

# Introduction To Parallel Computing

---

이 정 근 (Jeong-Gun Lee)

한림대학교 컴퓨터공학과, 임베디드 SoC 연구실

[www.onchip.net](http://www.onchip.net)


Email: [Jeonggun.Lee@hallym.ac.kr](mailto:Jeonggun.Lee@hallym.ac.kr)





# Contents

---

- **Introduction to Parallel Computing / High Performance Computing (HPC)**
  - **Concepts and terminology**
  - **Parallel programming models**
  - **Parallelizing your programs**
  - **Analytic Measure for Parallel Algorithms**
- 

# Parallel Computing

- Multicore/Manycore and GPU

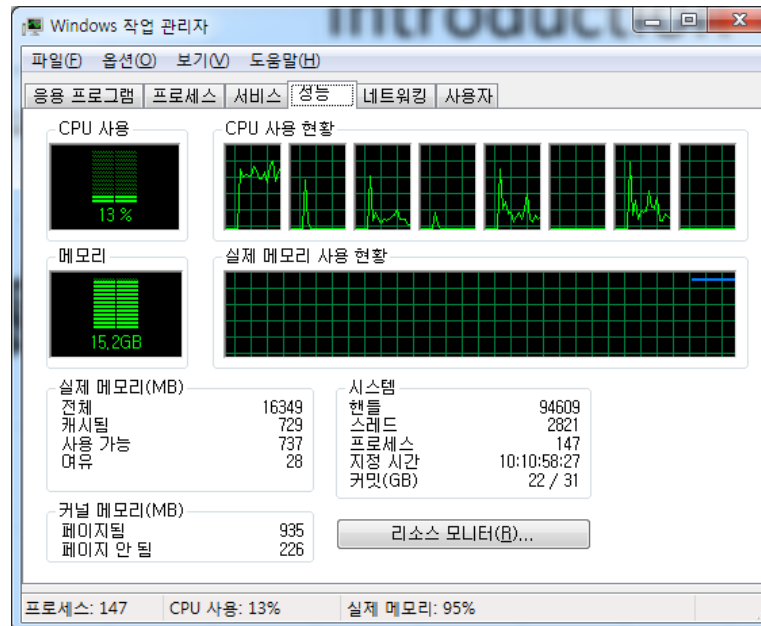


## 인텔 코어i7-7세대 7700 (카비레이크) (정품) 표준PC

인텔(소켓1151) / 쿼드 코어 / 쓰레드 8개 / 64비트 / 3.6GHz / 8MB / 인텔 HD 630 / 350MHz / 65W / 하이퍼스레딩

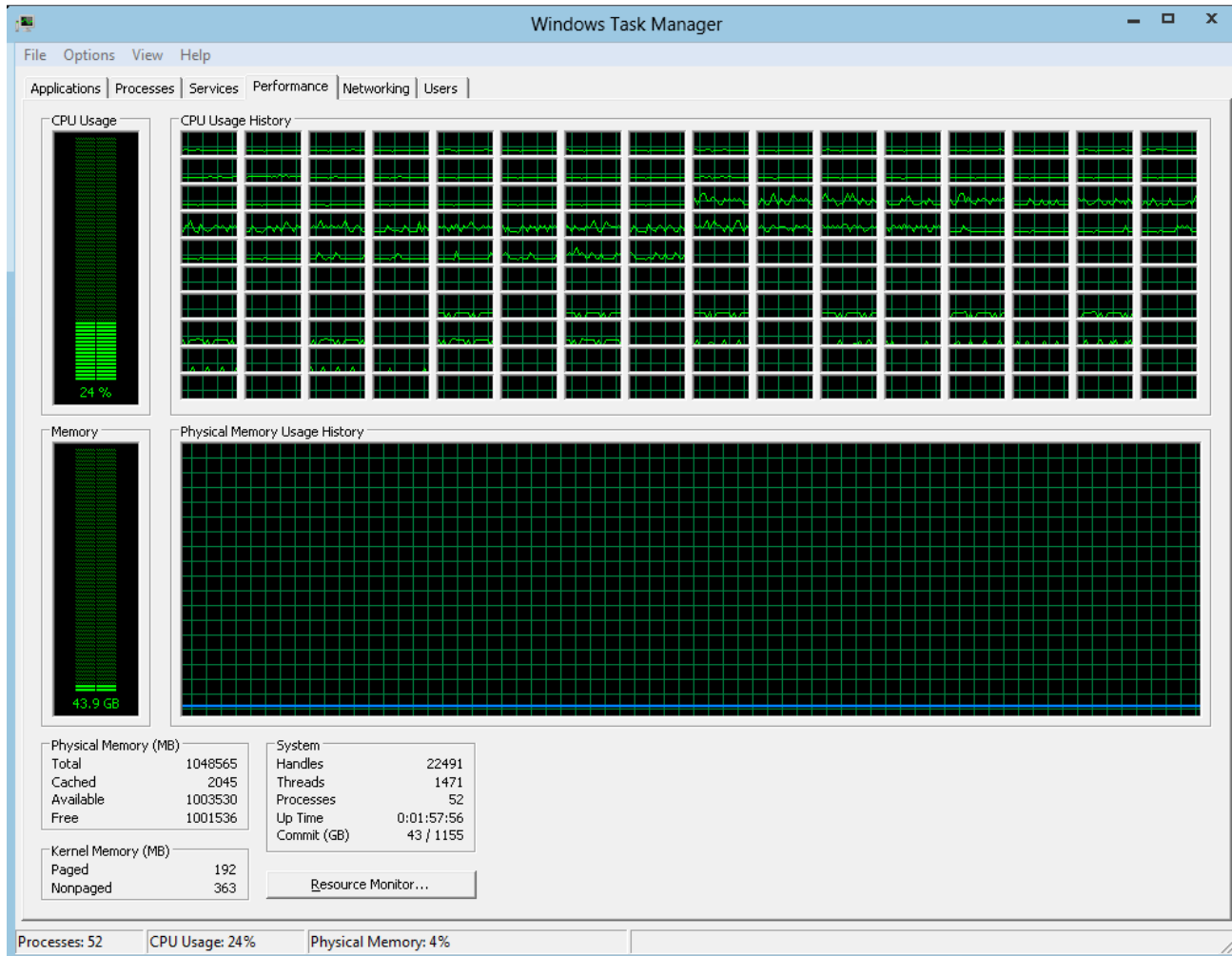
관련기사 인텔 공인대리점, 7세대 코어 프로세서 불맞이 퀴즈 이벤트

사용기 [똑똑한 리뷰씨] 게임에 강한 프로세서! 7세대 인텔 코어 프로세서



# Parallel Computing

- Multicore/Manycore and GPU



# Parallel Computing

- First, **Wiki** Definition ...

## 병렬 컴퓨팅

위키백과, 우리 모두의 백과사전.

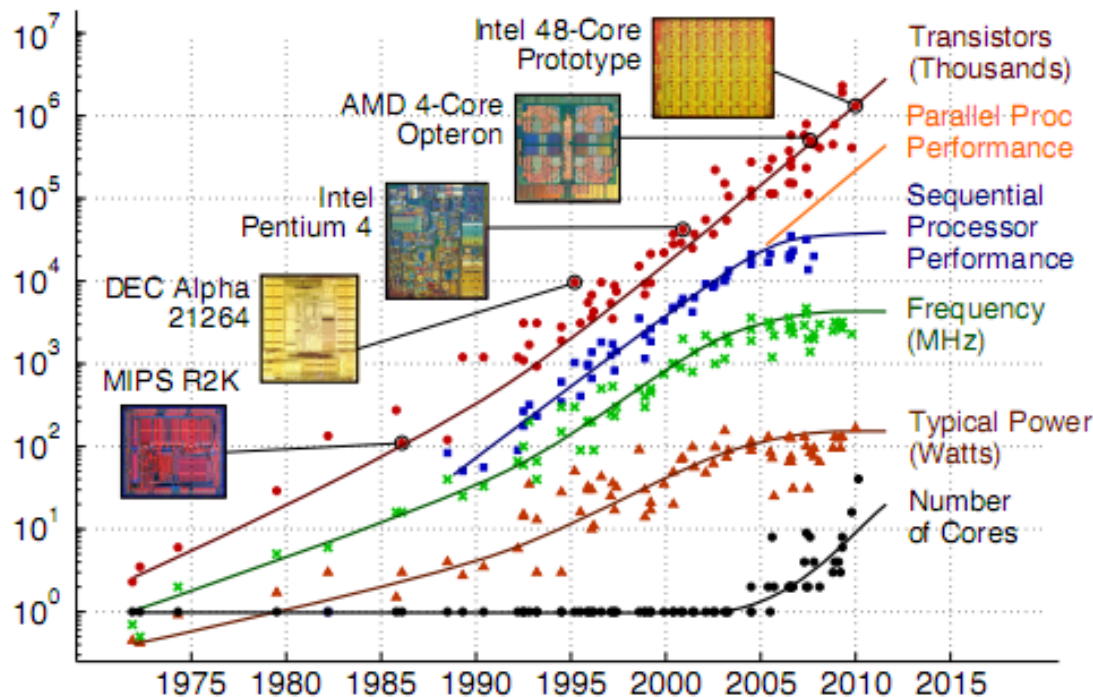
병렬 컴퓨팅(parallel computing) 또는 병렬 연산은 동시에 많은 계산을 하는 연산의 한 방법이다. 크고 복잡한 문제를 작게 나눠 동시에 병렬적으로 해결하는 데에 주로 사용되며<sup>[1]</sup>, 병렬 컴퓨팅에는 여러 방법과 종류가 존재한다. 그 예로, 비트 수준, 명령어 수준, 데이터, 작업 병렬 처리 방식 등이 있다. 병렬 컴퓨팅은 오래전부터 주로 고성능 연산에 이용되어 왔으며, 프로세서 주파수<sup>[2]</sup>의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목받게 되었다<sup>[3]</sup>.



프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

# Parallel Computing: Why ?

프로세서 주파수의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - <http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf>



Intel® Xeon Phi™ Processor 7290F  
16GB, 1.50 GHz, 72 core

Xeon Phi 7200 Series	sSpec Number	Cores (Threads)	Frequency	Turbo
Xeon Phi 7210	SR2ME (B0) SR2X4 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7210F	SR2X5 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7230	SR2MF (B0) SR2X3 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7230F	SR2X2 (B0)	64 (256)	1300 MHz	1500 MHz
Xeon Phi 7250	SR2MD (B0) SR2X1 (B0)	68 (272)	1400 MHz	1600 MHz
Xeon Phi 7250F	SR2X0 (B0)	68 (272)	1400 MHz	1600 MHz
Xeon Phi 7290	SR2WY (B0)	72 (288)	1500 MHz	1700 MHz
Xeon Phi 7290F	SR2WZ (B0)	72 (288)	1500 MHz	1700 MHz

Each core will have two 512-bit vector units and will support [AVX-512](#) SIMD instructions

# Parallel Computing: Why ?

프로세서 주파수의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

• Worse news: Power (normalized to i486) trends

*Small is Beautiful !*

*That does not immediately imply that  
“smallest is best” !*

- **Parallelism** is an **energy-efficient way to achieve performance** [Chandrakasan et al 1992]
- A larger number of smaller processing elements allows a finer-grained ability to perform dynamic voltage scaling and power down

# Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

- Multi-core Challenges: The **3 P's**
  - Performance challenge
  - **Power efficiency challenge**



Processor	Power	Perf.	Power Efficiency
Itanium 2	100W	1	1
RISC*	1/2W	1/8X**	<b>25X</b>

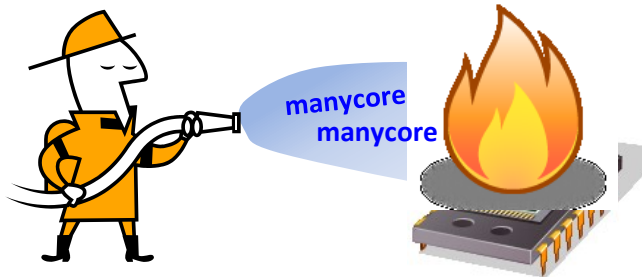
Assuming 130nm  
\* 90's RISC at 405 MHz  
\*\* e.g., Timberwolf (SpecInt)

- Programming challenge



# Parallel Computing: Why ?

프로세서 주파수의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

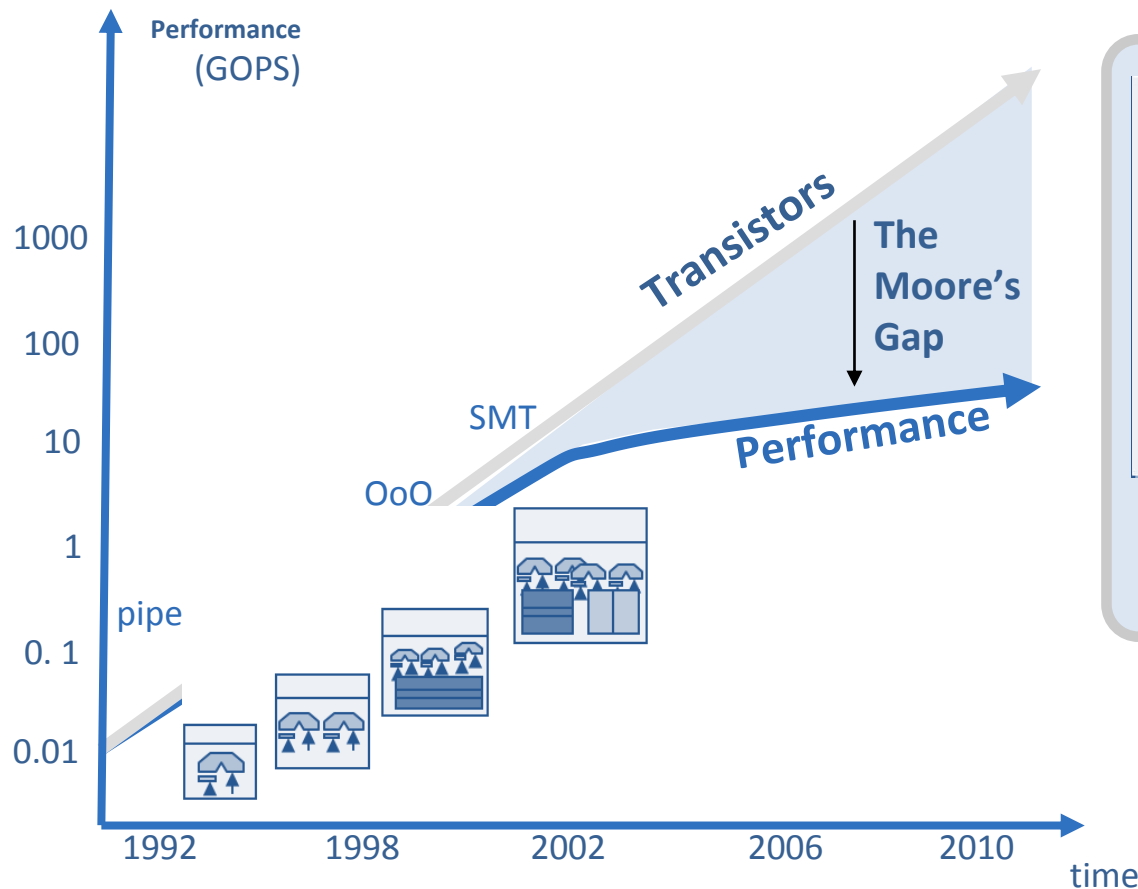


+ Diminishing Return

# Parallel Computing: Why ?

프로세서 주파수 이 물리적인 한계에 다가가면서 문제 복잡도가 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용이 더 발전한 전력소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

## Diminishing Return



Pentium 3	Pentium 4
- 1 GHz	- 1.4 GHz
- Year 2000	- Year 2000
- 0.18 um	- 0.18 um
- 28M Tr.	- 42M Tr. 50%↑
343 (SpecInt 2000)	393 15%↑ (SpecInt 2000)

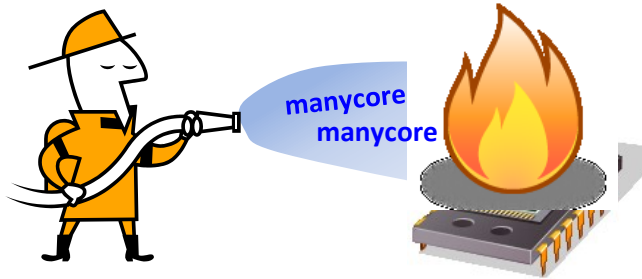
1. **Diminishing returns** from single CPU

2. **Wire** delays

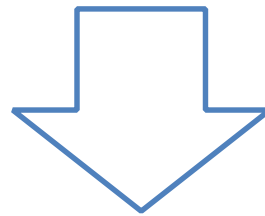
3. **Power** envelopes

# Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목



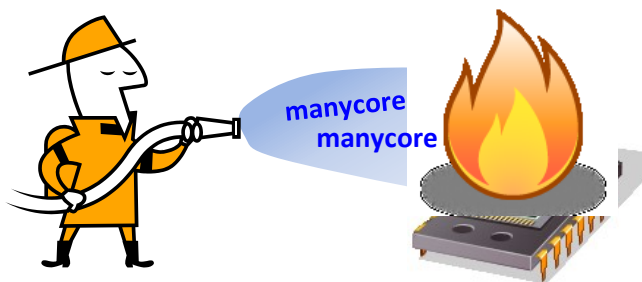
+ Diminishing Return



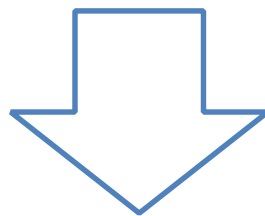
Manycore  
→ Parallel Computing

# Parallel Computing: Why ?

프로세서 주파수 의 물리적인 한계에 다가가면서 문제 의식이 높아진 이후에 더욱 주목 받게 되었다. 최근 컴퓨터 이용에서 발열과 전력 소모에 대한 관심이 높아지는 것과 더불어 멀티 코어 프로세서를 핵심으로 컴퓨터 구조에서 강력한 패러다임으로 주목

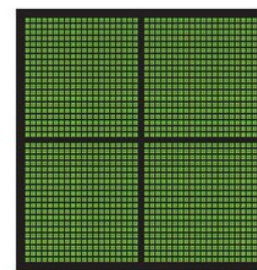


+ Diminishing Return



Manycore

→ Parallel Computing



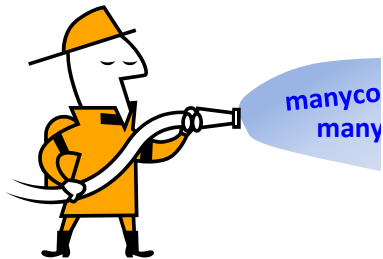
GPU

GPU  
THOUSANDS OF CORES

# Parallel Computing: Why ?

프로세서 주파수의 물리적 한계에  
도달했다. 최근 컴퓨터 아키텍처는  
코어 프로세서의 수를 늘려

후에 더욱 주목 받게  
된 것과 더불어 멀티  
코어 프로세서



g Return

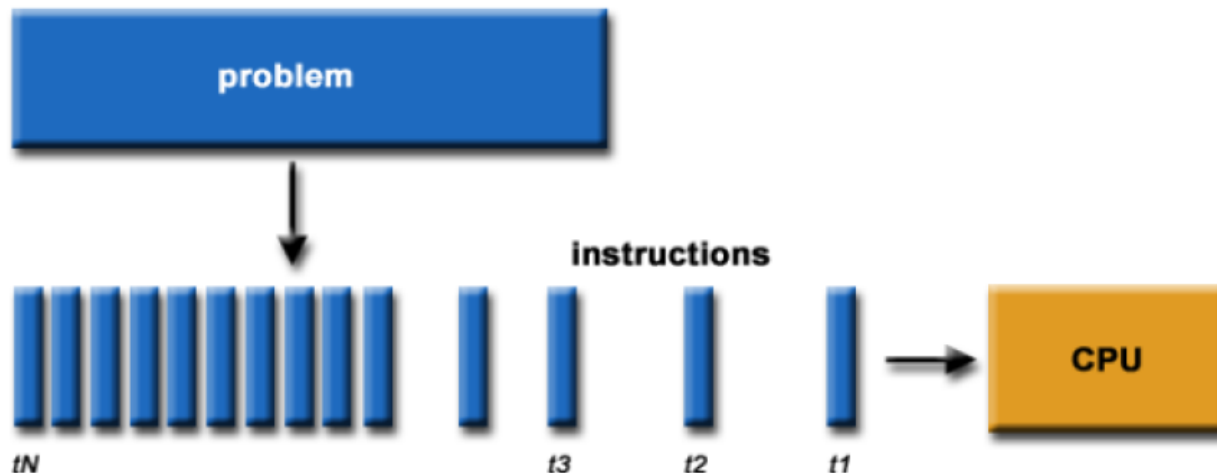


5

→ Parallel Computing

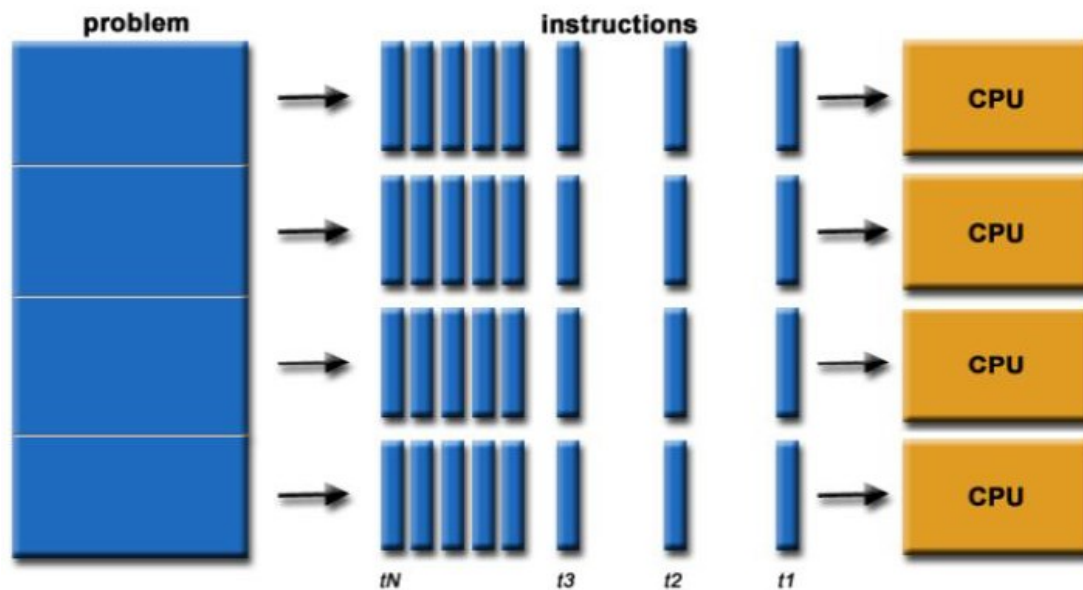
# Serial Computation

- Traditionally software has been written for **serial** computations:
  - To be run on a single computer having a single Central Processing Unit (CPU)
    - A problem is broken into a discrete set of instructions
    - Instructions are executed one after another
    - Only one instruction can be executed at any moment in time



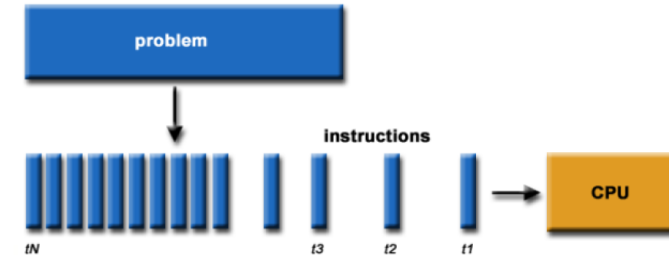
# Parallel Computing

- Simply, parallel computing is the simultaneous use of **multiple compute resources** to solve a computational problem:
  - To be run using **multiple CPUs**
    - A problem is broken into discrete parts that can be solved **concurrently**
    - Each part is further broken down to a series of instructions
    - Instructions from each part execute simultaneously on different CPUs

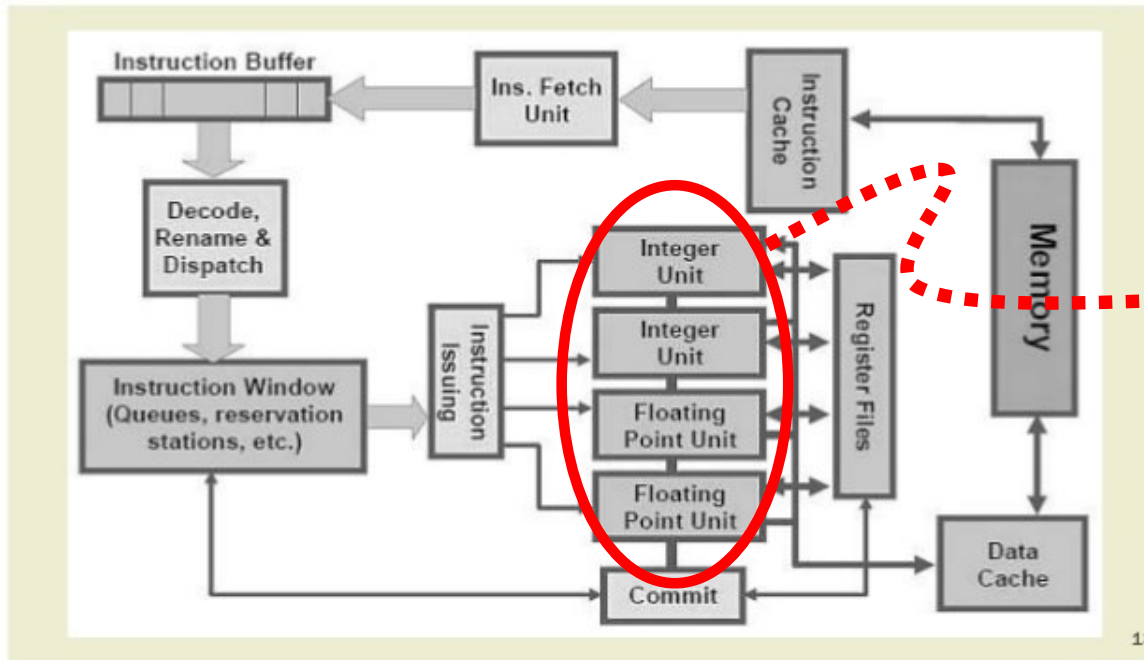


# Single CPU → No Parallelism ?

- If a CPU has multiple functional units ?



## Instruction Flow in Superscalar Architecture

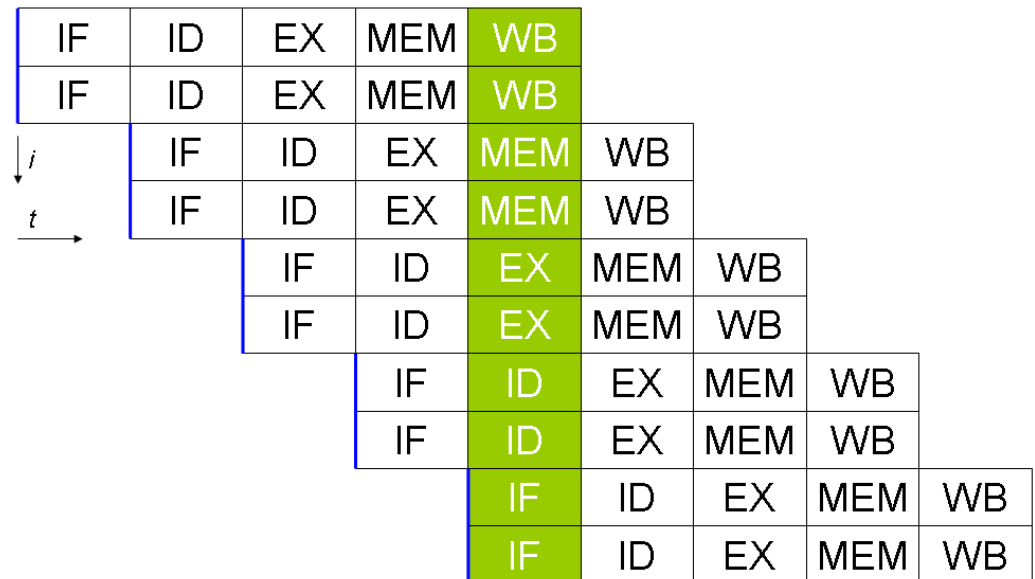
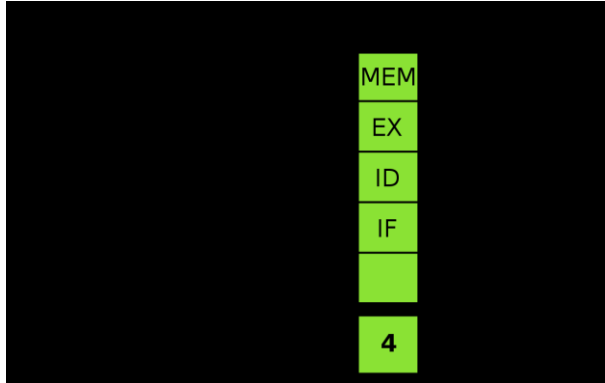


Instruction Level Parallelism (ILP)



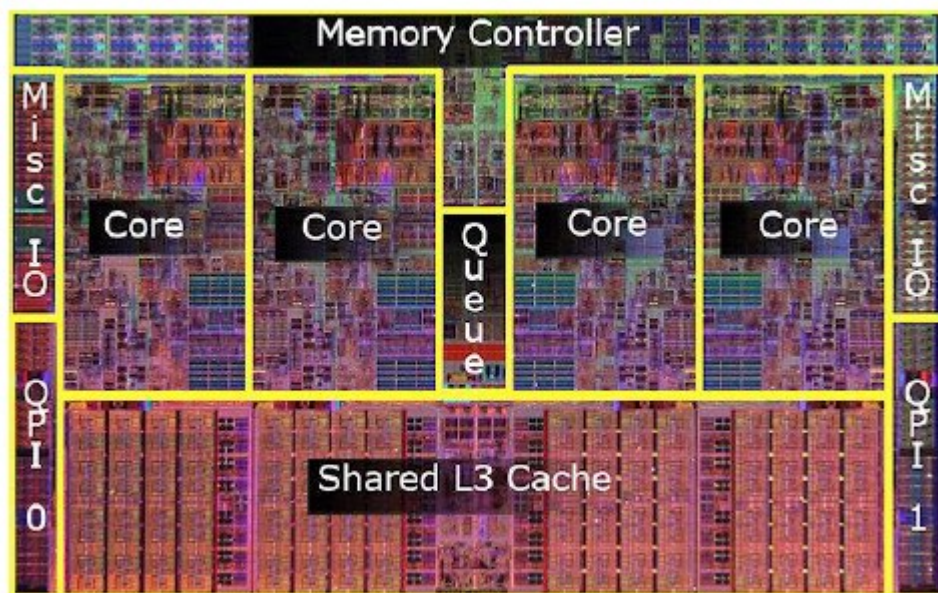
# Instruction Level Parallelism

- **ILP** = Instruction Level Parallelism
  - Even in a single CPU, more than one instruction can be executed at a time with advanced microarchitecture techniques



# Parallel Computers

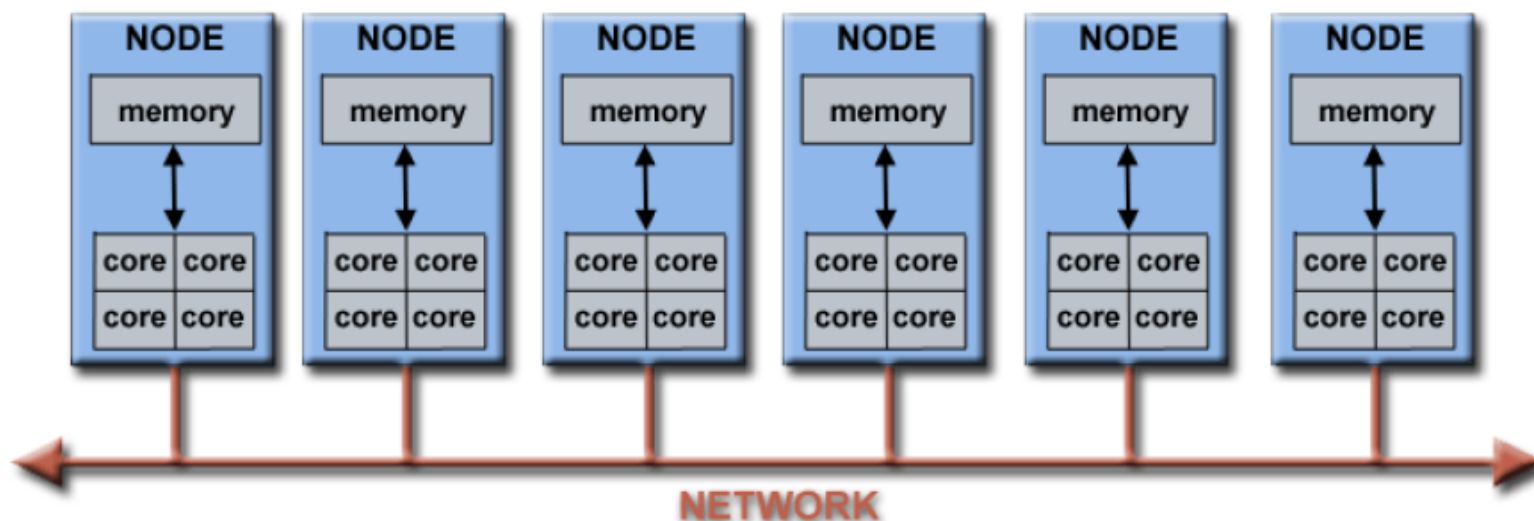
- Virtually all stand-alone computers today are parallel machines from a hardware perspective:
  - Multiple functional units (floating point, integer, GPU, etc.)
  - Multiple execution units / cores
  - Multiple hardware threads



Intel Core i7 CPU and  
its major components

# Parallel Computers

- Networks connect multiple stand-alone computers (nodes) to create **larger parallel computer clusters**
  - Each compute node is a multi-processor parallel computer in itself
  - Multiple compute nodes are networked together with an InfiniBand network
  - Special purpose nodes, also multi-processor, are used for other purposes





# HPC Terminology 1

---

- **Supercomputing / High-Performance Computing (HPC)**
- **Flop(s)** – Floating point operation(s)
- **Node** – a stand alone **computer**
- **CPU / Core** – a modern CPU usually has several cores (individual processing units )
- **Task** – a logically discrete section from the computational work
- **Communication** – data exchange between parallel tasks



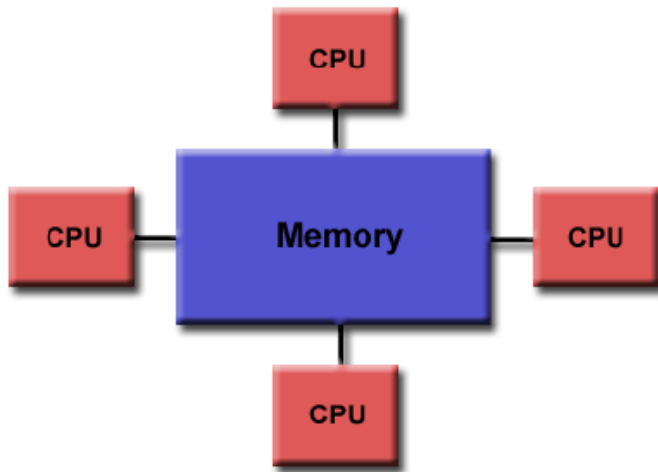
# HPC Terminology 2

---

- **Speedup** – time of serial execution / time of parallel execution
- **Massively Parallel** – refer to **hardware** of parallel systems with many processors (“many” = hundreds of thousands)
- **Pleasantly Parallel** – solving many similar but independent tasks simultaneously. Requires very **little communication** (*Embarrassingly Parallel*)
- **Scalability** - a proportionate increase in parallel speedup with the addition of more processors

# Parallel Computer Memory Architectures

- *Shared Memory:*



- Multiple processors can operate independently, but **share the same memory** resources
- Changes in a memory location caused by one CPU are visible to all processors

## Advantages:

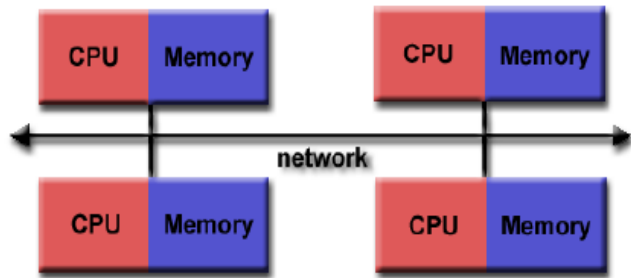
- Global address space provides a user-friendly programming perspective to memory
- Fast and uniform data sharing due to proximity of memory to CPUs

## Disadvantages:

- **Lack of scalability** between memory and CPUs. Adding more CPUs increases traffic on the shared memory-CPU path
- Programmer responsibility for “correct” access to global memory

# Parallel Computer Memory Architectures

- *Distributed Memory:*



- Requires a communication network to connect inter-processor memory
- Processors have their **own local memory**.
- Changes made by one CPU have no effect on others
- Requires communication to exchange data among processors

## Advantages:

- Memory is **scalable** with the number of CPUs
- Each CPU can rapidly access its own memory without overhead incurred with trying to maintain global cache **coherency**

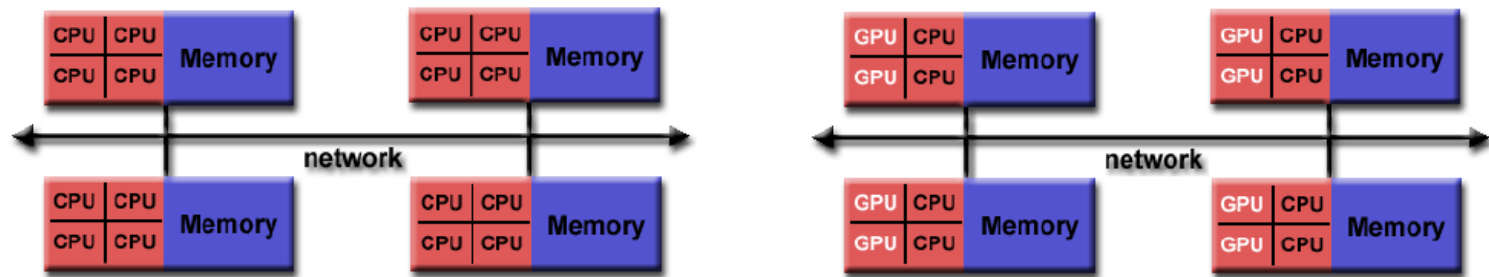
## Disadvantages:

- Programmer is responsible for **many of the details** associated with data communication between processors

# Parallel Computer Memory Architectures

- *Hybrid Distributed-Shared Memory:*

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- Shared memory component can be a shared memory machine and/or GPU
- Processors on a compute node share same memory space
- Requires communication to exchange data between compute nodes
- Advantages and Disadvantages:
  - Increased scalability is an important advantage
  - Increased programming **complexity** is a major disadvantage



# Parallel Programming Models

- Parallel Programming Models exist as an abstraction above hardware and memory architectures
  - Shared Threads Models (**Pthreads**, **OpenMP**)
  - Distributed Memory / Message Passing (**MPI**)
  - Hybrid
  - **Single Instruction/Program Multiple Data (SIMD, SPMD)**
    - Vector Processing (MMX, SSE, AVX, ...)
    - Single Instruction Multiple Thread (SIMT for GPU)
  - Multiple Program Multiple Data
  - ...

# Shared Threads Models

- **POSIX Threads**

## POSIX

From Wikipedia, the free encyclopedia

*Not to be confused with [Unix](#), [Unix-like](#), or [Linux](#).*

The **Portable Operating System Interface (POSIX)**<sup>[1]</sup> is a family of [standards](#) specified by the [IEEE Computer Society](#) for maintaining compatibility between [operating systems](#). POSIX defines the [application programming interface](#) (API), along with command line [shells](#) and utility interfaces, for software compatibility with variants of [Unix](#) and other operating systems.<sup>[2][3]</sup>

- **Library** based; requires parallel coding
- C Language only; Interfaces for Perl, Python and others exist
- Commonly referred to as **Pthreads**
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
- Very **explicit parallelism**; requires significant programmer attention to detail

# Shared Threads Models

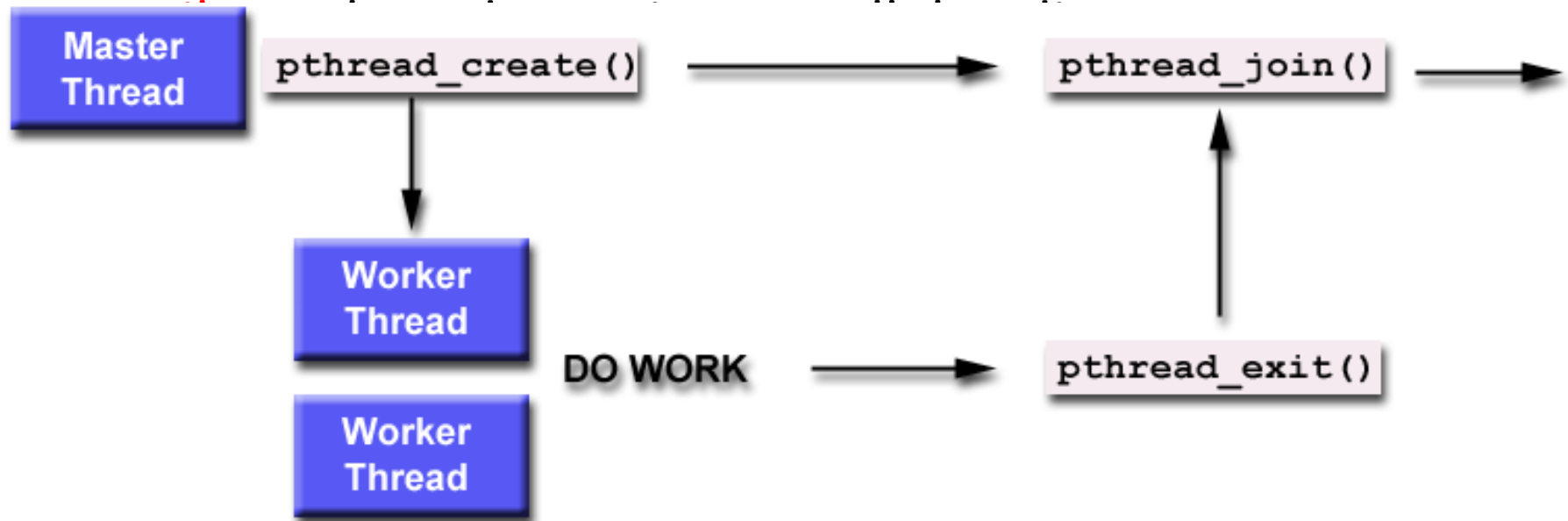
- POSIX Threads

## POSIX

From Wikipedia, the free encyclopedia

*Not to be confused with [Unix](#), [Unix-like](#), or [Linux](#).*

The **Portable Operating System Interface** (**POSIX**)<sup>[1]</sup> is a family of [standards](#) specified by the [IEEE Computer Society](#) for maintaining compatibility between [operating systems](#). POSIX defines the [application programming interface](#) (API), along with command line [shells](#) and utility interfaces, for software compatibility with variants of [Unix](#) and other operating systems.<sup>[2][3]</sup>






# Shared Threads Models

---

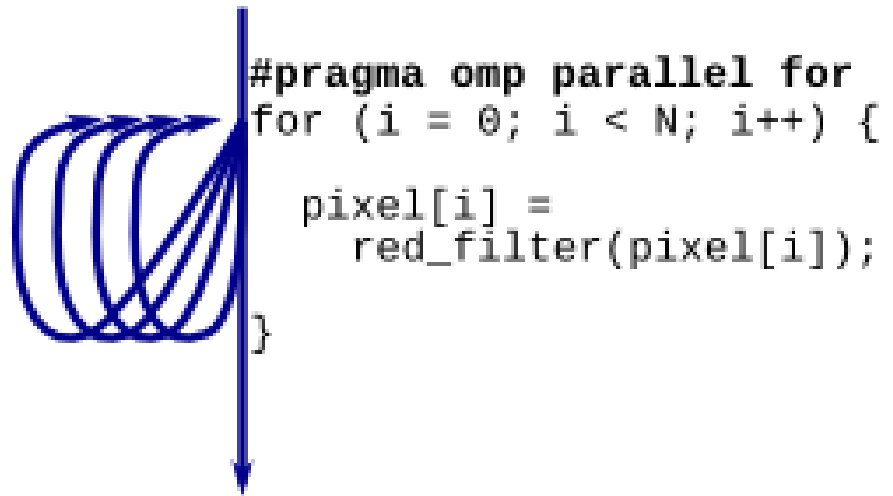
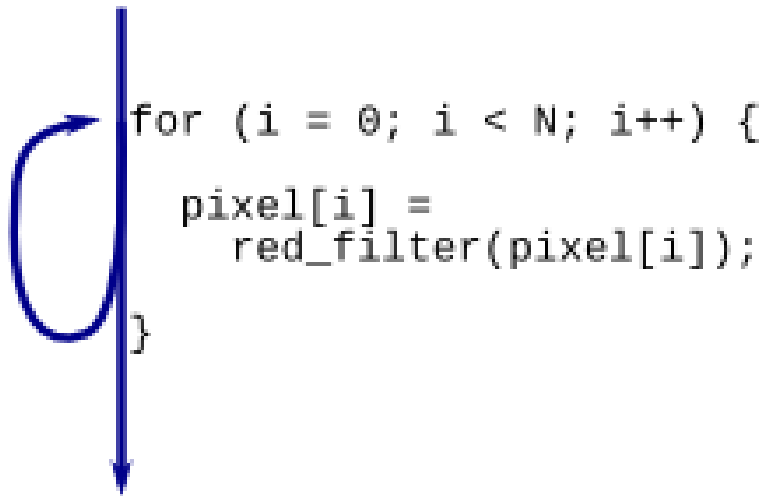
- **OpenMP**

- **Compiler directive based**; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
  - **Portable / multi-platform, including Unix and Windows platforms**
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for **"incremental parallelism"**
- 

# Shared Threads Models

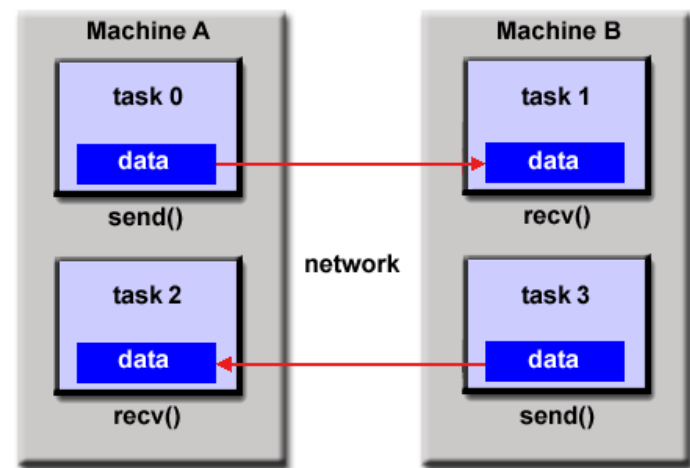
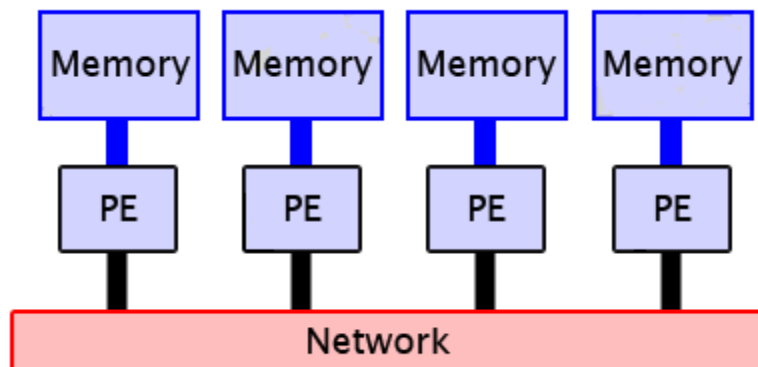
- **OpenMP**

- **Compiler directive based**; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors



# Distributed Memory / Message Passing Models

- A set of tasks that use **their own local memory** during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines
- Tasks **exchange data through communications by sending and receiving messages**



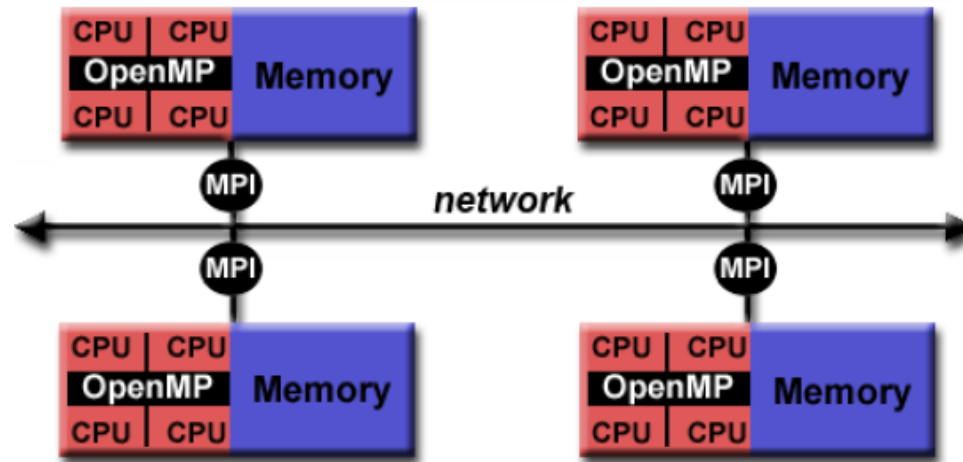


# Distributed Memory / Message Passing Models

---

- Data transfer usually **requires cooperative operations** to be performed by each process. For example, a send operation must have a matching receive operation
- **Message Passing Interface (MPI)** is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms

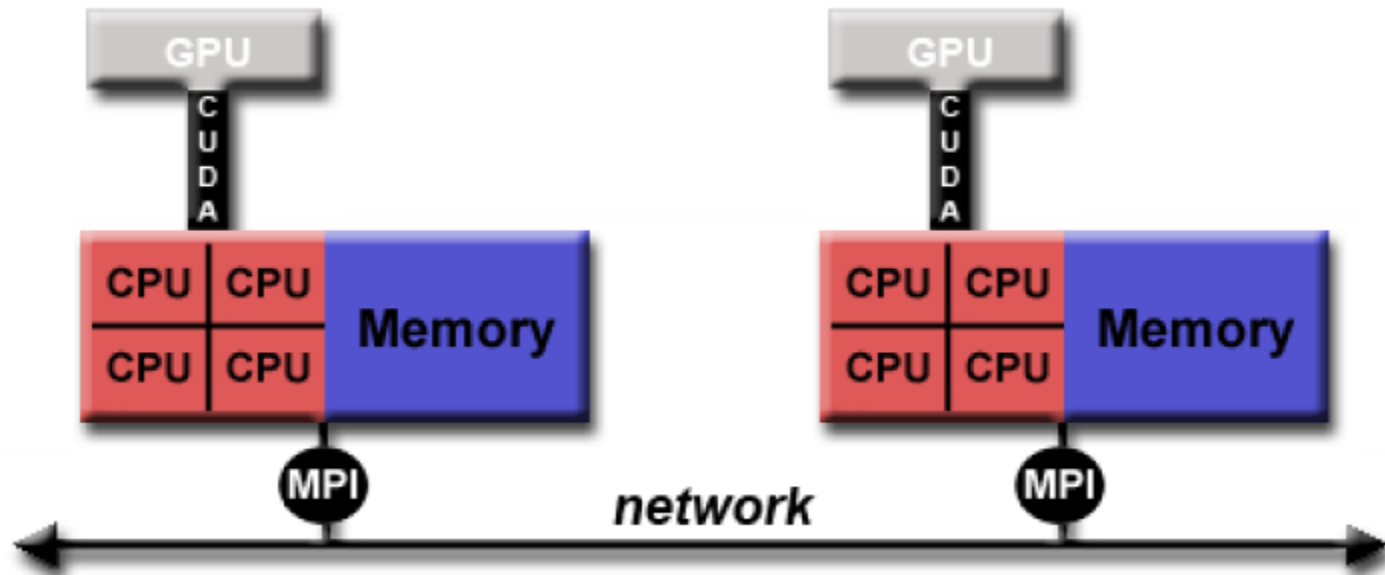
# Hybrid Parallel Programming Models



- Currently, a common example of a hybrid model is the combination of the message passing model (**MPI**) with the threads model (**OpenMP**)
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the increasingly common hardware environment of **clustered multi/many-core machines**



# Hybrid Parallel Programming Models



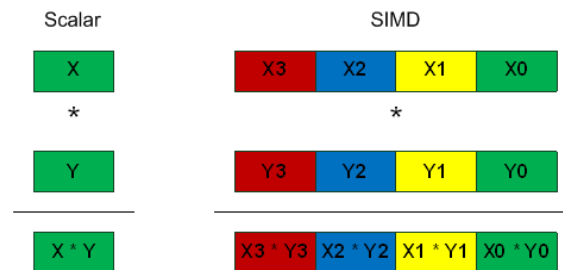
- Another similar and increasingly popular example of a hybrid model is using **MPI** with **GPU** (Graphics Processing Unit) programming
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

# Single Instruction Multiple Data

- Single Instruction/Program Multiple Data (SIMD/SPMD)

- Vector Processing

- Intel: MMX, SSE, **AVX**
- ARM: **NEON**



- Single Instruction Multiple Thread (SIMT)

- **GPU !**

## Single instruction, multiple threads

From Wikipedia, the free encyclopedia

**Single instruction, multiple thread** (SIMT) is an execution model used in [parallel computing](#) where [single instruction, multiple data](#) (SIMD) is combined with [multithreading](#).

The processors, say a number  $p$  of them, seem to execute many more than  $p$  tasks. This is achieved by each processor having multiple "threads" (or "work-items" or "Sequence of SIMD Lane operations"), which execute in lock-step, and are analogous to [SIMD lanes](#).<sup>[1]</sup>



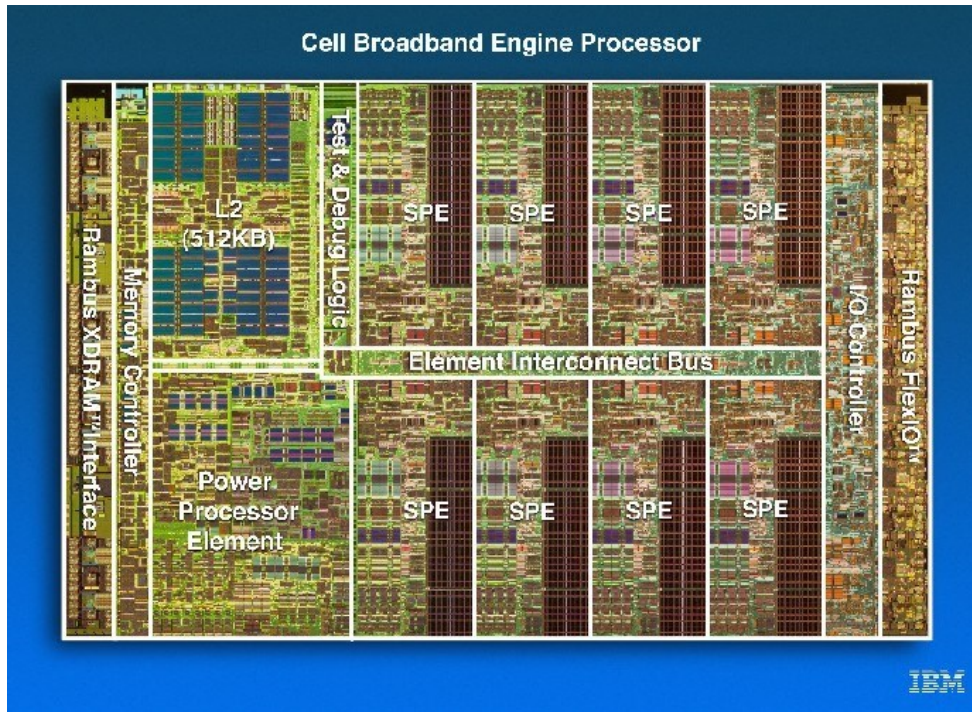
# Multiple Program Multiple Data

---

- Multiple autonomous processors simultaneously operating at least 2 independent programs.
- Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "Manager/Worker" strategy), which runs one program that farms out data to all the other nodes which all run a second program.
- Those other nodes then return their results directly to the manager.
  - An example of this would be the Sony PlayStation 3 game console, with its SPU/PPU processor.

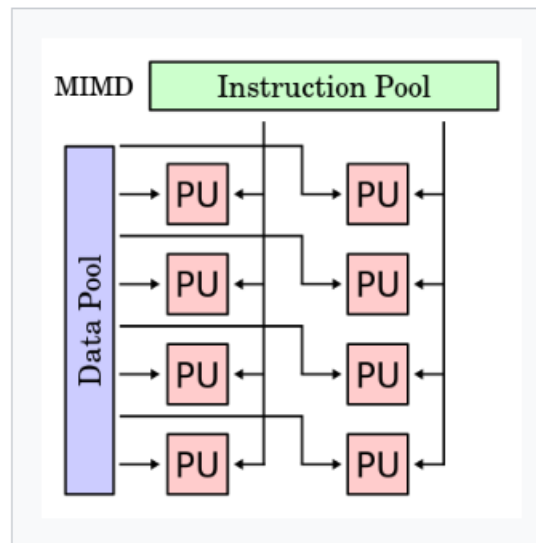
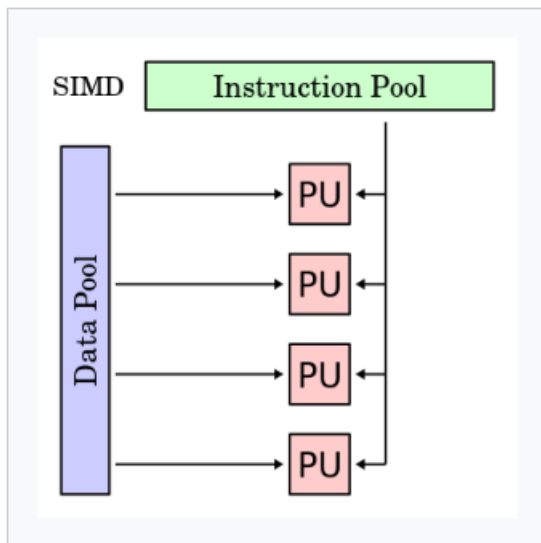
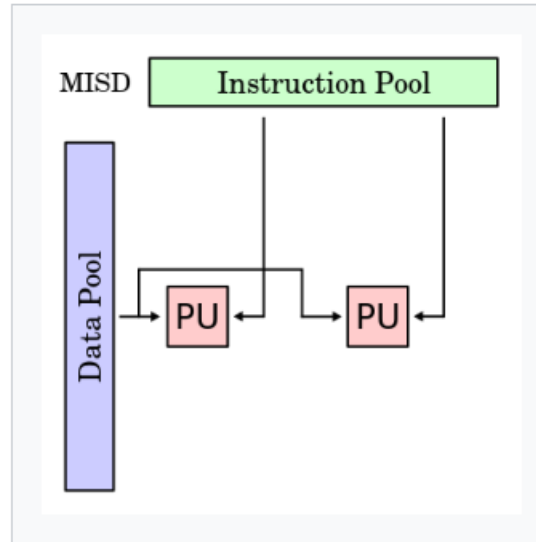
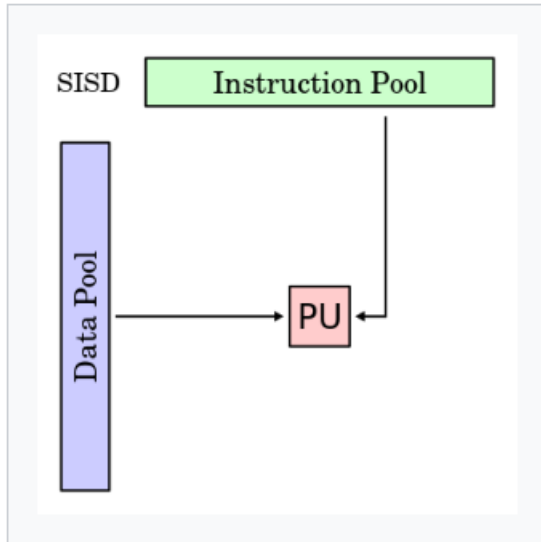
# Multiple Program Multiple Data

- Those other nodes then return their results directly to the manager.
  - An example of this would be the Sony PlayStation 3 game console, with its **SPU/PPU processor**.

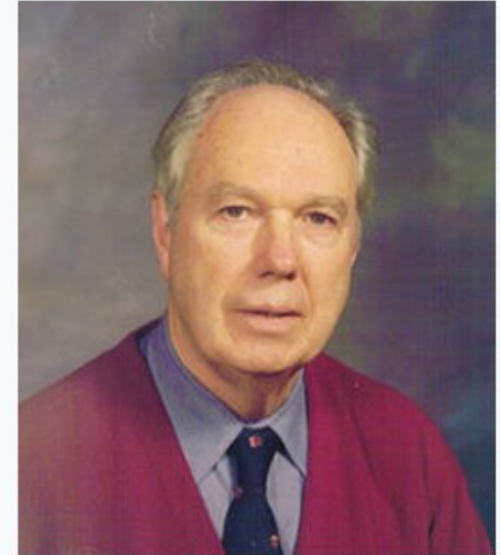


- 1 PPE core @ 3.2GHz
  - 64bit hyperthreaded PowerPC
  - 512KB L2 cache
- 8 SPE cores @ 3.2GHz
  - 128bit SIMD optimized
  - 256KB SRAM

# Flynn's Classification



**Michael Flynn**



<b>Born</b>	May 20, 1934 (age 82) New York City
<b>Nationality</b>	American
<b>Institutions</b>	Stanford University
<b>Alma mater</b>	Manhattan College Syracuse University Purdue University
<b>Known for</b>	Flynn's taxonomy
<b>Notable awards</b>	Harry H. Goode Memorial Award

# Can my code be parallelized?

## Interesting Quotes about Parallel Programming


- 1 “There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are.”  
~ W. Somerset Maugham, Gary Montry
- 2 “The wall is there. We probably won’t have any more products without multicore processors [but] we see a lot of problems in parallel programming.” ~ Alex Bachmutsky
- 3 “We can solve [the software crisis in parallel computing], but only if we work from the algorithm down to the hardware — not the traditional hardware-first mentality.”  
~ Tim Mattson
- 4 “[The processor industry is adding] more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.” ~ Steve Jobs





# Can my code be parallelized?

---

- Does it have **large loops** that repeat the same operations?
  - Does your code do **multiple tasks** that are **not dependent on one another**? If not, is the dependency weak?
  - Is the **amount of communications small**?
  - Do multiple tasks depend on the **same data**?
  - Does the **order of operations matter**?
- 





# Guidance for Efficient Parallelization

---

- Is it even worth parallelizing my code?
  - Does your code take an intractably long amount of time to complete?
  - Do you run a single large model or do statistics on multiple small runs?
  - Would the amount of time it take to parallelize your code be worth the gain in speed?



# Guidance for Efficient Parallelization

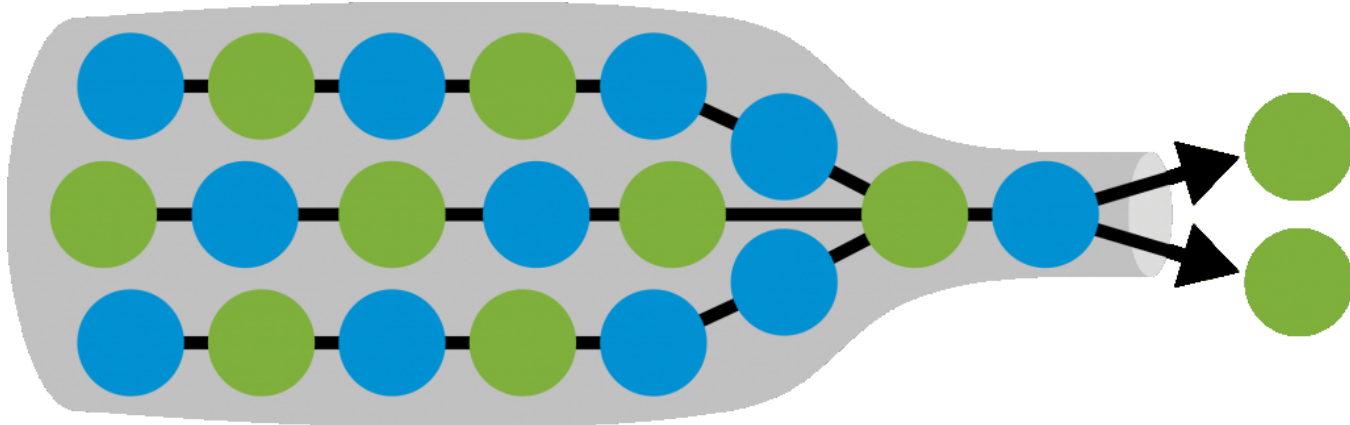
---

- Parallelizing established code vs. starting from scratch
  - **Established code**: Maybe easier / faster to parallelize, but may not give good performance or scaling (OpenMP)
  - **Start from scratch**: Takes longer, but will give better performance, accuracy, and gives the opportunity to turn a “black box” into a code you understand



# Guidance for Efficient Parallelization

- Increase the fraction of your program that can be parallelized
  - **Identify the most time consuming parts** of your program and parallelize them. This could require modifying your intrinsic algorithm and code's organization



# Guidance for Efficient Parallelization

- Balance parallel workload
- Minimize time spent in communication
- Use **simple arrays** instead of user defined derived types

```
int simpleArray[MAX]
```



0 1 2 3 4 5 6 7

(ta-da!)

# Considerations about parallelization

- You parallelize your program to run faster, and to solve larger and more complex problems.
- How much faster will the program run?

**Speedup:**

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the job  
on **one** process

Time to complete the job  
on **n** process

**Efficiency:**

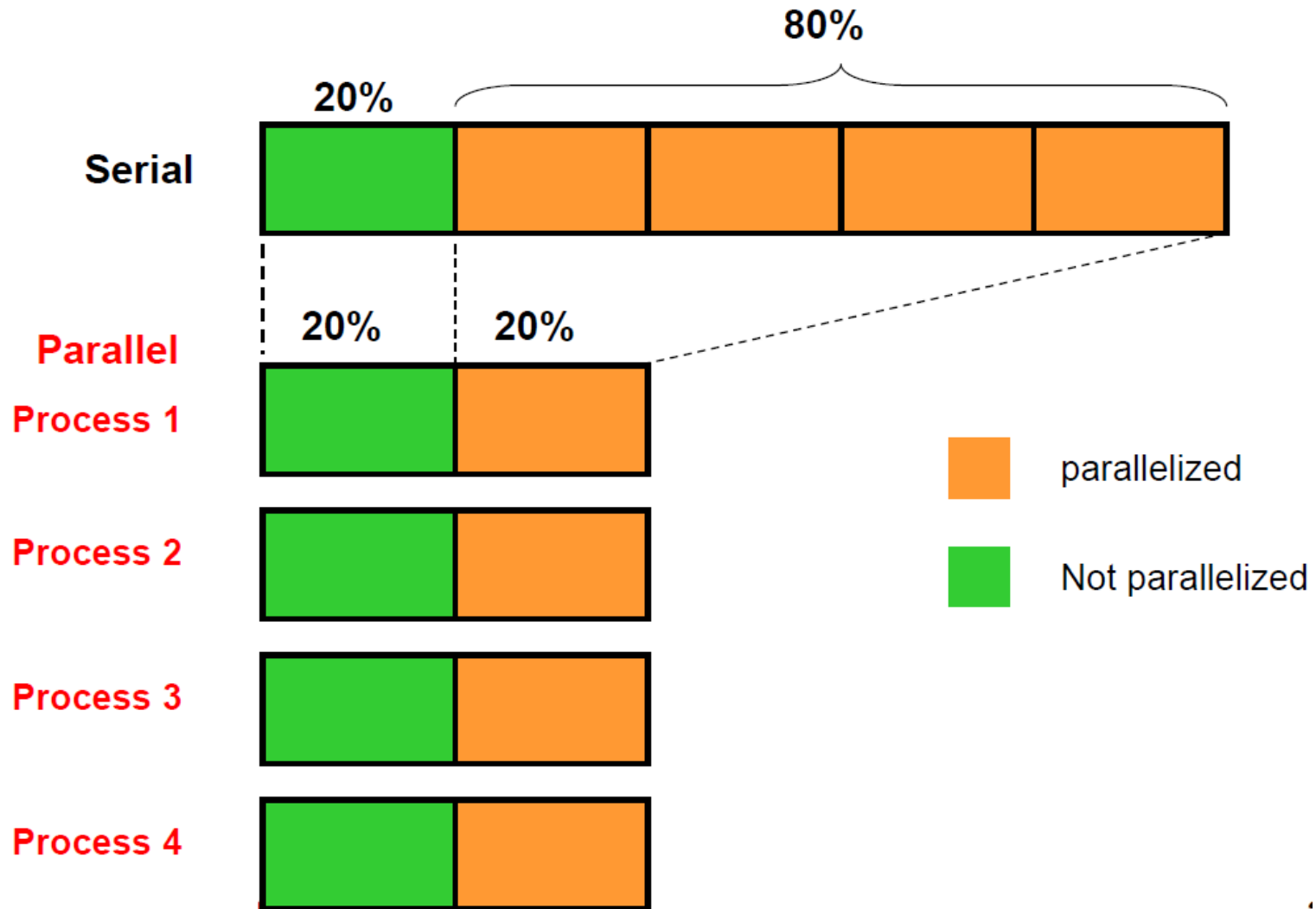
$$E(n) = \frac{S(n)}{n}$$

Tells you how efficiently you parallelize  
your code

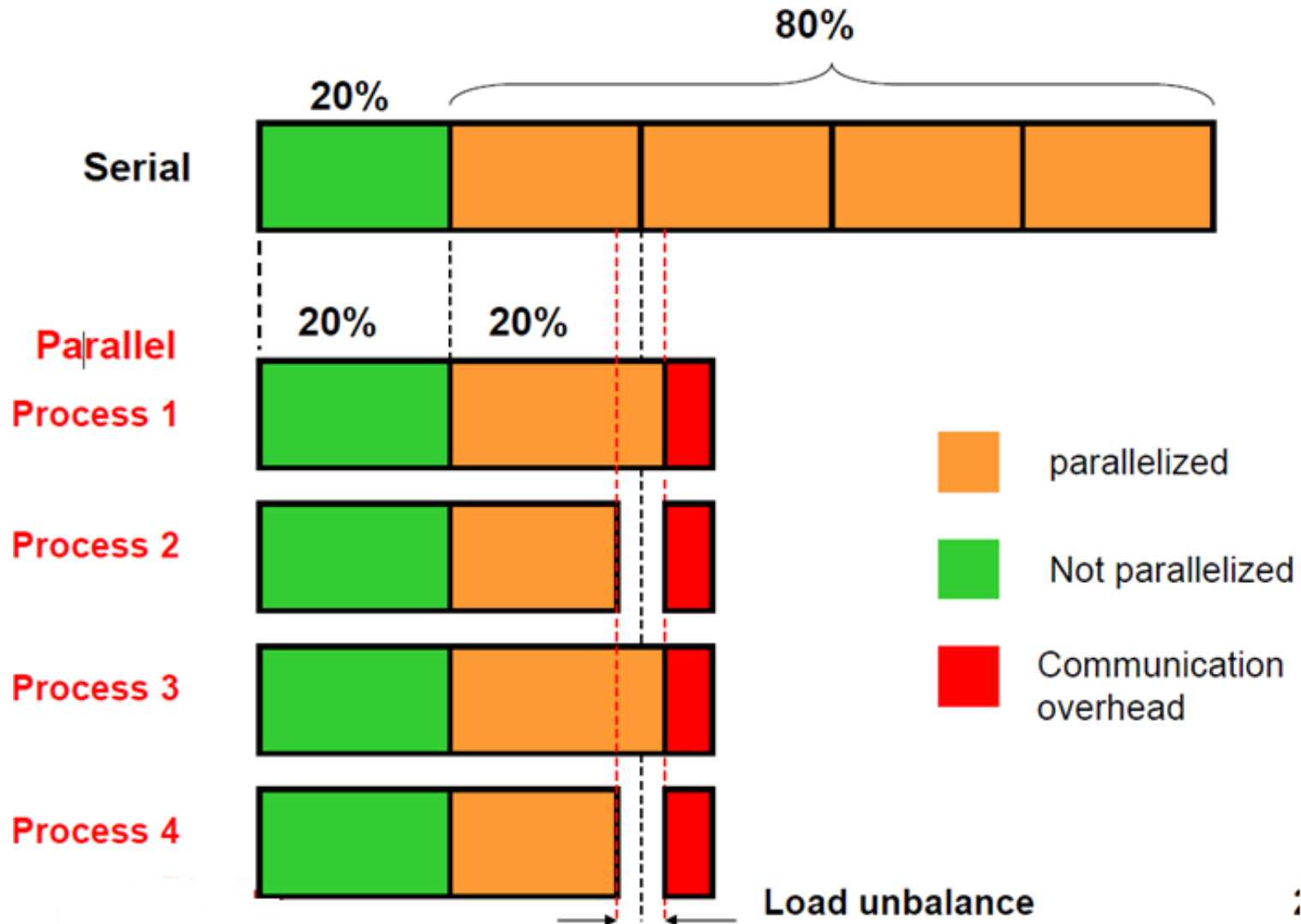
# Oversimplified example

- “p” : fraction of program that can be parallelized
- “1 – p” : fraction of program that cannot be parallelized
- “n”: number of processors
  - Then the time of running the parallel program will be “(1 – p) + p/n” of the time for running the serial program
- 80% can be parallelized & 20 % cannot be parallelized & n = 4
  - $[1 - 0.8] + [0.8 / 4] = 0.4$  i.e., 40% of the time for running the serial code
- You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

# Oversimplified example

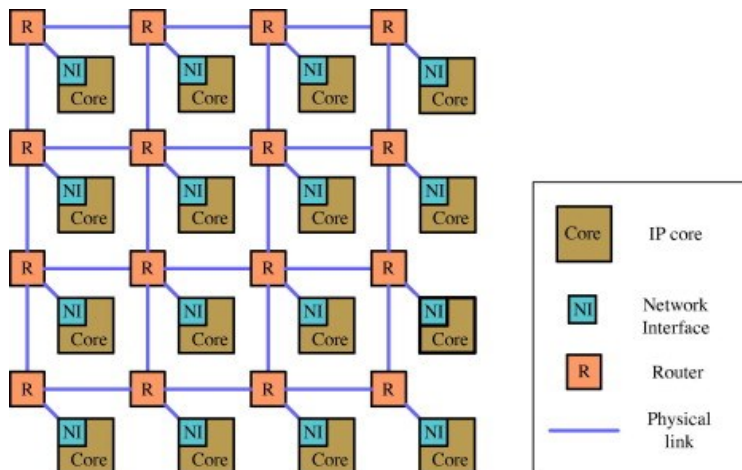


# More realistic example:



# Designing parallel programs - communication

- Most parallel applications require tasks to share data with each other.
  - **Cost of communication**: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.
  - **Latency vs. Bandwidth**: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

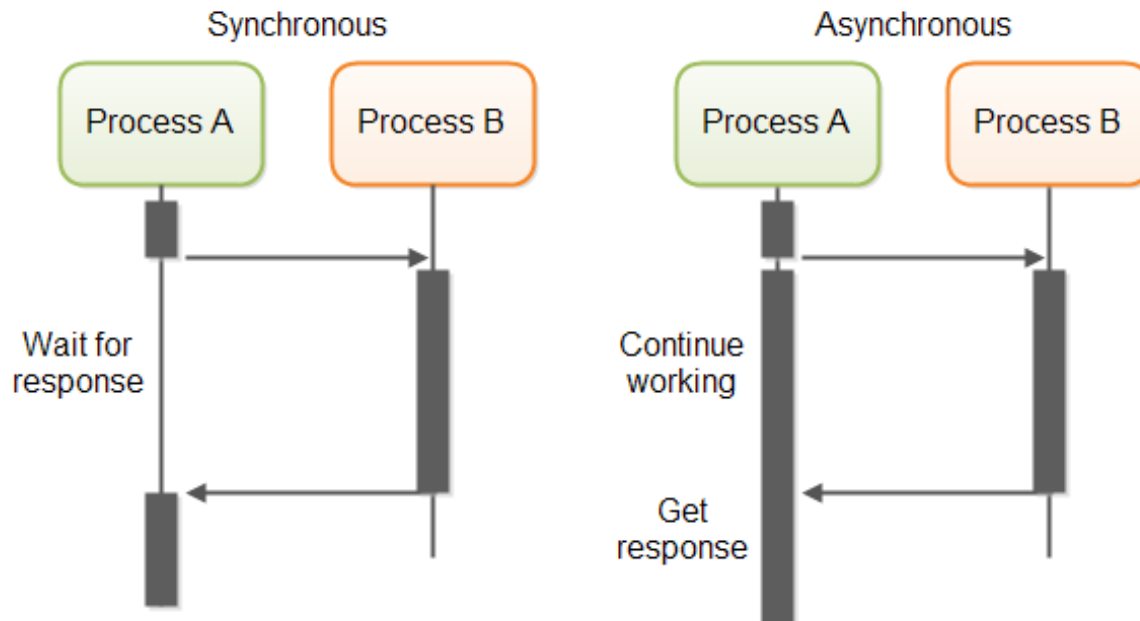


**Network on a Chip !**



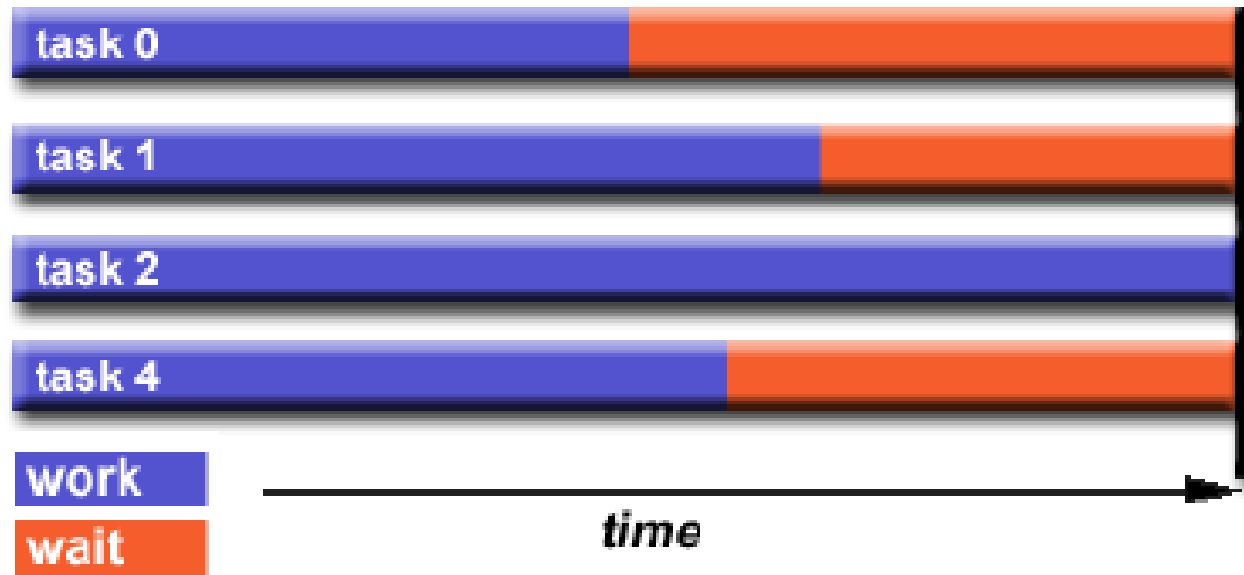
# Designing parallel programs - communication

- Most parallel applications require tasks to share data
  - **Synchronous vs. Asynchronous communication:**
    - Synchronous communication is referred to as blocking communication other work stops until the communication is completed.
    - Asynchronous communication is referred to as non-blocking since other work can be done while communication is taking place.



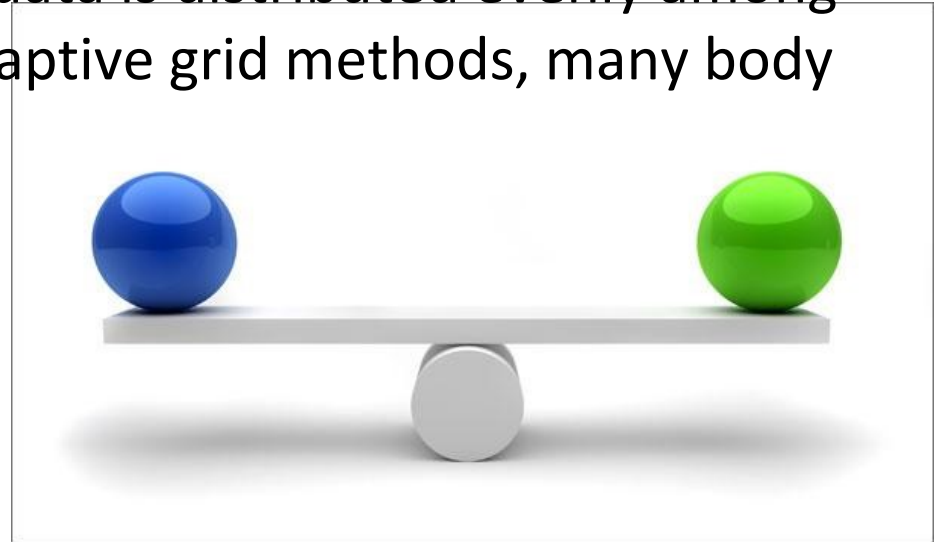
# Designing parallel programs – load balancing

- Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.

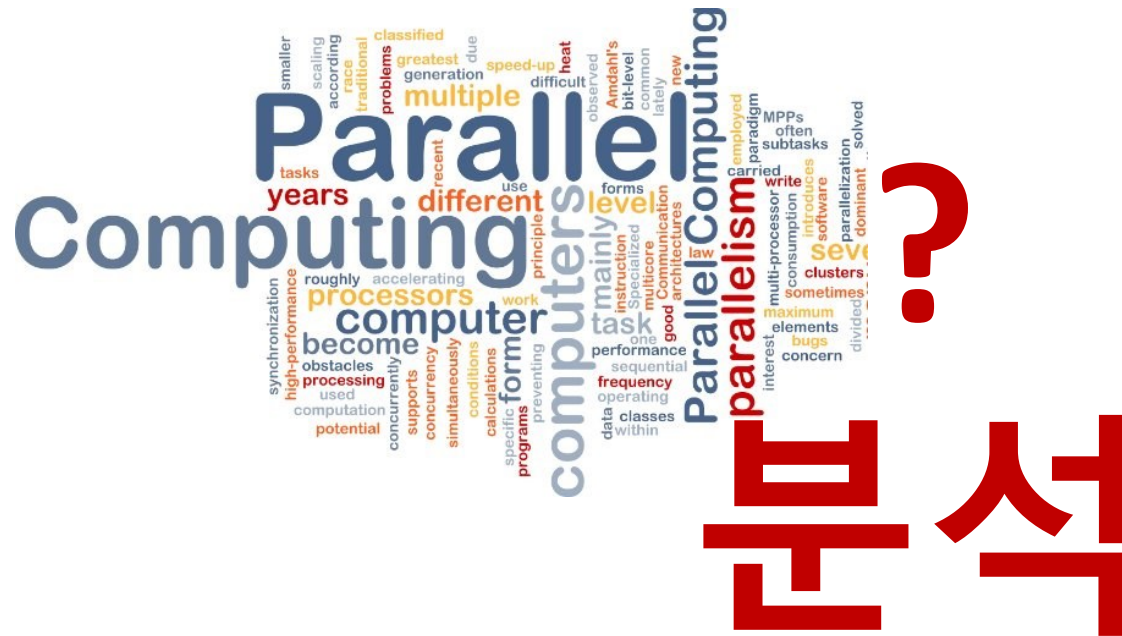


# Designing parallel programs – load balancing

- How to Achieve Load Balance?
  - **Equally partition the work** given to each task: For array/matrix operations equally distribute the data set among parallel tasks. For loop iterations where the work done for each iteration is equal, evenly distribute iterations among tasks.
  - **Use dynamic work assignment**: Certain class problems result in load imbalance even if data is distributed evenly among tasks (sparse matrices, adaptive grid methods, many body simulations, etc.).



# Analytic Measure for Parallel Algorithms



# Performance Metrics for Parallel Applications

- There are a number of metrics, the best known are:

- Speedup
- Efficiency
- Redundancy
- Utilization
- Quality



- Some laws/metrics that try to explain and assert the **potential performance of a parallel application**:
  - Amdahl Law
  - Gustafson-Barsis Law

# Speedup

- Speedup is a **measure of performance**. It measures the ration between the sequential execution time and the parallel execution time.

$$S(p) = \frac{T(1)}{T(p)}$$

$T(1)$  is the execution time with one processor

$T(p)$  is the execution time with  $p$  processors

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
$T(p)$	1000	520	280	160	100
$S(p)$	1	1,92	3,57	6,25	10,00

# Efficiency

- Efficiency is a **measure of the usage of the computational resources**. It measures the ration between performance and the resources used to achieve that performance.

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \times T(p)}$$

$S(p)$  is the speedup for  $p$  processors

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
$S(p)$	1	1,92	3,57	6,25	10,00
$E(p)$	1	0,96	0,89	0,78	0,63

# Redundancy

- Redundancy measures the **increase in the required computation** when using more processors. It measures the ration between the number of operations performed by the parallel execution and by the sequential execution.

$$R(p) = \frac{O(p)}{O(1)}$$

$O(1)$  is the total number of operations performed with 1 processor

$O(p)$  is the total number of operations performed with  $p$  processors

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
$O(p)$	10000	10250	11000	12250	15000
$R(p)$	1	1,03	1,10	1,23	1,50



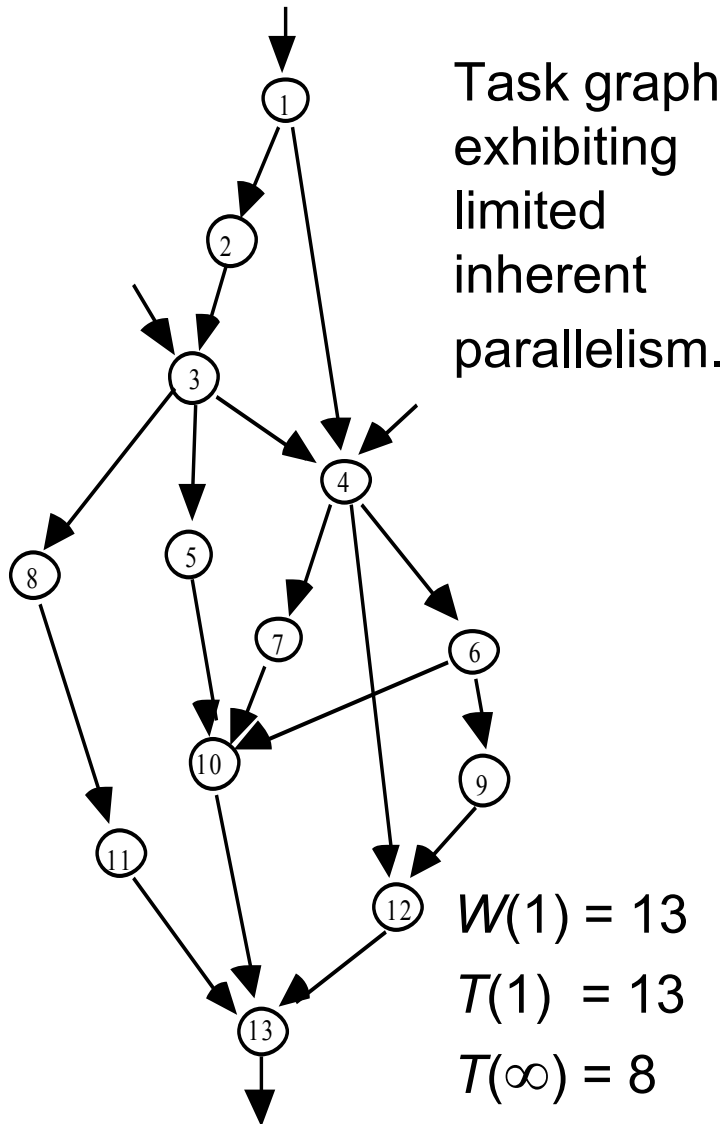
# Quality

- Quality is a measure of the relevancy of using parallel computing.

$$Q(p) = \frac{S(p) \times E(p)}{R(p)}$$

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
$S(p)$	1	1,92	3,57	6,25	10,00
$E(p)$	1	0,96	0,89	0,78	0,63
$R(p)$	1	1,03	1,10	1,23	1,50
$Q(p)$	1	1,79	2,89	3,96	4,20

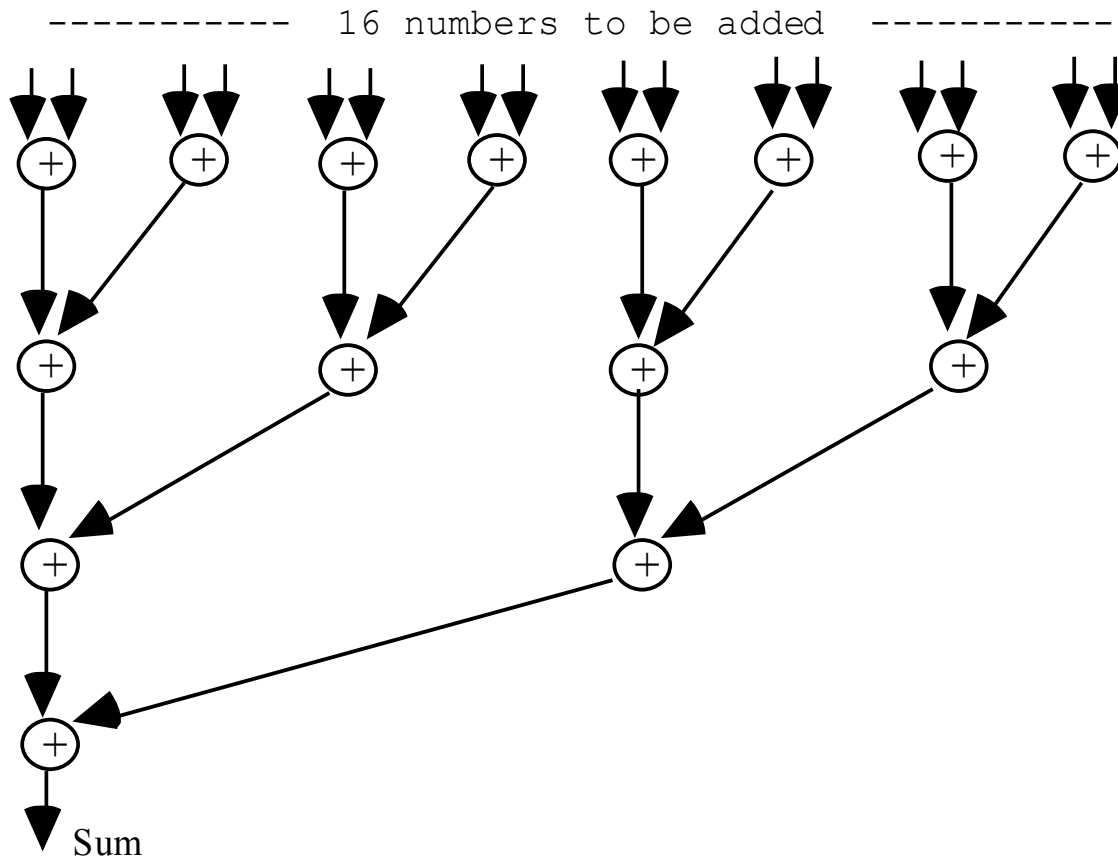
# Effectiveness of Parallel Processing



$p$	Number of processors
$W(p)$	Work performed by $p$ processors
$T(p)$	Execution time with $p$ processors $T(1) = W(1)$ ; $T(p) \leq W(p)$
$S(p)$	Speedup = $T(1) / T(p)$
$E(p)$	Efficiency = $T(1) / [p T(p)]$
$R(p)$	Redundancy = $W(p) / W(1)$
$Q(p)$	Quality = $T^3(1) / [p T^2(p) W(p)]$

# Reduction or Fan-in Computation

- Example: Adding 16 numbers, 8 processors, unit-time additions



*Zero-time communication*

$$E(8) = 15 / (8 \times 4) = 47\%$$

$$S(8) = 15 / 4 = 3.75$$

$$R(8) = 15 / 15 = 1$$

$$Q(8) = 1.76$$

*Unit-time communication*

$$E(8) = 15 / (8 \times 7) = 27\%$$

$$S(8) = 15 / 7 = 2.14$$

$$R(8) = 22 / 15 = 1.47$$

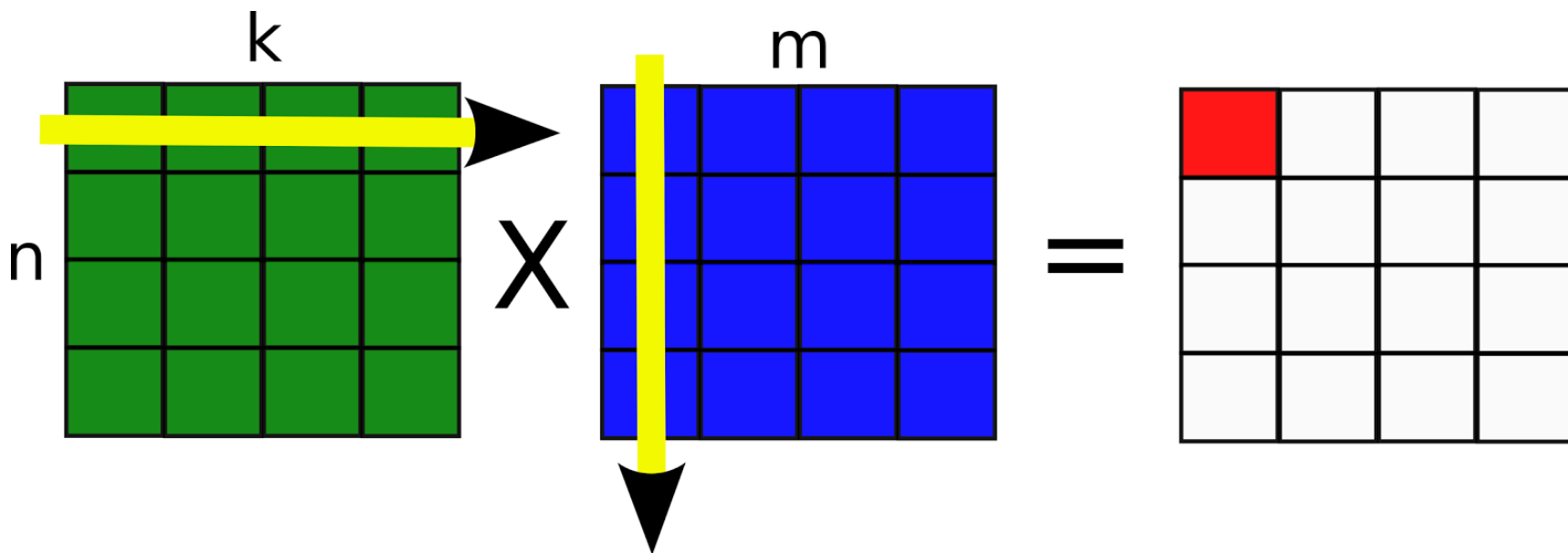
$$Q(8) = 0.39$$

Computation graph for finding the sum of 16 numbers .

# Matrix Multiplication ?

- What about  $n \times n$  matrix multiplication ?
  - Assume “+” or “ $\times$ ” take one time unit.

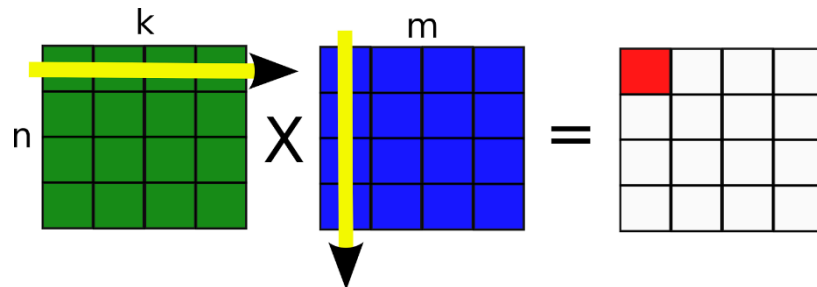
$$\sum_{i=0}^n \sum_{j=0}^n C[i, j] = \sum_{i=0}^n \sum_{j=0}^n \sum_{k=0}^n A[i, k] * B[k, j]$$



# Matrix Multiplication ?

- What about  $n \times n$  matrix multiplication ?
  - Assume “+” or “ $\times$ ” take one time unit.

$$\sum_{i=0}^n \sum_{j=0}^n C[i,j] = \sum_{i=0}^n \sum_{j=0}^n \sum_{k=0}^n A[i,k] * B[k,j]$$



– If we have 1 processor...

- $n \times n \times (n_{\times} + (n-1)_{+})$  ops =  $n^2 \times (2n - 1) = O(n^3)$

– If we have  $p$  processors =  $n$  ?

- $n \times (n_{\times} + (n-1)_{+})$  ops  $\sim O(n^2)$

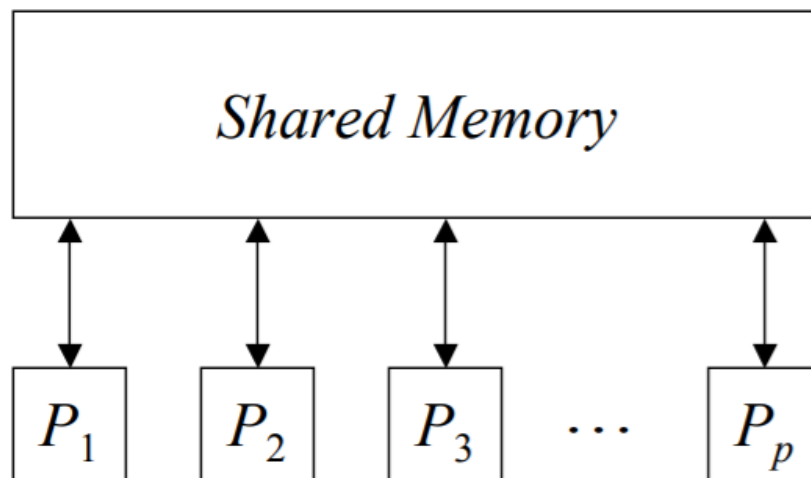
– If we have  $p$  processors =  $n^2$  ?

– If we have  $p$  processors =  $n^3$  ?

# Matrix Multiplication ?

- What about  $n \times n$  matrix multiplication
  - Assume “+” or “ $\times$ ” take one time unit.
  - What about **binary** matrix multiplication ?

## PRAM Architecture / Model





# PRAM

---

- Parallel Random Access Machine (PRAM)
  - Natural extension of RAM: each processor is a RAM
  - Processors operate synchronously
- Model is refined for concurrent read/write capability
  - Exclusive Read Exclusive Write (**EREW**)
  - Concurrent Read Exclusive Write (**CREW**)
  - Concurrent Read Concurrent Write (**CRCW**)
- CRCW PRAM
  - **Common** CRCW: all processors must write the same value
  - **Arbitrary** CRCW: one of the processors succeeds in writing
  - **Priority** CRCW: processor with highest priority succeeds in writing

# Matrix Multiplication ?

- What about  $n \times n$  matrix multiplication
  - Assume “+” or “ $\times$ ” take one time unit.
  - What about **binary** matrix multiplication ?

```
All processor_i_j_k ( $1 \leq i, j, k \leq n$ )  
  read A[i,k] to a;  
  read B[k,j] to b;  
  temp = a*b;  
  write '1' to C[i,j];  
  write temp to C[i,j]; // with CRCW
```



# Amdahl Law

- The computations performed by a parallel application are of 3 types:
  - $C(seq)$ : computations that can only be realized sequentially.
  - $C(par)$ : computations that can be realized in parallel.
  - $C(com)$ : computations related to **communication / synchronization / initialization**.
- Using these 3 classes, the speedup of an application can be defined as:

$$S(p) = \frac{T(1)}{T(p)} = \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p} + C(com)}$$

# Amdahl Law

- Since  $C(\text{com}) \geq 0$  then:

$$S(p) \leq \frac{C(\text{seq}) + C(\text{par})}{C(\text{seq}) + \frac{C(\text{par})}{p}}$$

- If ' $f$ ' is the fraction of the computation that can only be realized sequentially, then:

$$f = \frac{C(\text{seq})}{C(\text{seq}) + C(\text{par})} \quad \text{and} \quad S(p) \leq \frac{\frac{C(\text{seq})}{f}}{C(\text{seq}) + \frac{C(\text{seq}) \times \left(\frac{1}{f} - 1\right)}{p}}$$

# Amdahl Law

- Simplifying:

$$S(p) \leq \frac{\frac{C(seq)}{f}}{C(seq) + \frac{C(seq) \times \left(\frac{1}{f} - 1\right)}{p}}$$

$$\Rightarrow S(p) \leq \frac{\frac{1}{f}}{1 + \frac{\frac{1}{f} - 1}{p}} \quad \Rightarrow \quad S(p) \leq \frac{1}{f + \frac{1-f}{p}}$$

# Amdahl Law

- Let  $0 \leq f \leq 1$  be the computation fraction that can only be realized sequentially. The Amdahl law tells us that the maximum speedup that a parallel application can attain with  $p$  processors is:

$$S(p) \leq \frac{1}{f + \frac{1-f}{p}}$$

- The Amdahl law can also be used to determine **the limit of maximum speedup** that a determined application can achieve regardless of the number of processors used.



# Amdahl Law

- Suppose one wants to determine if it is advantageous to develop a parallel version of a certain sequential application. Through experimentation, it was verified that 90% of the execution time is spent in procedures that may be parallelizable. What is the maximum speedup that can be achieved with a parallel version of the problem executing on 8 processors?

$$S(p) \leq \frac{1}{0,1 + \frac{1 - 0,1}{8}} \approx 4,71$$

- And the limit of the maximum speedup that can be attained?

$$\lim_{p \rightarrow \infty} \frac{1}{0,1 + \frac{1 - 0,1}{p}} = 10$$

# Limitations of the Amdahl Law

- The Amdahl law ignores the cost with communication / synchronization operations associated to the introduction of parallelism in an application. For this reason, the Amdahl law can result in predictions **not very realistic** for certain problems.
- Consider a parallel application, with complexity  $O(n^2)$ , whose execution pattern is the following, where  $n$  is the size of the problem:
  - Execution time of the sequential part (input and output of data):
    - $18.000 + n$
  - Execution time of the parallel part:  $n^2 / 100$
  - Total communication/synchronization points per processor:  $\lceil \log n \rceil$
  - Execution time due to communication/synchronization ( $n=10.000$ ) :  $1000 * \lceil \log p \rceil + n/p$

# Limitations of the Amdahl Law

- What is the maximum speedup attainable?
- Using Amdahl law:

$$f = \frac{18.000 + n}{18.000 + n + \frac{n^2}{100}} \quad \text{and} \quad S(p) \leq \frac{18.000 + n + \frac{n^2}{100}}{18.000 + n + \frac{n^2}{p \times 100}}$$

- Using the speedup measure:

$$S(p) = \frac{18.000 + n + \frac{n^2}{100}}{18.000 + n + \frac{n^2}{p \times 100} + \lceil \log n \rceil \times \left( 10.000 \times \lceil \log p \rceil + \frac{n}{10} \right)}$$

# Limitations of the Amdahl Law

		1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
Amdahl law	$n = 10.000$	1	1,95	3,70	6,72	11,36
	$n = 20.000$	1	1,98	3,89	7,51	14,02
	$n = 30.000$	1	1,99	3,94	7,71	14,82
Speedup	$n = 10.000$	1	1,61	2,11	2,22	2,57
	$n = 20.000$	1	1,87	3,21	4,71	6,64
	$n = 30.000$	1	1,93	3,55	5,89	9,29

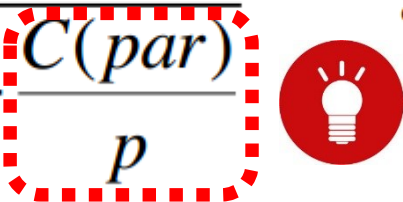


# Gustafson-Barsis Law

- Consider again the speedup measure defined previously:

$$S(p) \leq \frac{C(seq) + C(par)}{C(seq) + \frac{C(par)}{p}}$$

- If “f” is the fraction of the time spent in the sequential part, then (1-f) is the fraction of the time spent in the parallel part:

$$f = \frac{C(seq)}{C(seq) + \frac{C(par)}{p}} \quad \text{and} \quad (1-f) = \frac{\frac{C(par)}{p}}{C(seq) + \frac{C(par)}{p}}$$


# Gustafson-Barsis Law

- Then:

$$C(seq) = f \times \left( C(seq) + \frac{C(par)}{p} \right)$$

$$C(par) = p \times (1 - f) \times \left( C(seq) + \frac{C(par)}{p} \right)$$

- Simplifying:

$$S(p) \leq \frac{(f + p \times (1 - f)) \times \left( C(seq) + \frac{C(par)}{p} \right)}{C(seq) + \frac{C(par)}{p}}$$

$$\Rightarrow S(p) \leq f + p \times (1 - f) \quad \Rightarrow \quad S(p) \leq p + f \times (1 - p)$$

# Gustafson-Barsis Law

- Consider that a certain application executes in 220 seconds in 64 processors. What is the maximum speedup of an application knowing, by experimentation, that 5% of the execution time is spent on sequential computations.

$$S(p) \leq 64 + (0.05) \times (1 - 64) = 64 - 3.15 = 60.85$$
$$f \times 16,383 \leq 1,384$$

- Suppose that a certain company wants to buy a supercomputer with 16,384 processors to achieve a speedup of 15,000 in an important fundamental problem. What is the maximum fraction of the parallel execution time to attain the expected speedup

$$15,000 \leq 16,384 + f \times (1 - 16,384)$$
$$f \times 16,383 \leq 1,384$$

$$f \leq 0.084$$

# Preliminaries

- Law and Rule

- **Amdahl's law**: “Speedup vs. Parallelism” for **fixed** workload

$$Sp = \frac{1}{s + \frac{1-s}{N}}$$

- **Gustafson's law**: “Speedup vs. Parallelism” for **scalable** workload

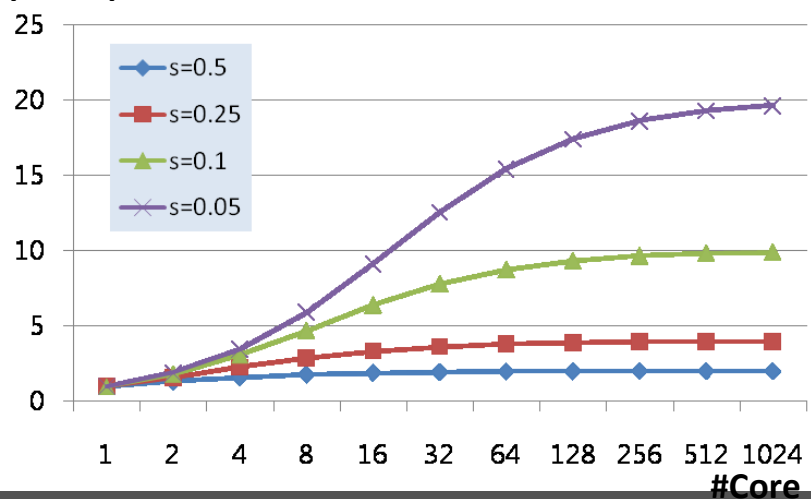
$$Sp = \frac{s + (1-s) \cdot N}{s + (1-s)} = s + (1-s) \cdot N$$

\* John Gustafson, “Reevaluating Amdahl's Law,” Communications of the ACM 31(5), 1988. pp. 532-533

# Amdahl .vs. Gustafson

- **Amdahl's law**: “Speedup vs. Parallelism” for **fixed** workload
- **Gustafson's law**: “Speedup vs. Parallelism” for **scalable** workload

Speedup



Speedup

