Introduction to Parallel Computing: Models, Concepts, and Algorithms

SC3260/5260 High-Performance Computing

Hongyang Sun
(hongyang.sun@vanderbilt.edu)

Vanderbilt University

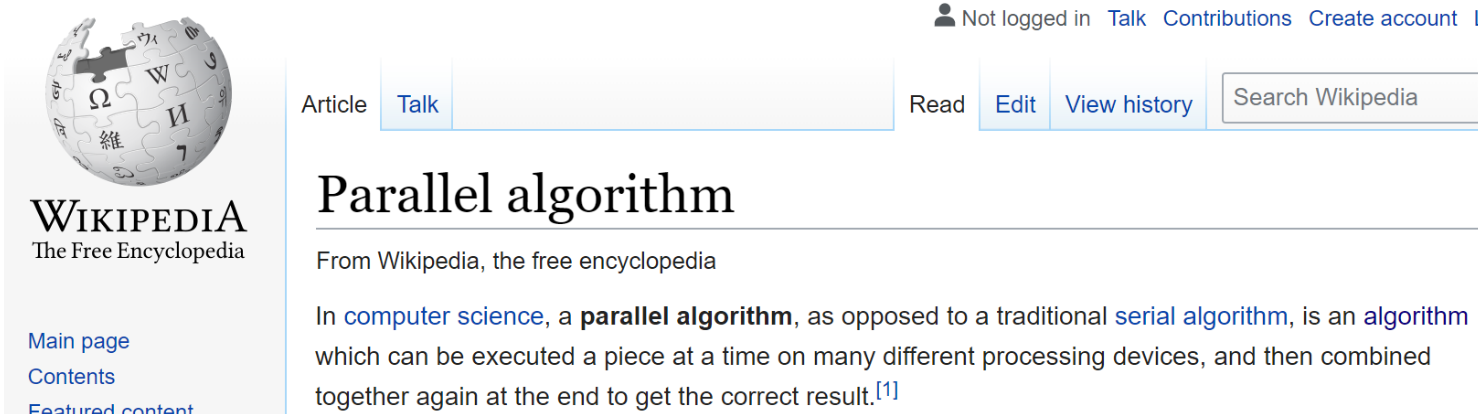
Spring 2020

Dr. Hongyang Sun
Research Assistant Professor
Department of EECS
https://my.vanderbilt.edu/hongyangsun/
Email: hongyang.sun@vanderbilt.edu
Office: FGH 382

Office Hours: Mon & Wed afternoons, or by appointment

- Topics I will cover in this course:
  - An introduction of parallel computing (2 lectures)
  - Shared-memory algorithms (4 lectures)
    - 1 programming assignment
  - Interconnection networks and communication patterns (1-2 lectures)
  - Distributed-memory algorithms (4 lectures)
    - 1 programming assignment

Why Parallel Computing/Algorithms?



Parallel algorithm

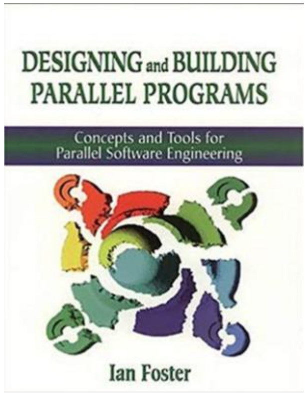
From Wikipedia, the free encyclopedia

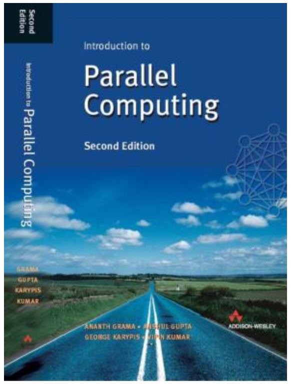
In computer science, a **parallel algorithm**, as opposed to a traditional serial algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result.[1]

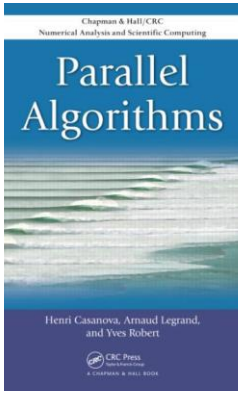
- Big-data applications need to run faster that could be enabled by efficient parallel algorithms.
- Most computing devices are parallel, and easy access to parallel computing resources.
- Availability of parallel computing tools, middleware, and programming languages.
- Many practical algorithms in reality are already running in parallel.

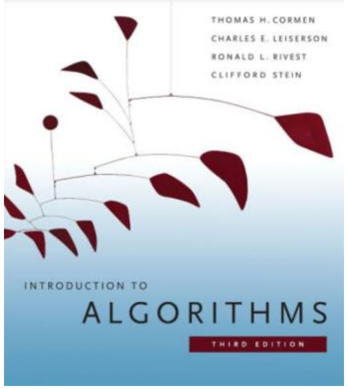
Reference Books

- I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, *Addison-Wesley Longman Publishing Co., Inc*.
- A. Grama, A. Gupta, G. Karypis, V. Kumar. Introduction to Parallel Computing (2nd Edition). *Addison-Wesley Professional.*
- H. Casanova, A. Legrand, Y. Robert. Parallel Algorithms. *Chapman & Hall/CRC*.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms (3rd Edition). *MIT Press and McGraw-Hill*.



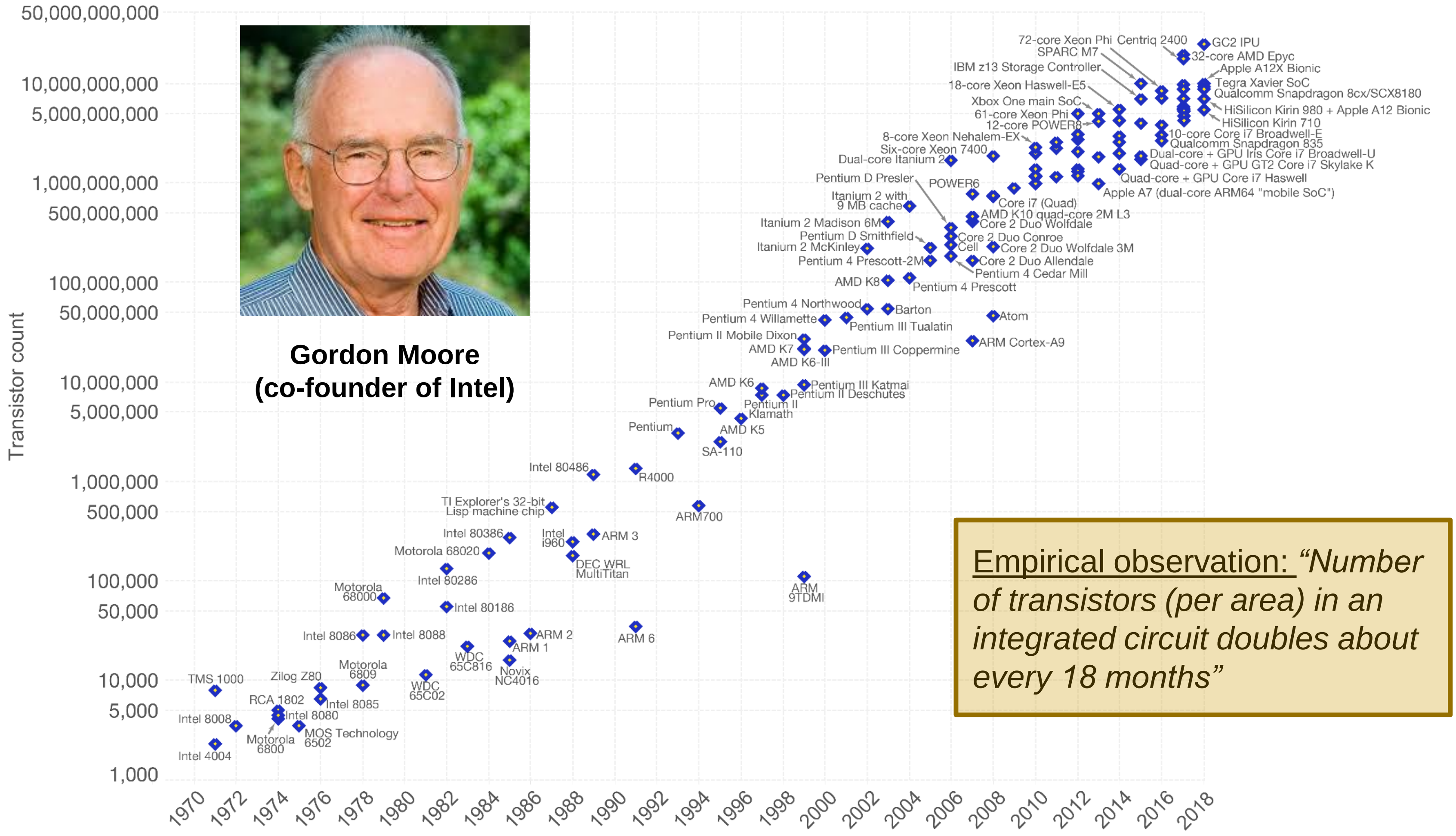






Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

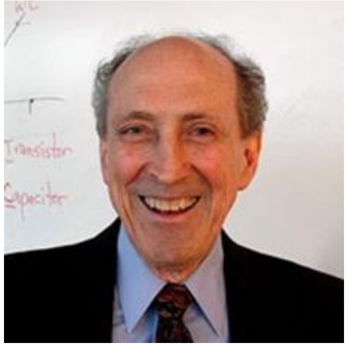


Gordon Moore
(co-founder of Intel)

Empirical observation: *"Number of transistors (per area) in an integrated circuit doubles about every 18 months"*

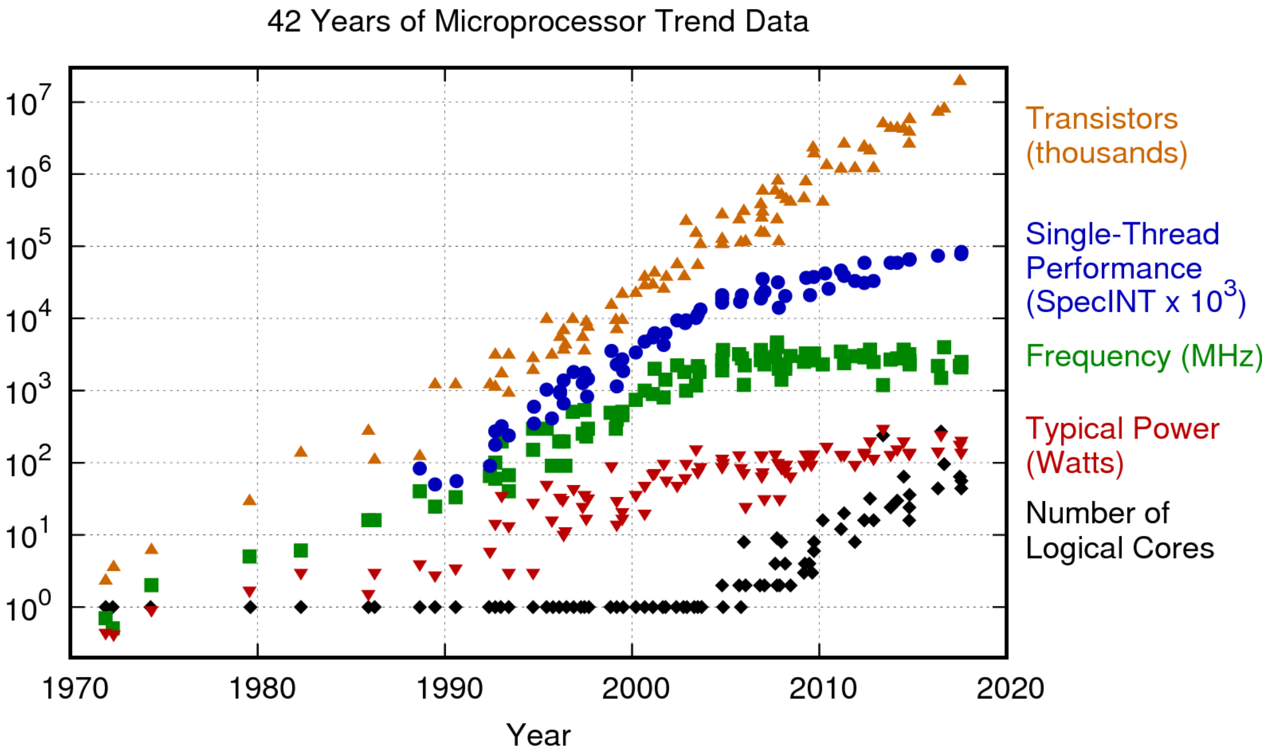
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Dennard's Scaling: Secret Behind Moore's Law



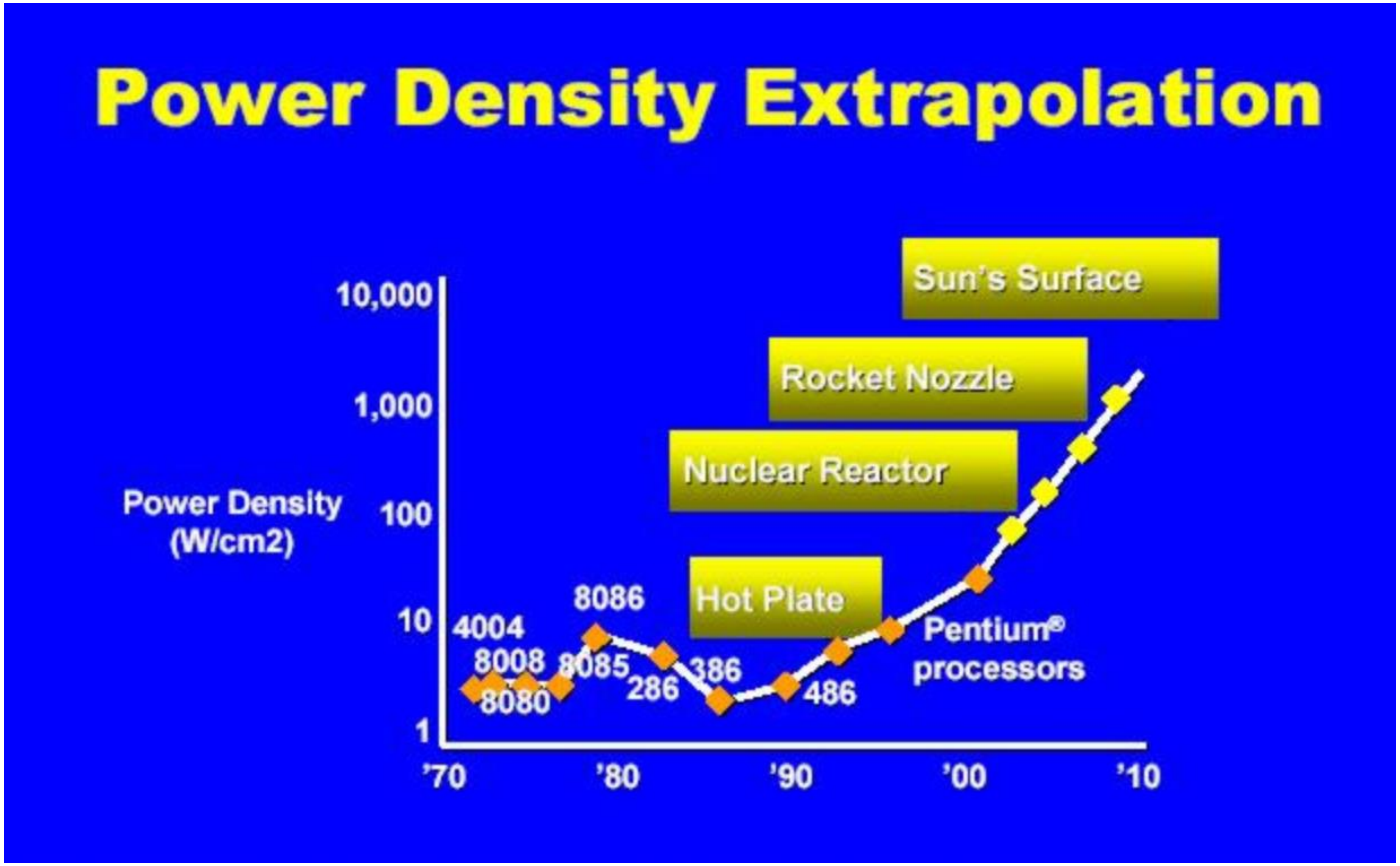
Robert Dennard,
(Inventor of DRAM)

"Power density of transistors (per area) stays constant"



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

- Combined with Moore's law: **performance per watt** doubles every 18 months!
- Ended around 2006 due to leakage current and thermal effect.



Power Density Extrapolation

Source: Patrick Gelsinger, Intel Developer's Forum, Intel Corporation, 2004.

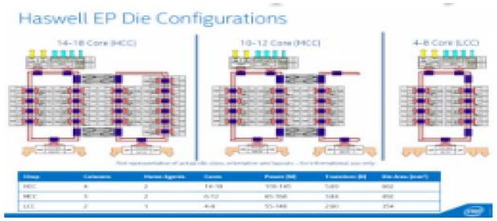
Going Multi-core Saves Power

- CMOS-based processors:
  - ✓ $\text{Power} \propto \text{Voltage} \times \text{Frequency}^2$
  - ✓ $\text{Frequency} \propto \text{Voltage}$
  - ✓ $\text{Power} \propto \text{Frequency}^3$

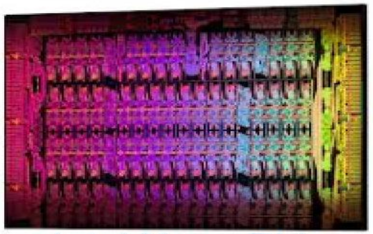
	Cores	Voltage	Freq.	Perf.	Power	Perf/Watt
Single-core	1	1	1	1	1	1
“Super-core”	1	1.5	1.5	1.5	3.38	0.44
Multi-core (x2)	2	0.75	0.75	1.5	0.84	1.78
Multi-core (x8)	8	0.75	0.75	6	3.36	1.78

Preferable to use multiple slow cores than a single fast core
(Same perf. with less power, or more perf. with same power)

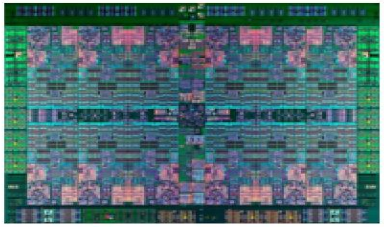
Everything is Multi-core Today!



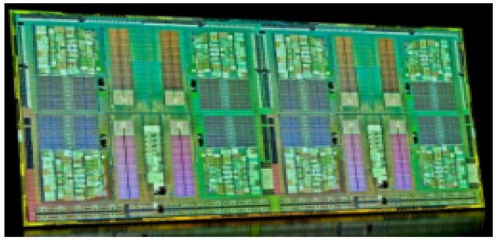
Intel Haswell (18 cores)



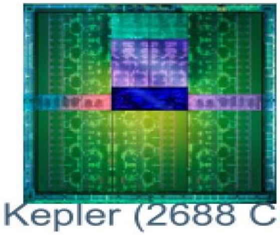
Intel Xeon Phi
(60 cores)



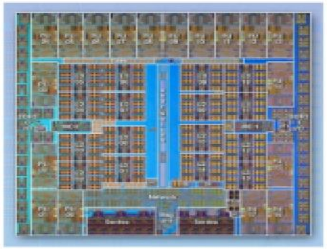
IBM Power 8 (12 cores)



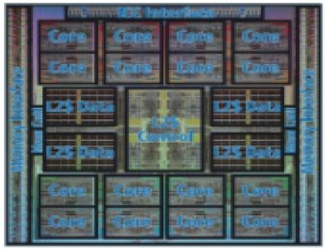
AMD Interlagos (16 cores)



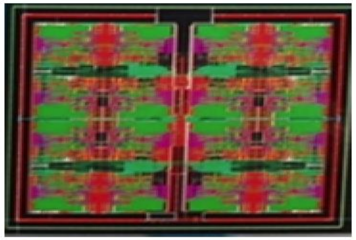
Nvidia Kepler (2688 Cuda cores)



IBM BG/Q (18 cores)



Fujitsu Venus (16 cores)



ShenWei (16 core)

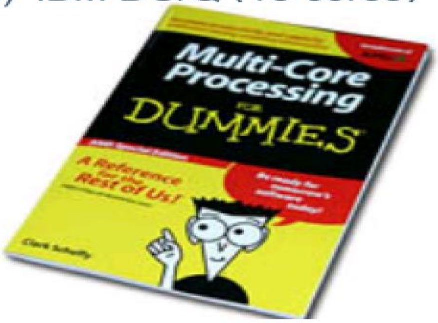


Figure courtesy of Jack Dongarra

Performance Measure

- Floating point operations per second (FLOPS)

$$\boxed{\mathrm{FLOPS} = \mathrm{nodes(sockets)} \times \frac{\mathrm{cores}}{\mathrm{sockets}} \times \mathrm{clock} \times \frac{\mathrm{FLOPs}}{\mathrm{cycle}}}$$

e.g., Intel Knights Landing (KNL) node, 68 cores, 1.4GHz (10^9), 2x AVX−512 (16 DP/cycle, 32 SP/cycle), ~3 TeraFLOPs (10^{12}) peak.

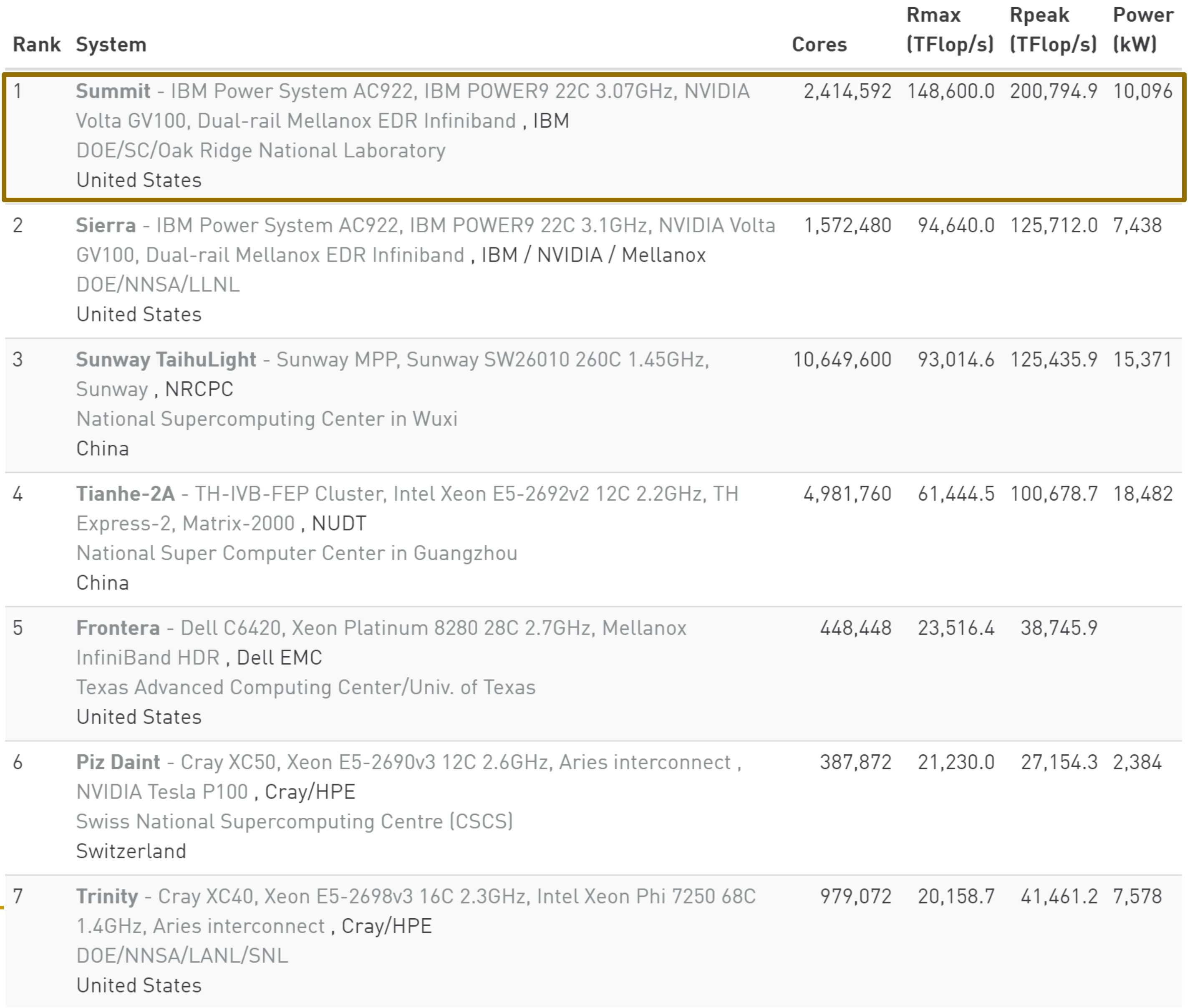
- Top500 (www.top500.org)
  - List of 500 most powerful computers in the world
    Updated twice per year (June & Nov.)
  - Benchmark: Rmax from LINPACK (HPL)

$$Ax = b \text{ (Solving a dense system of linear equations)}$$

Top500 List from Nov. 2019

# 1 Summit

- ~2.5 million cores
- ~200 PFLOPS (10^{15}) peak
- ~150 PFLOPS running Linpack
- ~10 MW of power!!!



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
7	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578

Exascale Computing: Next Milestone

- First exascale (10^{18} FLOPS) computer ("Aurora") in the US is expected in 2021 to be delivered to Argonne National Lab based on Intel hardware.

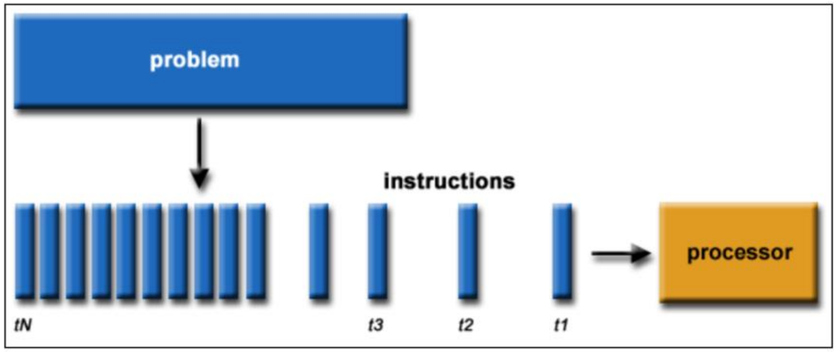
- Two other planned at Oak Ridge National Lab ("Frontier" based on AMD hardware) and Lawrence Livermore National Lab ("El Capitan" by Cray).
- Similar exascale projects exist in other countries and regions (e.g., EU, China, Japan, …).

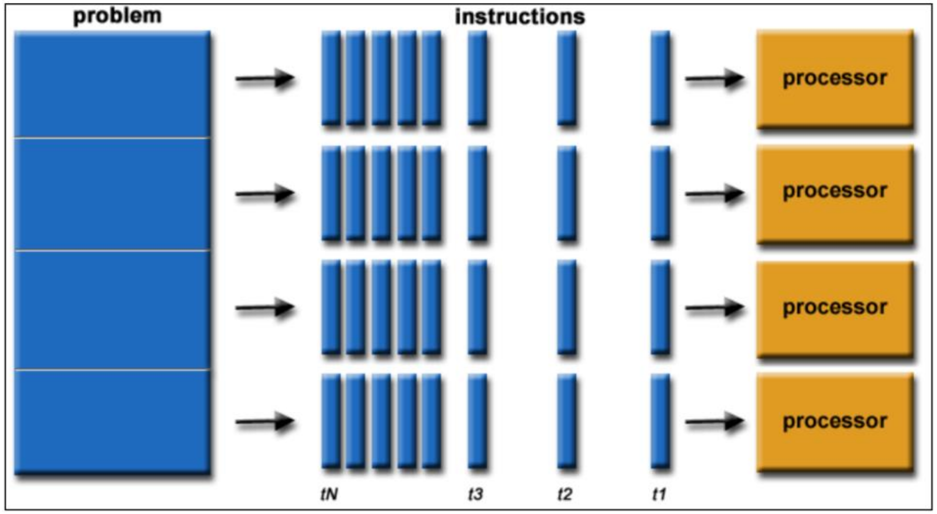
Other Benchmarks/Rankings

- **HPL** (High-Performance Linpack): models dense matrix computation (Ax=b) in many scientific computing applications (used for Top500 ranking).
- **HPCG** (High-Performance Conjugate Gradient): models some real-world applications that perform sparse matrix computations.
- **Graph500**: models some data-intensive applications by performing traversals on large graphs.
- **Green500**: measures the power-efficiency of supercomputers based on Linpack performance.

Parallel Computing Models

What is Parallel Computing?

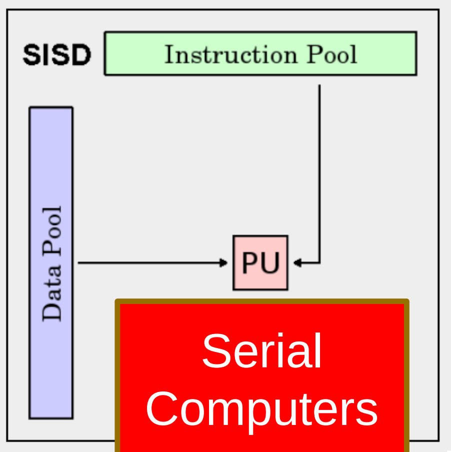




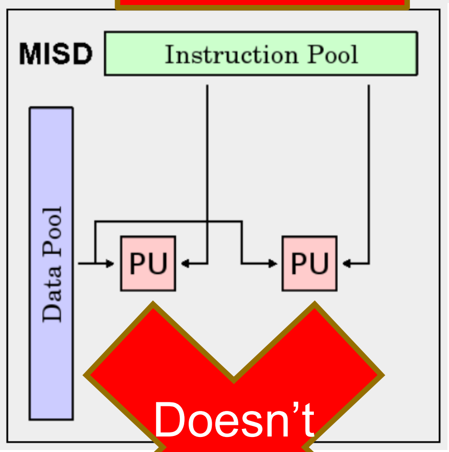
- **Serial Computing**
  - Problem broken into a series of instructions.
  - Instructions executed sequentially on a *single* processor.
  - Only 1 instruction executed at any time.
- **Parallel Computing**
  - Problem broken into several parts that can be solved concurrently.
  - Each part is broken into a series of instructions.
  - Instructions from different parts executed *simultaneously* by different processors.

Parallel Architecture Model

(by Control Structure)

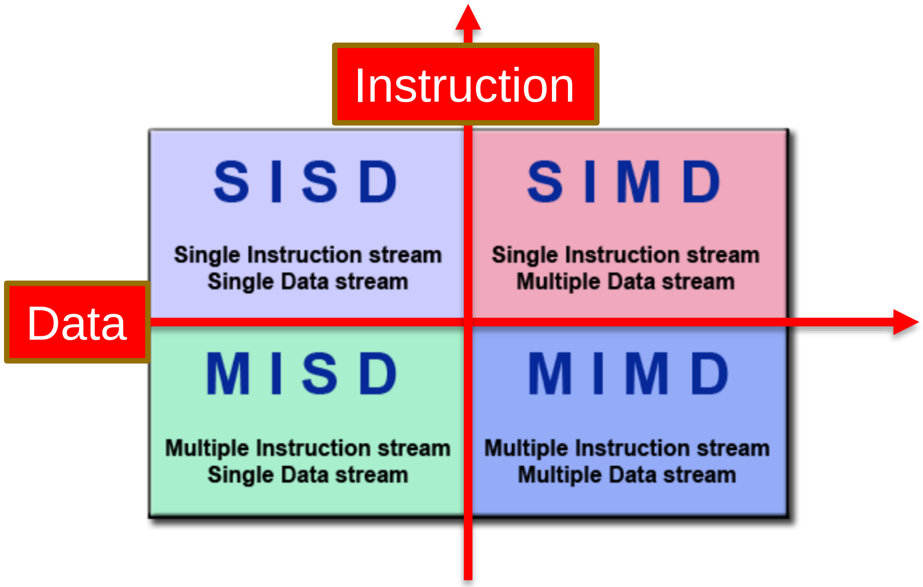


Serial Computers

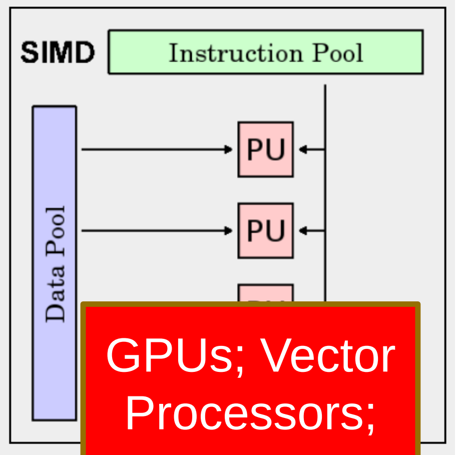


Doesn't Exist

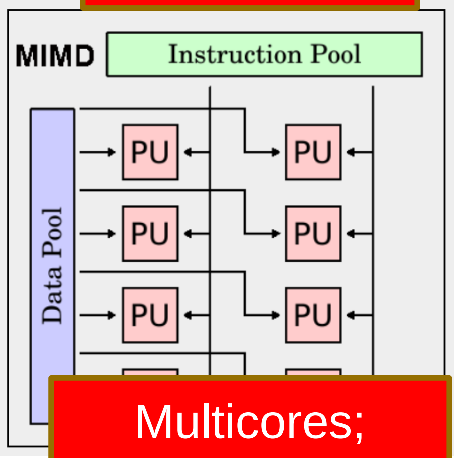
Flynn's Taxonomy



SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream



GPUs; Vector Processors;

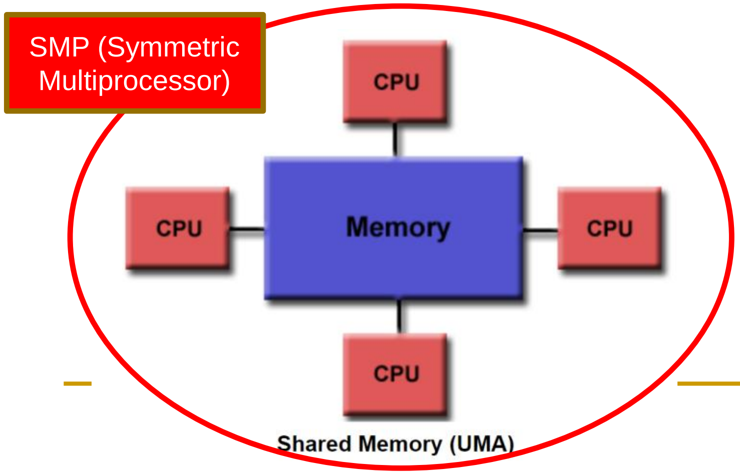


Multicores; Multiprocessors; Clusters

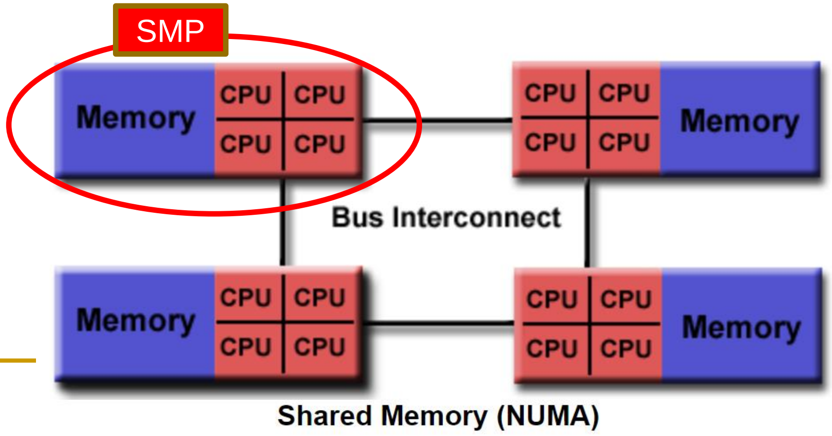
Parallel Architecture Model

(by Memory Structure)

- **Shared Memory**
  - All processors share all memory as global address space.
  - Update by one processor in a shared-memory location is visible to all other processors (*cache coherence* – handled by hardware).
  - Further classified into
    - *UMA (Uniform Memory Access)*: all processors have equal access latency to all memory locations.
    - *NUMA (Non-Uniform Memory Access)*: unequal access latency; slower across link.



Shared Memory (UMA)

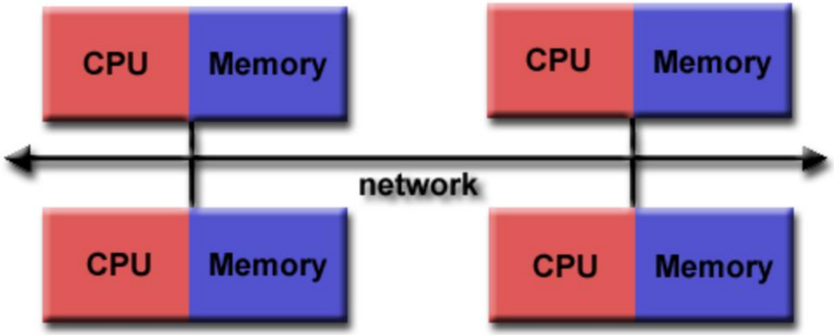


Shared Memory (NUMA)

Parallel Architecture Model

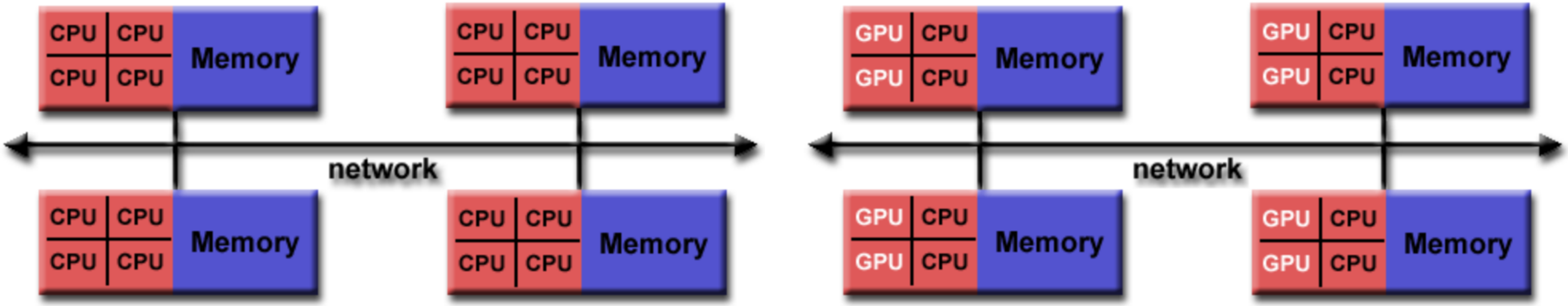
(by Memory Structure)

- **Distributed Memory**
  - Processors have own local memory (not shared), linked by an interconnection network (e.g., Ethernet, InfiniBand, Omni-Path).
  - Processors communicate by *passing messages* (*handled by programmer*).
  - Advantage: scalable (memory grows with processor).
  - Disadvantage: harder to program, larger communication cost.



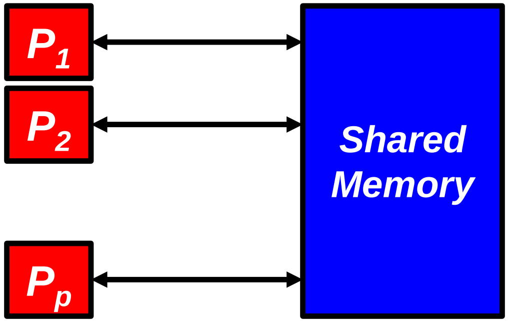
Parallel Architecture Model
(by Memory Structure)

- **Hybrid Distributed-Shared Memory**
  - Employed by most of today’s large supercomputers.
  - Distributed memory among multiple machines (nodes), and shared memory within a machine (cores, processors, GPUs).
  - Advantage: more scalable with low access latency.
  - Disadvantage: even harder to program.



Parallel Computation Model

- **PRAM (Parallel Random Access Machine)**
  - A parallel extension to RAM model for serial computing, used to analyze performance (time complexity) of parallel algorithms.
  - Models computations performed on shared-memory machines.
  - There are *p* identical processors with unbounded shared memory, and any processor can access any memory location in unit time. All processors can perform computation in parallel. Synchronization and communication overhead are ignored.



Parallel Computation Model

- **PRAM (Parallel Random Access Machine)**
  - Shared-memory conflicts (concurrent access of same location) handled by:
    - ***EREW****:* Exclusive Read and Exclusive Write (*Weakest model*).
    - ***CREW****:* Concurrent Read and Exclusive Write.
    - ***ERCW****:* Exclusive Read and Concurrent Write (*Never considered*).
    - ***CRCW****:* Concurrent Read and Concurrent Write (*Strongest model*).
  - Concurrent write (CW) is further handled by:
    - ***Common***: All processors must write a common value.
    - ***Arbitrary***: an arbitrary value written by one processor is selected.
    - ***Priority***: value written by the processor with highest priority is selected.
    - ***Combining/Reduction***: the value written is some combinations of all values attempted by all processors.
  - Not much difference: On p processors, 1 step of CRCW can be simulated by EREW in $O(\log p)$ steps.

Parallel Computation Model

- **PRAM (Parallel Random Access Machine)**
  - **An EREW algorithm**:

    Step 1: A *broadcast* of v to all procs. ($O(\log p)$)

    Step 2: All procs. *local search* concurrently ($O(n/p)$)

    Step 3: Combine results with a *reduction* ($O(\log p)$)

  - **A CREW algorithm:**

    Step 1: All procs. read value v *concurrently* ($O(1)$)

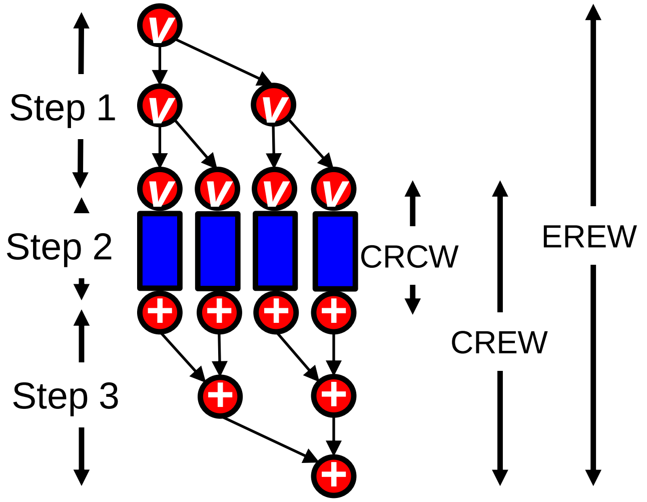
    Step 2-3: same as EREW

  - **A Combine-CRCW algorithm:**

    Step 1-2: same as CREW

    Step 3: all procs. write results *concurrently* ($O(1)$)

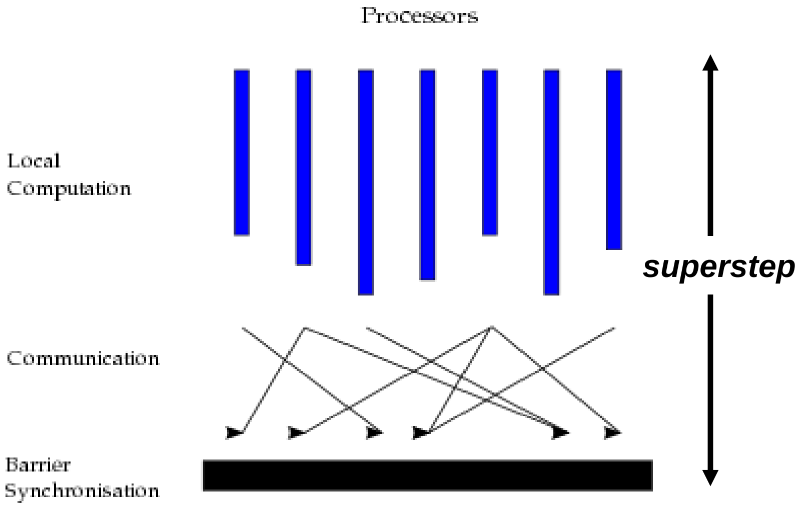
Example: consider a p-processor PRAM. The shared memory has an array of size n ($n > p$) and a value v. Find the number of occurrences of v in the array.



DAG model of computation

Parallel Computation Model

- **BSP (Bulk Synchronous Parallel)**
  - Different from the PRAM model by considering communication and synchronization costs.
  - A BSP computation has a sequence of supersteps, each consisting of:
    - **Concurrent computation**: local computations by all processors using local data.
    - **Communication**: processors communicate by exchanging data among each other.
    - **Barrier synchronization**: conclusion of a superstep and separation from next superstep.



Source: https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Parallel Computation Model

- **BSP (Bulk Synchronous Parallel)**
  - Cost of a BSP computation (with S supersteps).

$$\boxed{Cost = \sum_{s=1}^{S} \max_{i=1\ldots p} w_{s,i} + g \sum_{s=1}^{S} \max_{i=1\ldots p} h_{s,i} + Sl}$$

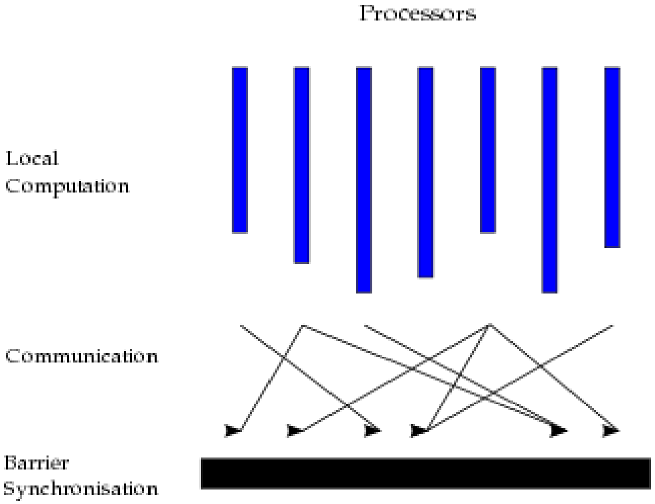
p: total number of processors.

$w_{s,i}$: cost of local computation of processor i in superstep s.

$h_{s,i}$: # msg sent or received by processor i in superstep s.

g: communication cost per message.

l: barrier synchronization cost.



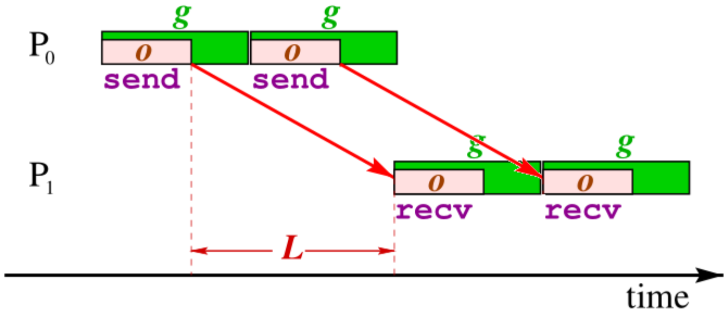
  - Applicable to both shared-memory and distributed-memory computations.

Parallel Computation Model

- **LogP Machine**
  - Different from BSP model by allowing asynchronous communication (of small messages) among processors (synchronous point-to-point).
  - Applicable to computations performed on distributed-memory machines.
  - Characterized by four parameters:
    - ***L:*** Latency of communication.
    - ***o:*** overhead of sending or receiving messages.
    - ***g:*** gap between successive sends or receives.
    - ***P:*** number of processors.

Example: sending two messages between two processors P_0 and P_1

Cost = $L + 2o + g$

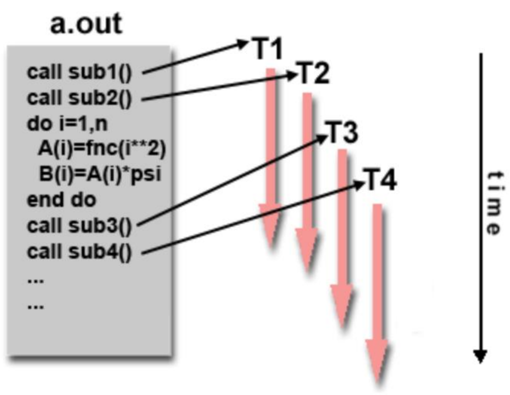


Parallel Programming Model
(by Process Interaction)

Parallel programming models exist as an abstraction above parallel architecture, and implemented by parallel programming languages or libraries.

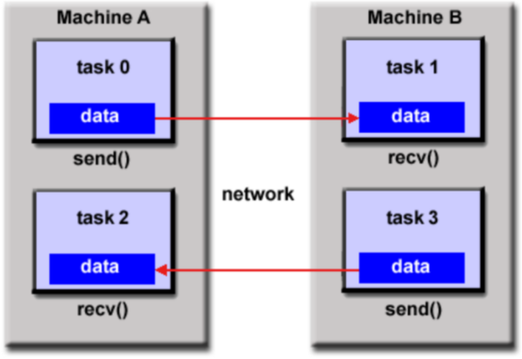
- **Shared-Memory/Threads Model**
  - Processes/threads share a global address space that they read and write to asynchronously.
  - Programming languages/libraries: POSIX Threads, OpenMP, Cilk, Threading Building Blocks (TBB), CUDA threads…
- **Distributed-Memory/Message-Passing Model**
  - Processes exchange data through passing messages, either point-to-point or collectively (e.g., broadcast, reduction).
  - Programming languages: MPI, Occam…

Threads Model



```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```

Message-Passing Model

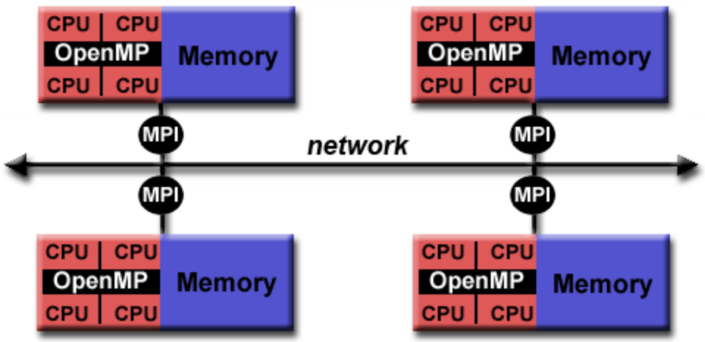


Parallel Programming Model

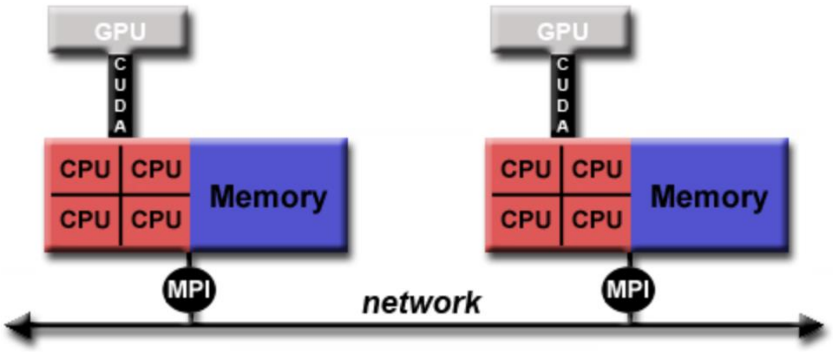
(by Process Interaction)

- **Hybrid Model**
  - Combining message-passing model (e.g., MPI) and threads model (e.g., Cilk, OpenMP or CUDA).

Hybrid Model (MPI + OpenMP)



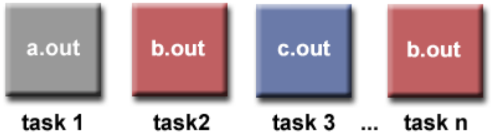
Hybrid Model (MPI + CUDA)



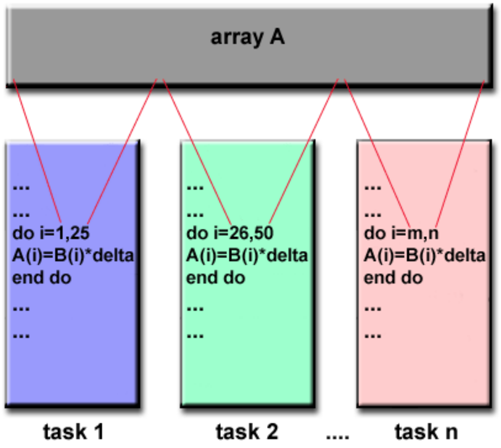
Parallel Programming Model
(by Problem Decomposition)

- **Task-Parallel Model**
  - Focuses on processes/threads that perform different tasks with distinct execution behaviors.
  - Processes/threads communicate via shared memory or message passing.
  - Multiple Programs Multiple Data (MPMD), mapping to the MIMD architecture model (Flynn's Taxonomy).
- **Data-Parallel Model**
  - Focuses on performing operations on a data set, typically organized in a regular structure (e.g., array).
  - A set of tasks work collectively on the data set, but each working on a different partition.
  - Single Program Multiple Data (SPMD), mapping to the SIMD architecture model (Flynn's Taxonomy).

Task-Parallel Model



Data-Parallel Model



```
...
...
do i=1,25
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=26,50
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=m,n
A(i)=B(i)*delta
end do
...
...
```

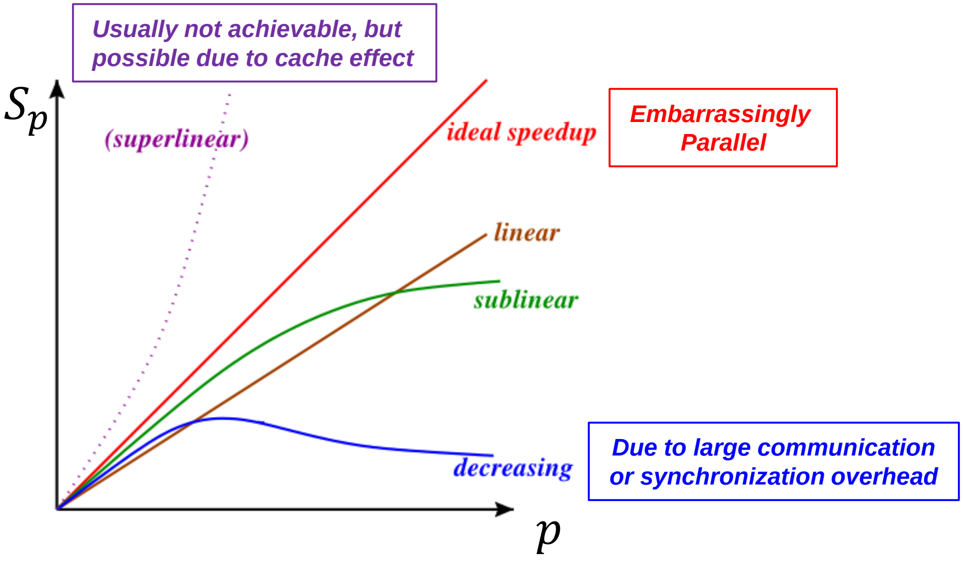
Performance Model

Given a parallel algorithm/program/application:

- T_1: serial execution time on a single processor.
- T_p: parallel execution time on p processors.

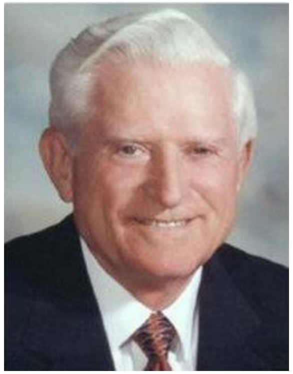
- **Speedup** : $S_p = \frac{T_1}{T_p}$

- **Efficiency**: $E_p = \frac{S_p}{p}$

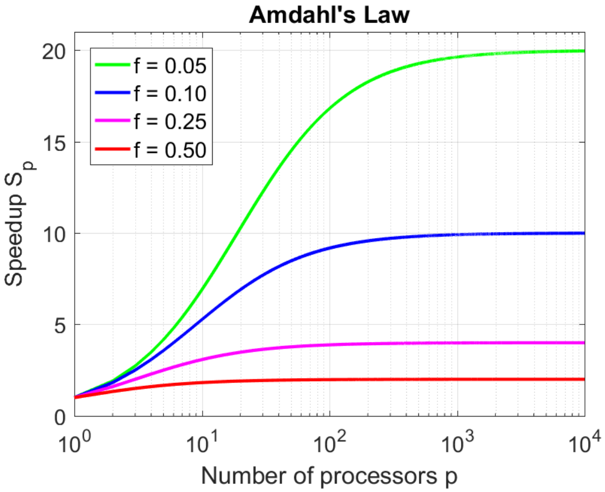


- **Cost** : $C_p = p \times T_p$
  - A parallel algorithm is cost-optimal if total parallel time = serial time, i.e., $C_p = T_1$

Amdahl’s Law: Fixed Problem Speedup



Gene Amdahl
(Computer Architect)



- Given a parallel program with *sequential fraction* f (i.e., $1 - f$ fraction can be perfectly parallelized), the speedup on p processors is:

$$\boxed{S_p = \frac{T_1}{T_p} = \frac{T_1}{fT_1 + \frac{(1-f)T_1}{p}} = \frac{1}{f + \frac{(1-f)}{p}} \leq \frac{1}{f}, \text{ for all } p}$$

(e.g., with only 5% sequential code, f = 0.05, max. achievable speedup is 20)

- Strong scaling: solving problems with fixed-size problems with more processors.
- Speedup is bounded by sequential fraction.
- *It’s hard to achieve strong scalability.*

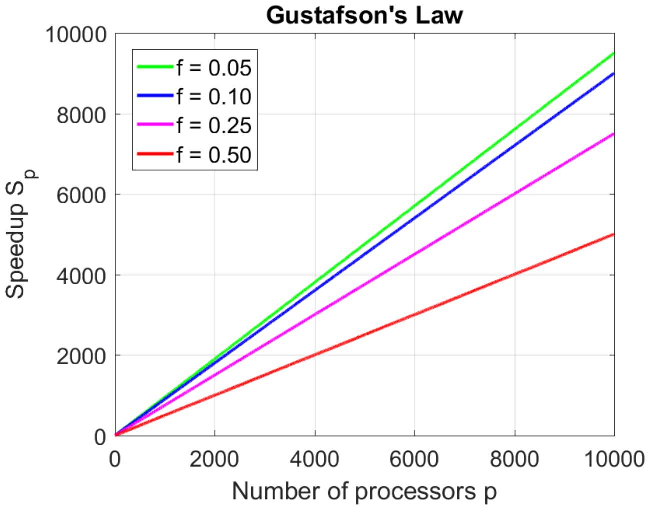
Gustafson's Law: Scaled Problem Speedup

John Gustafson (Computer Scientist)

- If a parallel program's parallelizable fraction $(1-f)$ grows with the number of processors, the speedup on p processors is:

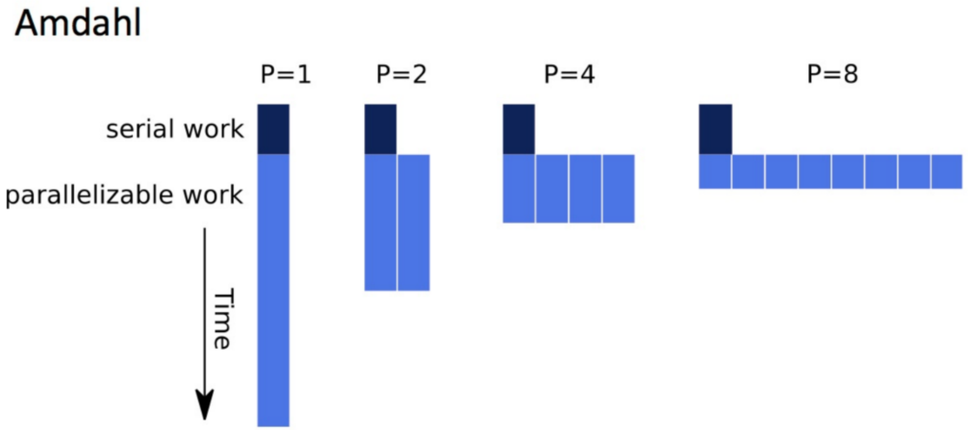
$$\boxed{S_p = \frac{T_1}{T_p} = \frac{fT_p + (1-f)T_p \cdot p}{T_p} = f + (1-f)p}$$

(Indeed, more processors are usually used to solve larger problems)

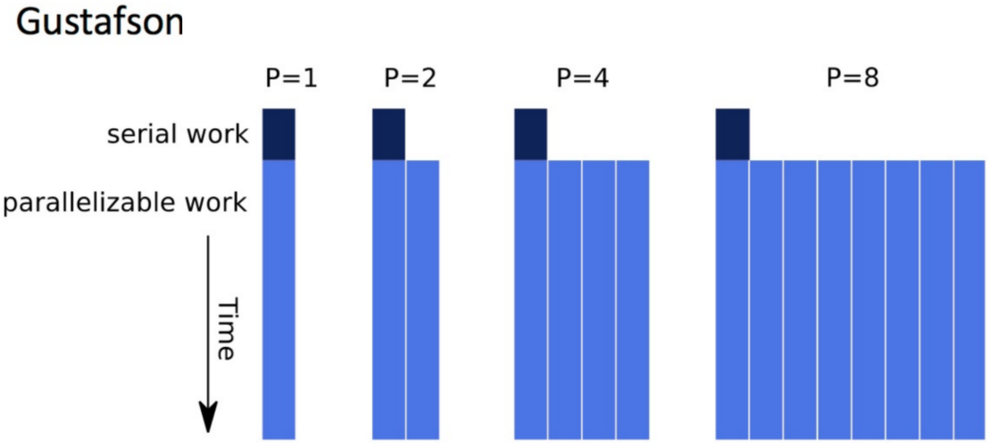


- Weak scaling: Applicable when solving larger problems with more processors.
- Speedup is more optimistic: linear with p
- *Weak scalability is relatively easy to achieve.*

Amdahl’s Law vs. Gustafson's Law



Strong Scaling:
Fixed problem size



Weak Scaling:
Increased problem size

Software Performance

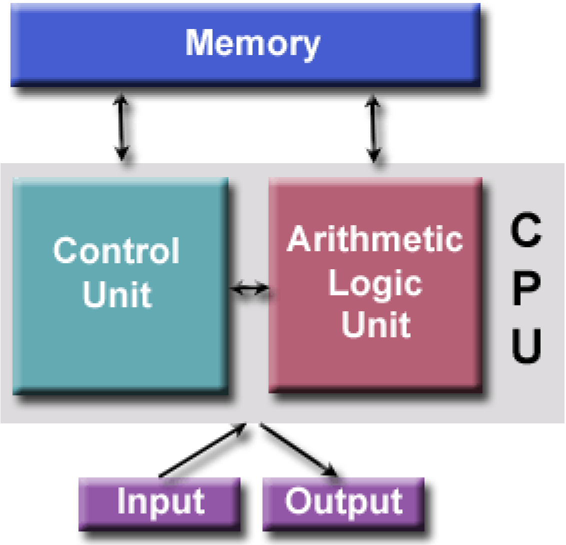
Improving Software Performance

- **Serial Optimization**
  - Programmer should optimize serial code before attempting parallelization.
- **Implicit Parallelism**
  - Hardware and compiler technology implicitly convert part of serial programs into parallel and execute them on multiple processing units of a processor (e.g., pipeline, superscalar, instruction-level parallelism)
  - No effort from programmer, but limited parallelism & performance gains
- **Explore Memory System**
  - Architectural features (e.g., cache, threads) to improve data-movement cost
  - Exploitation of locality (spatial and temporal) by hardware and programmer
- **Explicit Parallelism**
  - Programmer’s responsibility to identify explicit parallelism and write parallel code with help of parallel programming language and libraries to be run on multicore or distributed processors.
  - No simple “recipes”, but some design guidelines could help.

von Neumann Architecture

- Virtually, all computers follow the von Neumann architecture (a.k.a. stored-program computer), named after inventor John von Neumann.
- Consists of four main units
  - **Memory**: stores instructions and data.
  - **Control unit**: fetches instructions and data from memory, decodes instructions and coordinates computation.
  - **Arithmetic unit**: performs basic arithmetic operations.
  - **Input/Output**: Interface to external devices.

John von Neumann (Mathematician, Physicist, Computer Scientist)

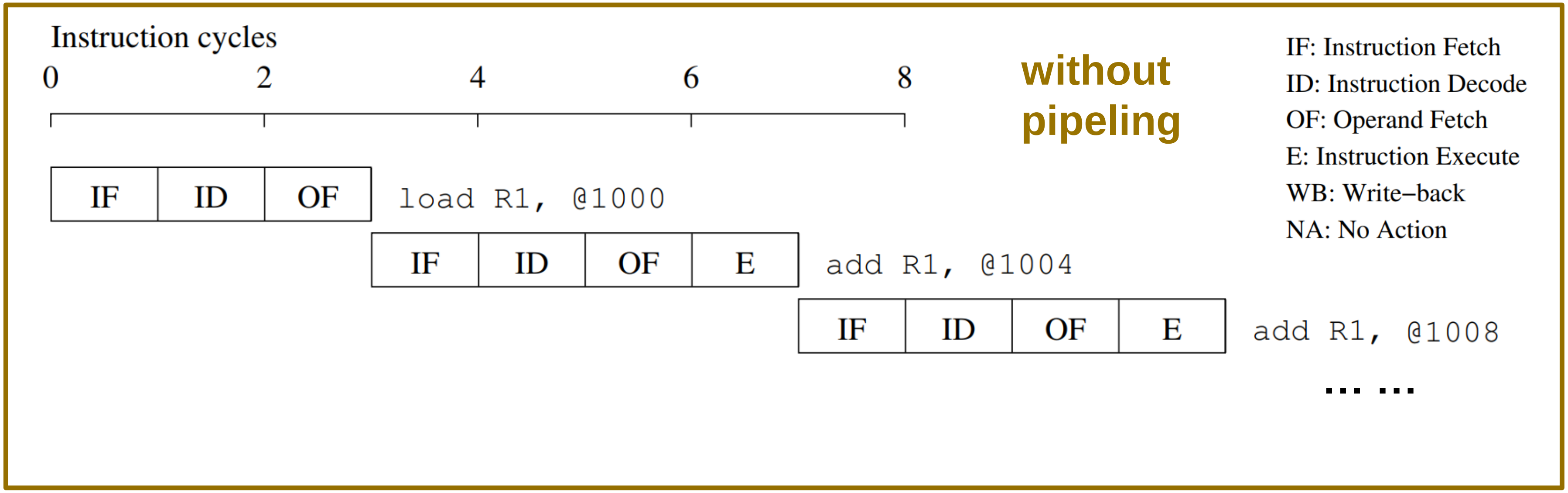


Implicit Parallelism – Pipelining

- Instructions in microprocessors are executed in different stages (e.g., fetch, decode, execute, store).
- Modern processors have 20+ stages to enable faster clock rate.

Ex. Add 4 numbers

```
1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
```



without pipeling

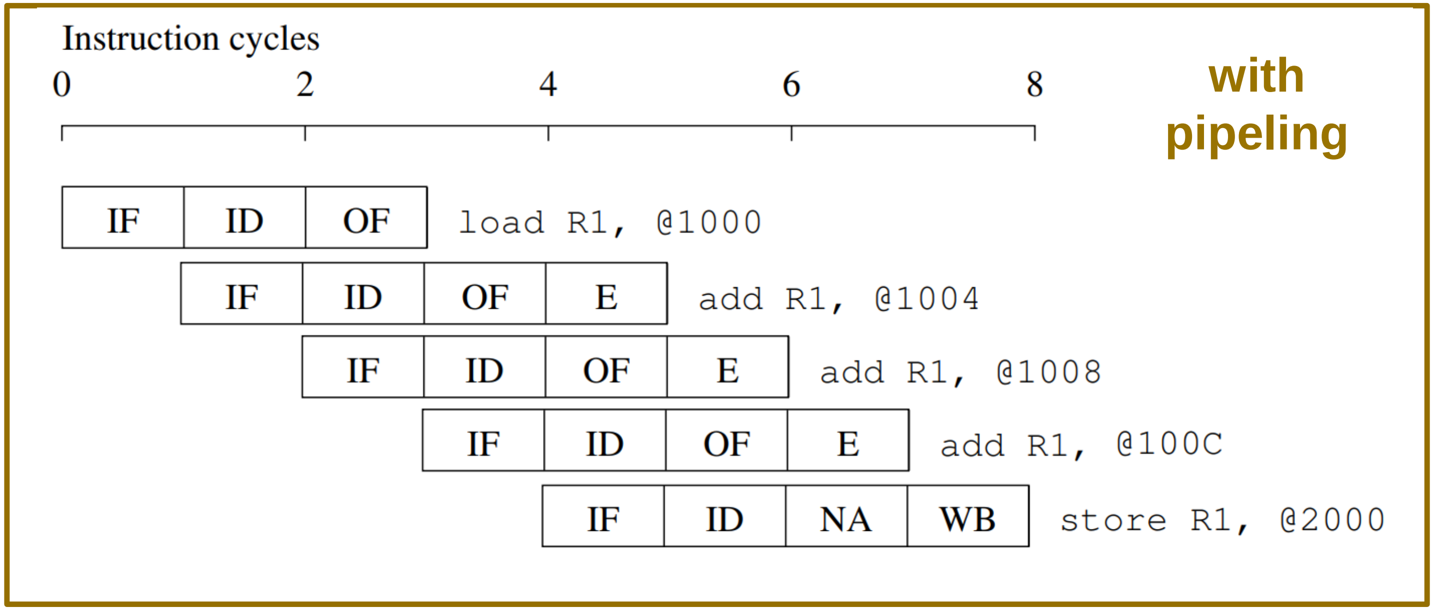
IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write–back
NA: No Action

Implicit Parallelism – Pipelining

- Pipelining (overlapping different stages) enables faster execution and increases the throughput of instruction execution.

Ex. Add 4 numbers

```
1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
```



Instruction cycles

0 2 4 6 8

IF ID OF load R1, @1000

IF ID OF E add R1, @1004

IF ID OF E add R1, @1008

IF ID OF E add R1, @100C

IF ID NA WB store R1, @2000

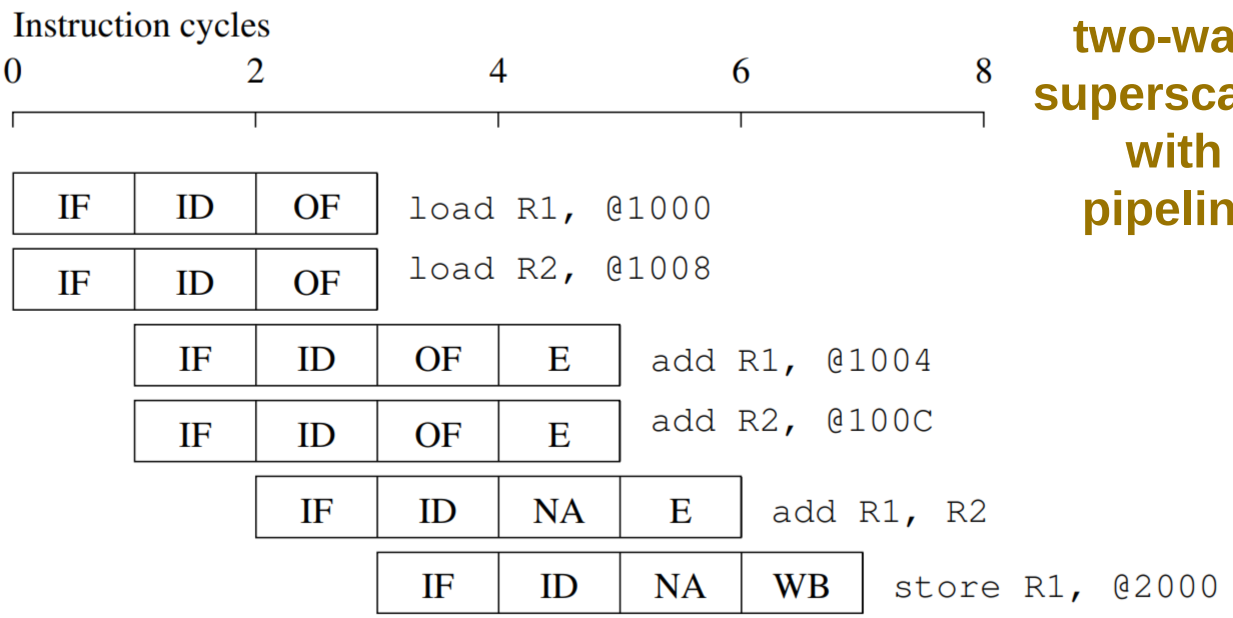
with pipeling

Implicit Parallelism – Superscalar

- Superscalar processors can execute more than one instructions per cycle by dynamically dispatching multiple instructions to different functional units.

Ex. Add 4 numbers

```
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```



IF ID OF load R1, @1000

IF ID OF load R2, @1008

IF ID OF E add R1, @1004

IF ID OF E add R2, @100C

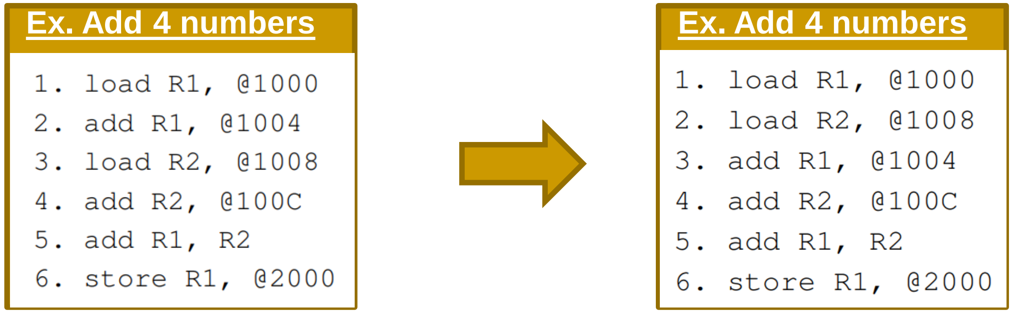
IF ID NA E add R1, R2

IF ID NA WB store R1, @2000

two-way superscalar with pipeling

Implicit Parallelism – Superscalar

- The ability of superscalar processors to detect and schedule concurrent instructions is critical. This can be achieved by instruction look-ahead and out-of-order execution.



Ex. Add 4 numbers

```
1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

Ex. Add 4 numbers

```
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

- Superscalar execution may not always be possible due to:
  - **Data dependency**: An instruction uses the data from the result of another instruction.
  - **Resource dependency**: Two instructions compete for the same functional unit.
  - **Branch dependency**: Execution of an instruction depends on the control flow (e.g., conditional jump).

Implicit Parallelism – VLIW

- Very long instruction word (VLIW) processors relies on compiler to resolve dependencies and issue more than one instructions at a time by static code analysis.
- Complexity is moved from hardware issue units to software (compiler), which is easier since compiler has a larger context and can use a variety of code transformation techniques.
  - Loop unrolling
  - Register renaming
  - Branch prediction
  - Speculative execution
- *Overall, instruction-level parallelism (ILP) provides very limited parallelization opportunities, and more explicit parallelism needs to be expressed.*