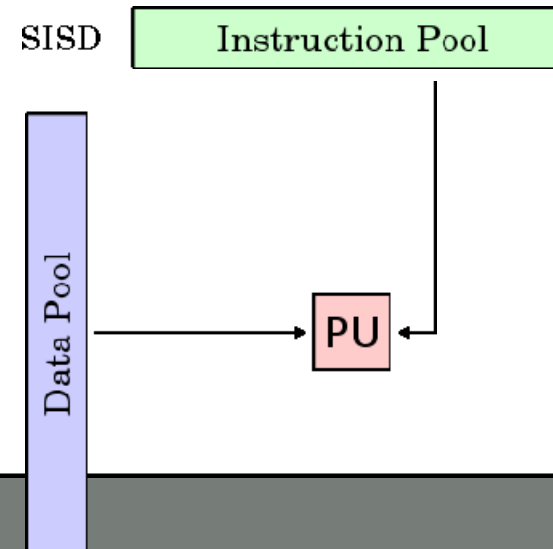
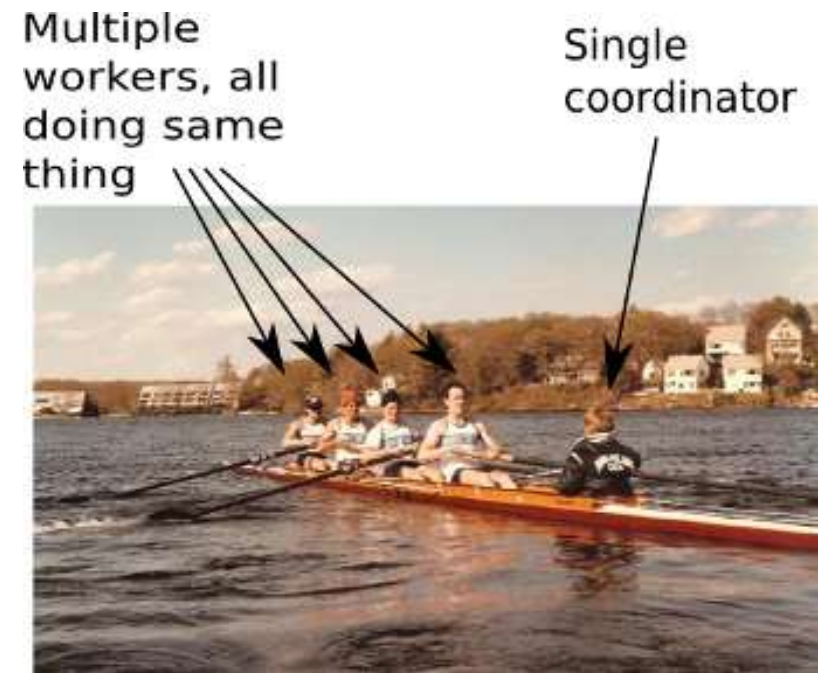
Flynn's Parallel Architecture Taxonomy (1966)

- **SISD**: single instruction single data – **traditional** serial processing!
- **MISD**: **rare** – multiple instructions on a single data item-
e.g., for fault tolerance!
- **SIMD**: single instruction on multiple data!
 - Some old architectures with a resurgence in accelerators!
 - **Vector processors** - pipelining!
- **MIMD**: **multiple instructions multiple data** - almost all **parallel computers !**

- [https://computing.llnl.gov/tutorials/parallel_comp/
#Terminology](https://computing.llnl.gov/tutorials/parallel_comp/#Terminology)

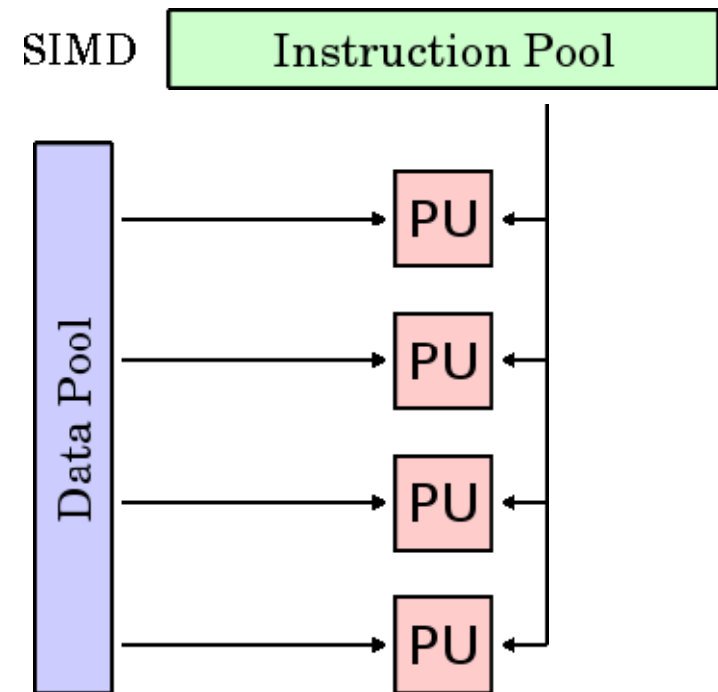
Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle

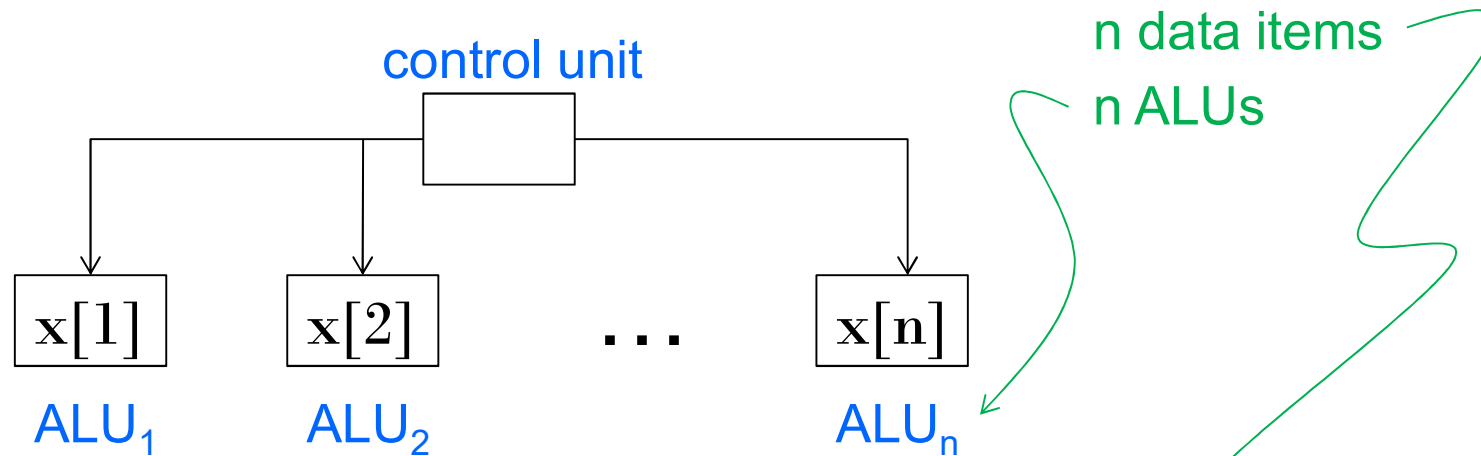


SIMD (Single Instruction Multiple Data)

- Parallelism achieved by dividing data among the processors.
Applies the **same instruction** to **multiple data** items.
- Called **data parallelism**.



SIMD example



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

SIMD

- 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.

Round	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

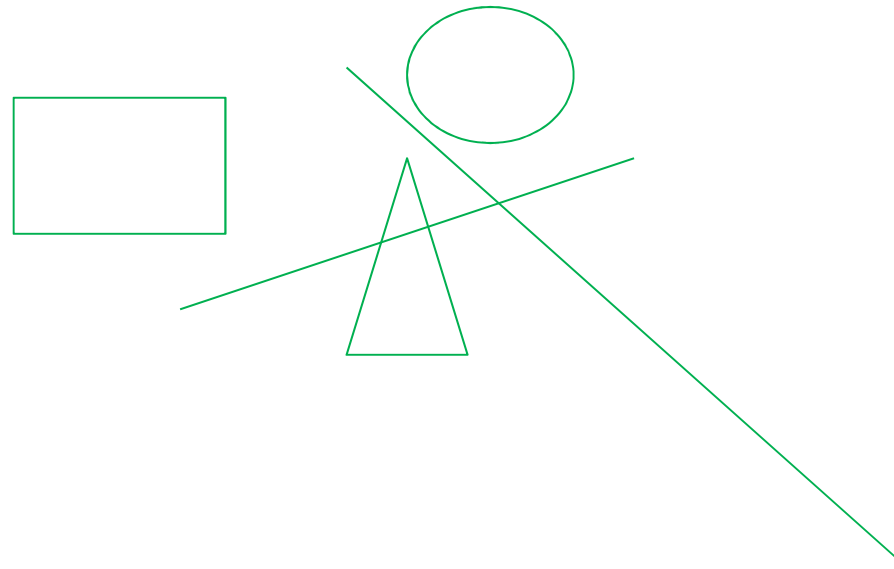
- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate **synchronously**.
- The ALUs have **no instruction storage**.
- **Efficient for large data parallel problems, but not other types of more complex parallel problems.**

SIMD example: Vector processors

- Operate on arrays or vectors of data
- Advantages:
 - Fast, Easy to use
 - Good vectoring compilers
 - High memory bandwidth.
- Disadvantages:
 - don't handle **irregular** data structures
 - Scalability

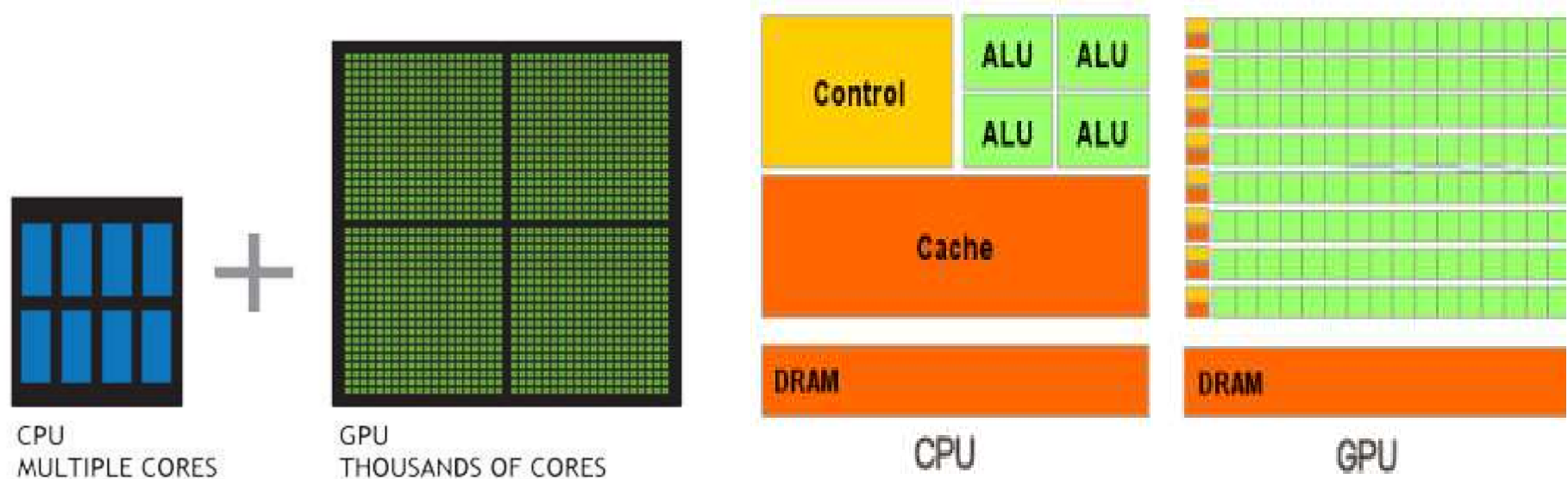
Not pure SIMD: Graphics Processing Units (GPU)

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



Graphics Processing Unit (GPU)

- A specialized electronic circuit with thousands of cores that are specifically designed to perform data-parallel computation
- It process multiple elements in the graphics stream.



GPUs: how they work

- Can rapidly manipulate and alter **memory** to accelerate the creation of **images** for output to a **display**
- Each **pixel** is **processed** by a **short program** before it was projected onto the screen

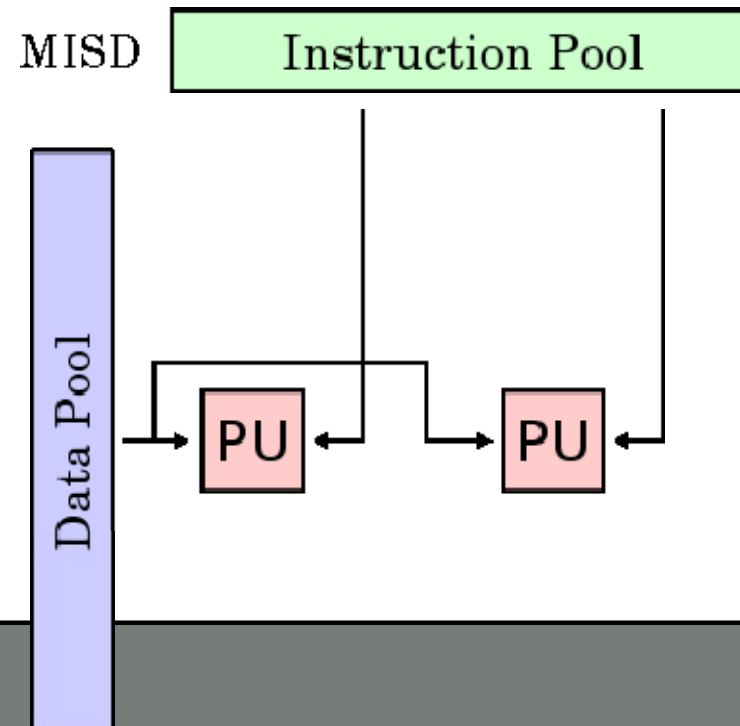
Other Applications

- GPUs are used as **vector processors** for **non-graphics** applications that require repetitive computations.



Multiple Instruction, Single Data (MISD):

- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- **Few** (if any) actual examples of this class of parallel computer have ever existed.



MIMD (multiple Instructions Multiple Data)

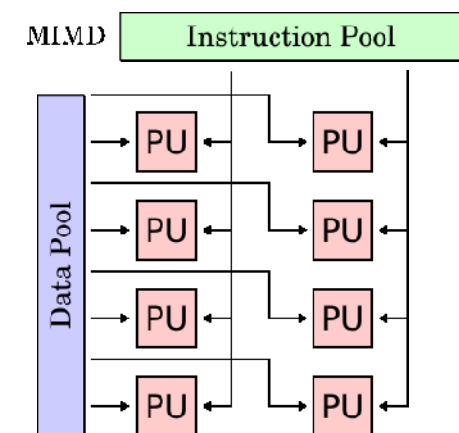
- **Multiple Instruction:** Every processor may be executing a different instruction
- **Multiple Data:** Every processor may be working with a different data
- Typically consist of a collection of **fully independent processing units or cores**, each of which has **its own control unit** and **its own ALU**.
- MIMD Architecture types:
 - **Shared memory**
 - **Distributed memory**
 - **Hybrid!**



Workers with same objective, doing completely different things

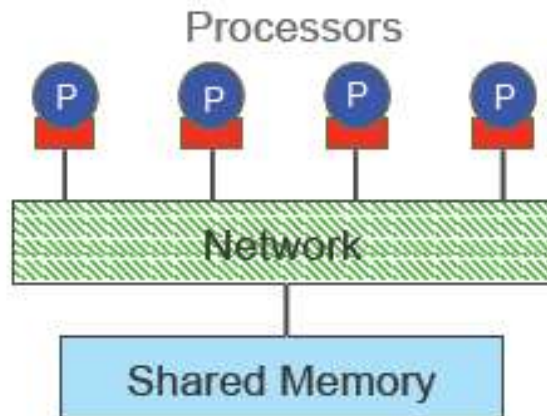


Cray XT3



Shared Memory System (1/2)

- Multiple processing units accessing global shared memory using a single address space



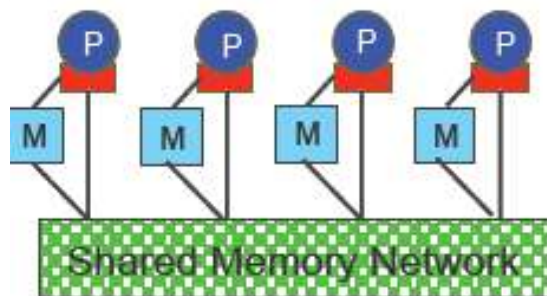
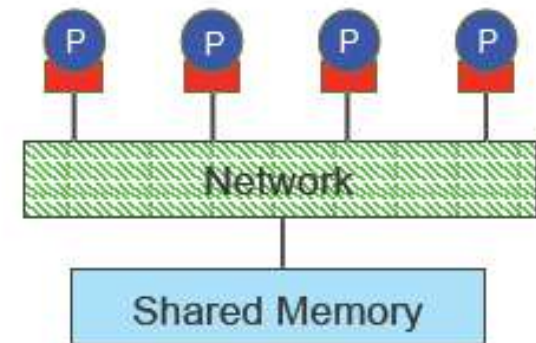
- Shared memory systems are easier to program
 - User responsible for synchronization of processors for correct data access and modification
- Scaling to large number of processors can be an issue

Share Memory Access (2/2)

Two types of shared memory systems based on access type:

UMA: Uniform Memory Access – all memory is “equidistant” from all processors

- Memory access can become a bottleneck

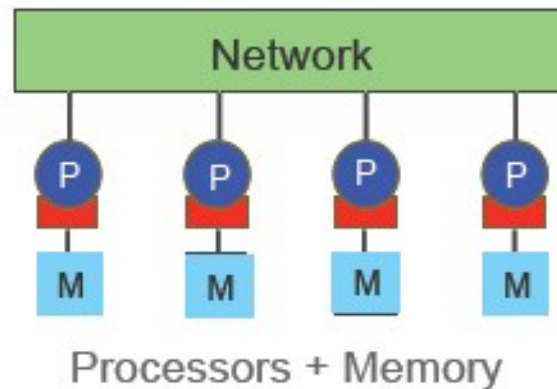


NUMA: Non-Uniform Memory Access – local memory versus distant memory

- Requires more complex interconnect hardware to support global shared memory
- Also called *Distributed shared memory systems*

Distributed Memory System

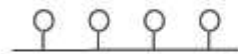
- Multiple processing units with independent local memory and address spaces



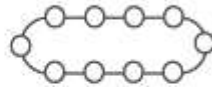
- Systems are easier to scale
- No implicit sharing of data – user is responsible for explicit communication of data amongst processors

Interconnection Networks

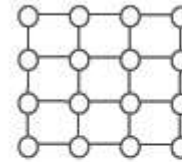
- Topologies:



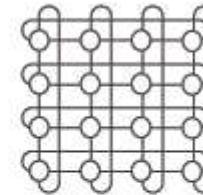
Bus



Ring



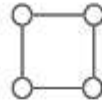
Mesh



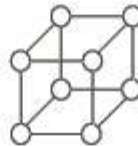
Toroidal Mesh



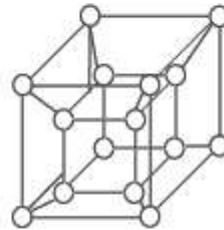
1d



2d

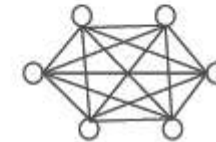


3d



4d

Hypercube



Fully Connected

- Network characteristics:

- Latency (l): time it takes for a link to transmit a unit of data (sec)
- Bandwidth (b): rate at which data is transmitted (bytes/sec)
- Message transmission time for n bytes = $l + n/b$
- Bisection (band)width: a measure of network quality – number of links connecting two halves of a network



PARALLEL SOFTWARE

The burden is on software

- In shared memory programs:
 - Start a single process and fork threads.
 - Threads carry out tasks.
 - Share a common memory space
- In distributed memory programs:
 - Start multiple processes.
 - Processes carry out tasks.
 - No shared memory space
 - Data is communicated between each other via message passing

Approaches to Parallel Programming

- Shared memory programming: assumes a global address space – data visible to all processes.
 - Issue: synchronizing updates of shared data
 - Software Tool: OpenMP
- Distributed memory programming: assumes distributed address spaces – each process sees only its local data.
 - Issue: communication of data to other processes
 - Software Tool : MPI (Message Passing Interface)

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.

2. Arrange for the processes/threads to synchronize.

3. Arrange for communication among processes/threads.

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

Shared Memory: Nondeterminism

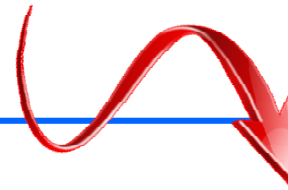
```
...  
printf ( "Thread %d > my_val = %d\n" ,  
        my_rank , my_x ) ;  
...
```

Private variable



Thread 1 > my_val = 19

Thread 0 > my_val = 7



Thread 0 > my_val = 7

Thread 1 > my_val = 19

Shared Memory: Nondeterminism

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

```
x += my_val ;
```

WRONG RESULT!

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

Shared Memory: Nondeterminism


- 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 ) ;
```

Shared Memory: busy-waiting


```
my_val = Compute_val ( my_rank ) ;  
i f ( my_rank == 1)  
    w h i l e ( ! o k _ f o r _ 1 ) ; /* Busy-wait loop */  
x += my_val ; /* Critical section */  
i f ( my_rank == 0)  
    o k _ f o r _ 1 = true ; /* Let thread 1 update x */
```

Shared variable



Distributed Memory: message-passing

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



Local variable

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*.



PERFORMANCE

Speedup

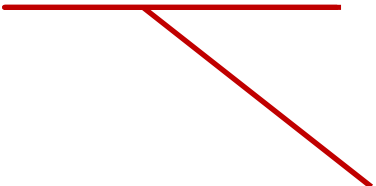
- Number of cores = p
- Serial run-time = T_{serial}
- Parallel run-time = T_{parallel}



linear speedup
(optimal parallel time)

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

Speedup of a parallel program


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

- Speedup is at most p .

Efficiency of a parallel program

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

➤ Efficiency is at most 1.

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

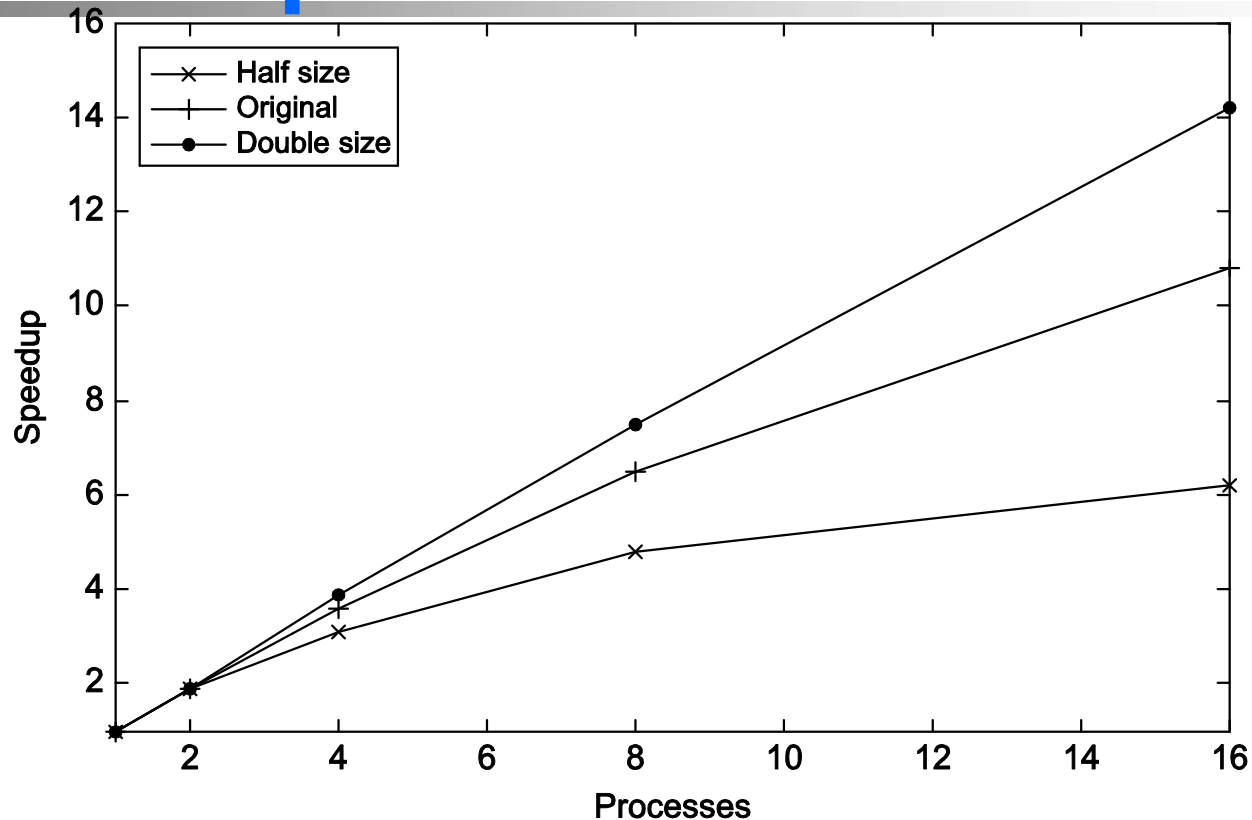
- Efficiency decreases as processors increase..

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

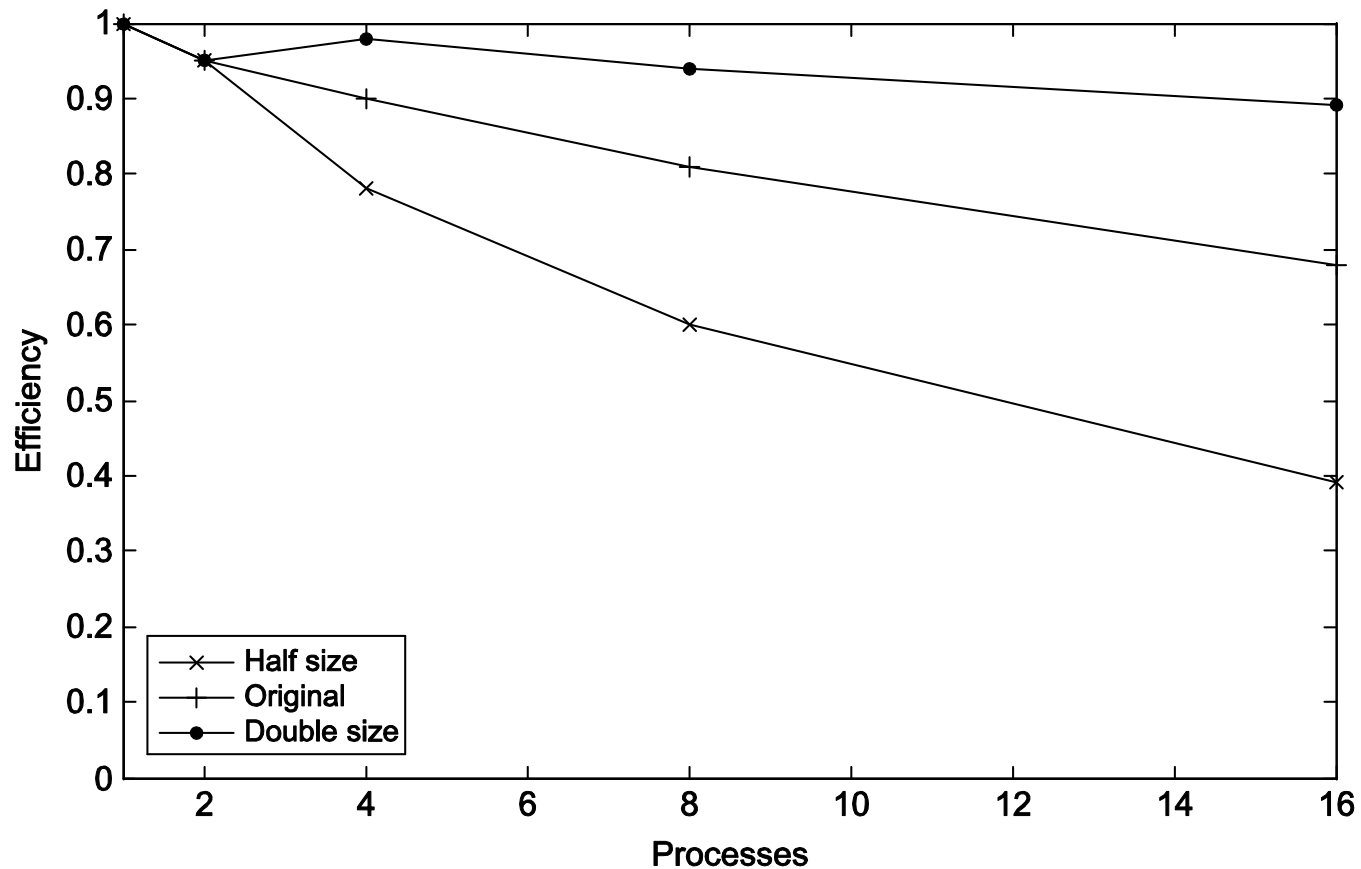
- For a fixed p , Efficiency **increases** as the size of the problem **increases**..

Speedup



- Speedup **increases** as the size of the problem **increases**..
- Speedup **increase decreases** as processors **increase**..

Efficiency



- Efficiency **decreases** as the processors increase more when the size of the problem **decreases**..

Effect of overhead

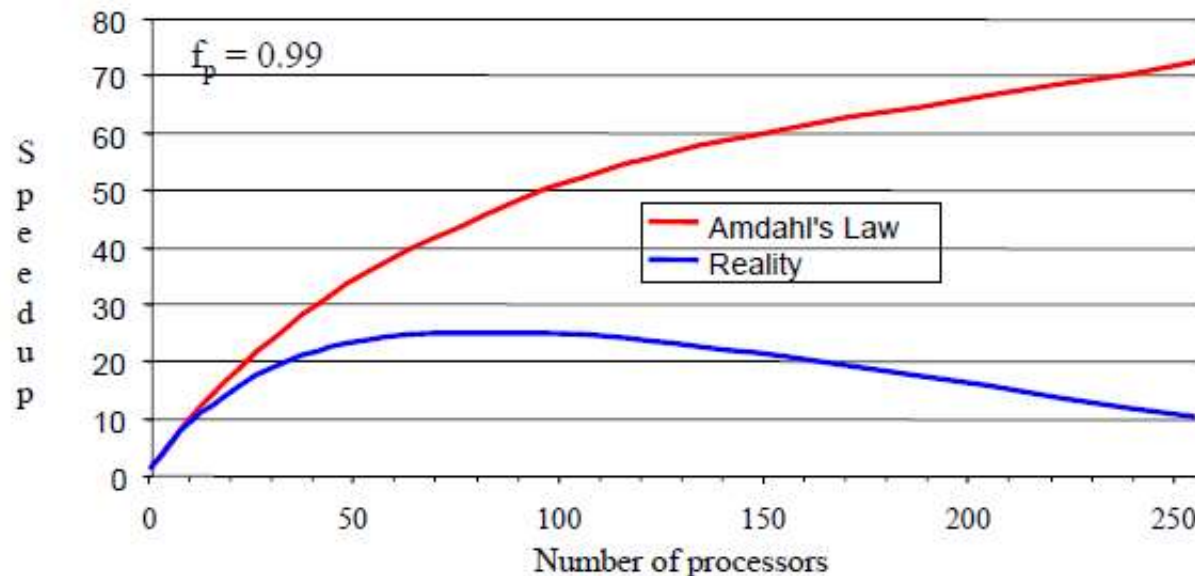
$$T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}}$$

Amdahl's Law

- Amdahl's Law shows a theoretical upper limit for speedup

In reality, the situation is even worse than predicted by Amdahl's Law due to:

- Load balancing (waiting)
- Scheduling (shared processors or memory)
- Communications
- I/O



Example

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” regardless of the number of cores p we use.
- $T_{\text{serial}} = 20$ seconds
- Runtime of parallelizable part is

$$0.9 \times T_{\text{serial}} / p = 18 / p$$

Example (cont.)

- Runtime of “unparallelizable” part is

$$0.1 \times T_{\text{serial}} = 2$$

- Overall parallel run-time is

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}} = 18 / p + 2$$

Example (cont.)

- Speed up

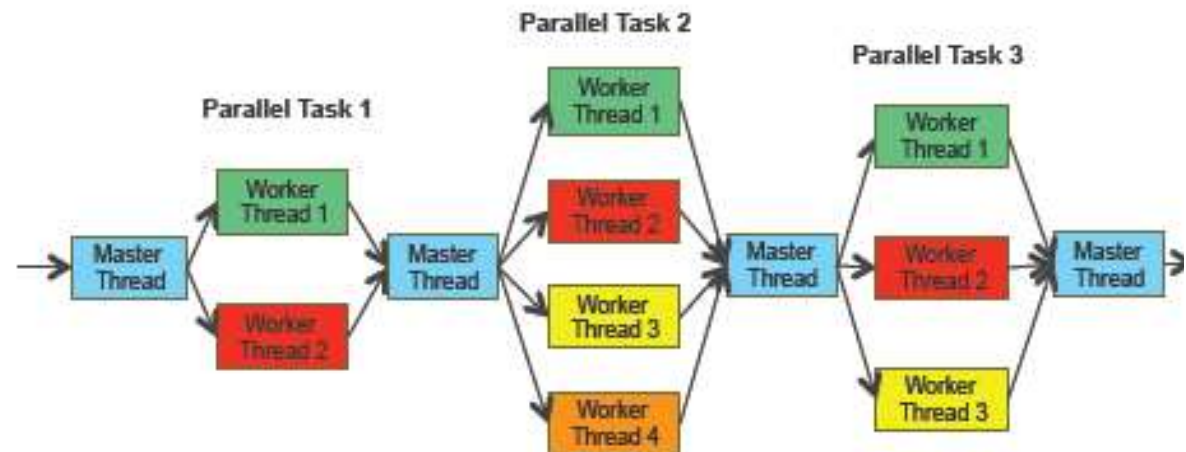
$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}}} = \frac{20}{18 / p + 2}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- 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*.

Shared Memory: OpenMP

- *OpenMP*: a standard API to support shared memory parallel programming
 - Managed by the OpenMP Architecture Review Board, OpenMP v1.0 was released in 1997, latest v3.1 released July 2011
- A directive-based approach to control:
 - *Parallel threads*: Master thread creates parallel worker threads and the work is divided amongst the workers
 - *Data sharing*: assumed a global address space
- Major components:
 - Parallel control structure
 - Work sharing
 - Data sharing and control
 - Synchronization
 - Other runtime functions



Example: Sum of Squares

```
...  
long sum = 0, loc_sum;  
int thread_id;
```

Forks off the threads and starts the work-sharing construct; declares *thread_id* and *loc_sum* private

```
#pragma omp parallel private(thread_id, loc_sum)
```

```
{
```

Each thread retrieves its own id

```
loc_sum = 0;
```

```
thread_id = omp_get_thread_num();
```

Parallel for splits loop range across the threads.

```
#pragma omp for
```

```
for(i = 0; i < N; i++) loc_sum = loc_sum + i * i;
```

Each thread computes and prints its id and local sum

```
printf("\n Thread %i: %li\n", thread_id, loc_sum);
```

```
#pragma omp critical
```

Threads cooperate to update global variable one by one

```
sum = sum + loc_sum;
```

```
}
```

```
printf("\n Sum of Squares = %li", sum);
```

Master thread prints result

OpenMP comments

✓ Plus:

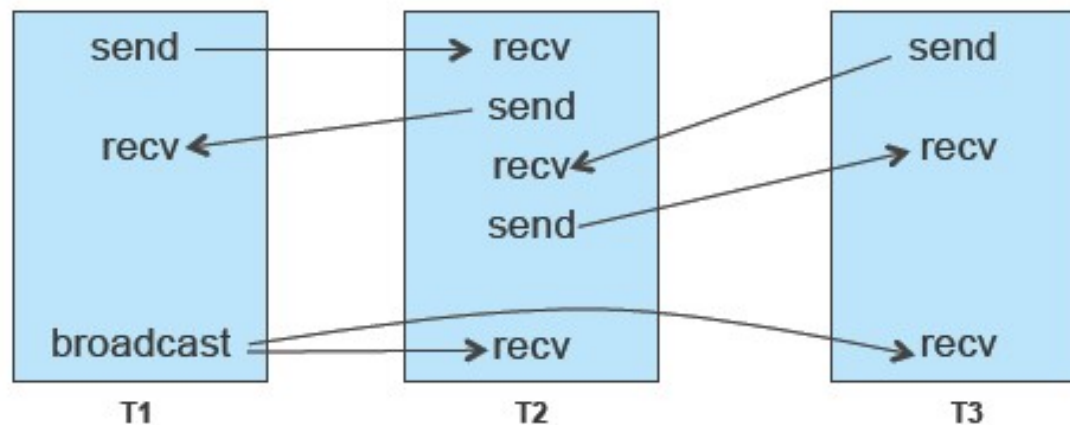
- Very simple to use
- High level parallel directives
- Not getting in low-level (thread) programming

↓ Drawbacks

- You can not use more than one cores using OpenMP
 - (due to the shared memory between threads)
 - To do so use need to use hybrid (openMP/MPI) parallel programming

Distributed Memory: MPI

- MPI (Message Passing Interface): a standard *message passing* library specification to support process communication on a variety of systems
 - MPI v1.0 (June 1994), latest MPI v2.2 (Sept 2009)
- MPI assumes a distributed address space, i.e., each process (rank) sees only local variables with explicit constructs to communicate data to other processes



MPI Features

- MPI-1
 - General: Init/finalize, Communication group size/rank
 - Point to Point communication:
 - send, recv with multiple modes (blocking/non blocking, ...)
 - Collective communication:
 - Barrier for synchronization
 - Broadcast
 - Gather/scatter
 - Reduction operations (built-in and user defined)
- MPI-2
 - One-sided communication: Put, Get, Accumulate
 - Extensions to collectives
 - Dynamic process management

MPI code: sum of squares

```
...  
int num_tasks, my_rank, rc;  
int sum, loc_sum, N = ...;
```

Number of
processes

```
MPI_Init();
```

```
MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Each process
retrieves its rank

Each process computes
and prints its rank and
local sum

```
for(i = 0; i < N; i += numtasks) loc_sum = loc_sum + i * i;  
printf("\n Thread %i: %li\n", my_rank, loc_sum);
```

Each process sends
local sum to rank 0

```
if (my_rank != 0)
```

```
    rc = MPI_Send(loc_sum, 1, MPI_INTEGER, 0, my_rank, ... );
```

```
else {
```

```
    sum = loc_sum
```

```
    for (i = 1; i < num_tasks; i++) {
```

```
        rc = MPI_Recv(&loc_sum, 1, MPI_INTEGER, i, i, ... );
```

```
        sum = sum + loc_sum
```

```
    }
```

```
    printf("\n Sum %i: %li\n", my_rank, loc_sum);
```

```
}
```

Rank 0 receives
data from all other
ranks, computes
and prints result

```
MPI_Finalize();
```

Example: Pi Calculation with openMP

Serial:

```
1  x=0;
2  sum = 0.0;
3  step = 1.0/(double) num_steps;
4  for (i=0; i < num_steps; ++i) {
5      x = (i+0.5)*step;
6      sum = sum + 4.0/(1.0+x*x);
7  }
8  pi = step * sum;
```

Parallel code:

```
1  x=0;
2  sum = 0.0;
3  step = 1.0/(double) num_steps;
4  #pragma omp parallel private(i,x,aux) shared(sum)
5  {
6  #pragma omp for schedule(static)
7      for (i=0; i<num_steps; i=i+1){
8
9          x=(i+0.5)*step;
10         aux=4.0/(1.0+x*x);
11 #pragma omp critical
12         sum = sum + aux;
13     }
14 }
15 pi=step*sum;
```

High Performance Computing at EUC

- **Astrophysics and High Performance Computing Research Group** (<http://ahpc.euc.ac.cy>)

- Applying HPC for developing efficient solutions

Currently:

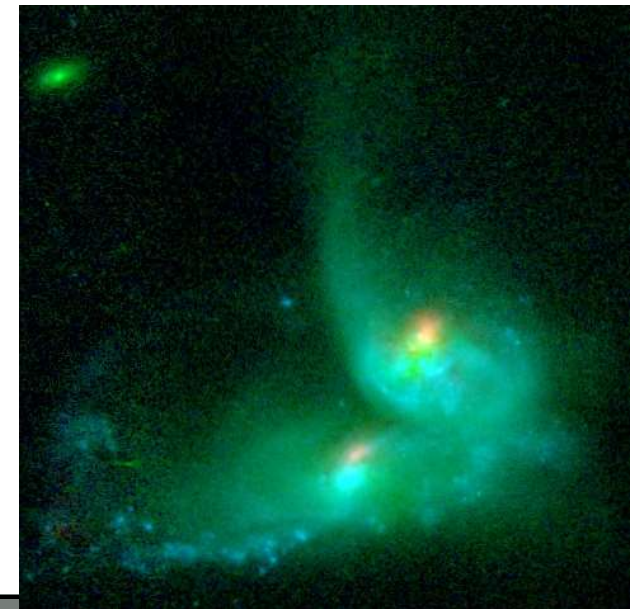
- **in Astrophysics**

- **In Galaxies evolution**

Future plans:

- in Health Sciences
- Medical imaging
- more

HST IMAGE OF 08572+3915



AHPC Research Group

- Head: Prof. Andreas Efsthathiou

Members:

Ass. Prof. Vicky Papadopoulou

- Researchers:

- Natalie Christopher
- Andreas Papadopoulos

- Phd students

- Elena Stylianou

- Undergraduate Students:

- Michalis Kyprianou (Bachelor CS, EUC)
- Andreas Howes (Bachelor CS, EUC)



Parallel Processing in Python

In another Colloquium's talk!

