

Parallel Programming

ACSE-6: Lecture 1

Adriana Paluszny

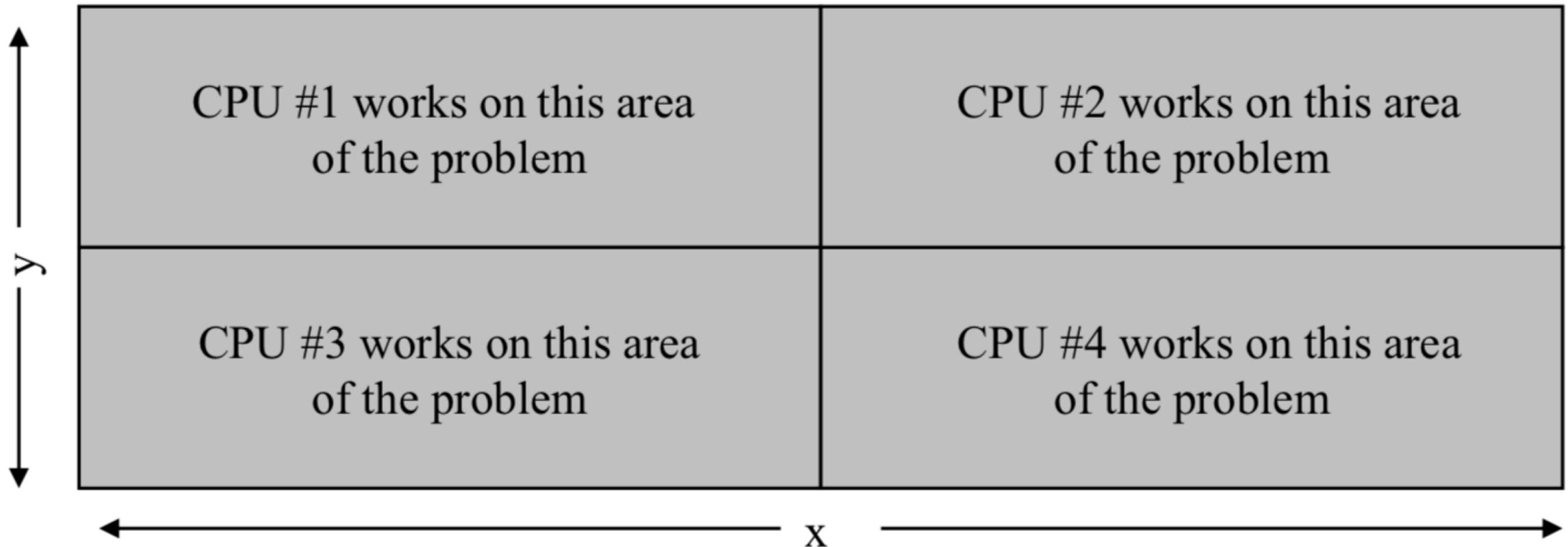
Royal Society University Research Fellow

What is Parallel Programming?

- Using more than one processor or computer to complete a task
- Each processor works on its section of the problem (functional parallelism)
- Each processor works on its section of the data (data parallelism)
- Processors can exchange information

What is Parallel Programming?

Grid of Problem to be solved

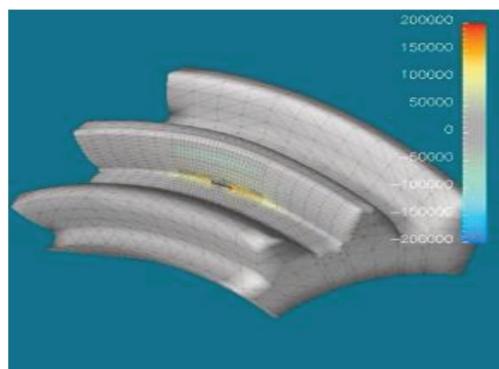


Why Do Parallel Programming?

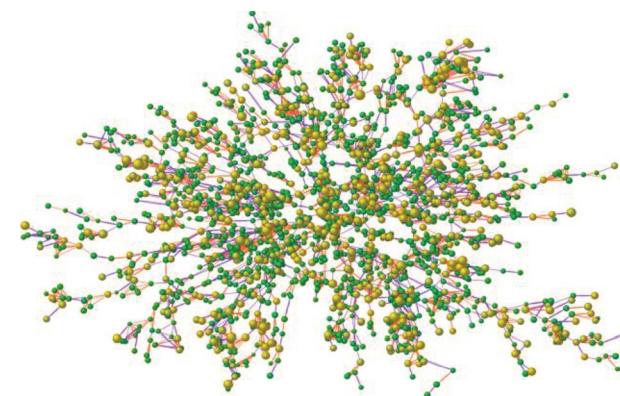
- Limits of single CPU computing
 - performance
 - available memory
- Parallel computing allows one to:
 - solve problems that don't fit on a single CPU
 - solve problems that can't be solved in a reasonable time
- We can solve...
 - larger problems
 - faster
 - more cases

Areas that rely on parallel programming

Finite Element Analysis



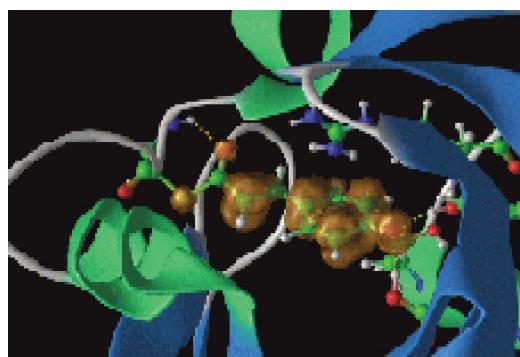
Social Media Networks



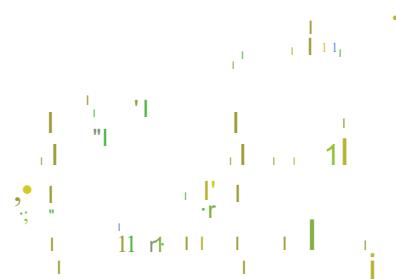
Climate Modeling



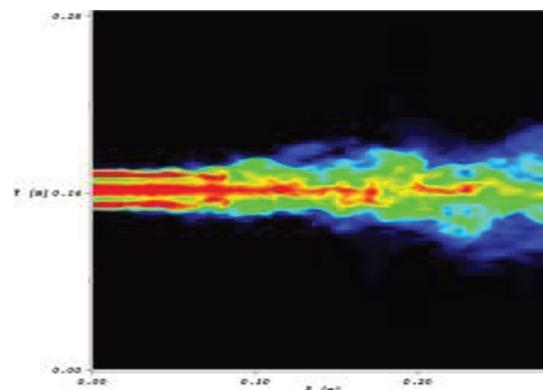
Protein Folding



Gene Sequencing



Fluid Dynamics



Parallel Computing: what for? (1)

- Planetary and galactic orbits
- Weather and ocean patterns
- Tectonic plate drift
- Rush hour traffic in Paris
- Automobile assembly line
- Daily operations within a business
- Building a shopping mall
- Ordering a hamburger at the drive through.

“grand challenge” problems:

- weather and climate
- chemical and nuclear reactions
- biological, human genome
- geological, seismic activity
- mechanical devices - from prosthetics to spacecraft
- electronic circuits
- manufacturing processes

Parallel Computing: what for? (2)

- Applications that require the processing of large amounts of data in sophisticated ways, include:
 - parallel databases, data mining
 - oil exploration
 - web search engines, web based business services
 - computer-aided diagnosis in medicine
 - management of national and multi-national corporations
 - advanced graphics and virtual reality, particularly in the entertainment industry
 - networked video and multi-media technologies
 - collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.

Why Parallel Computing? (1)

- This is a legitimate question! Parallel computing is complex on any aspect!
- The primary reasons for using parallel computing:
 - Save time - wall clock time
 - Solve larger problems
 - Provide concurrency (do multiple things at the same time)

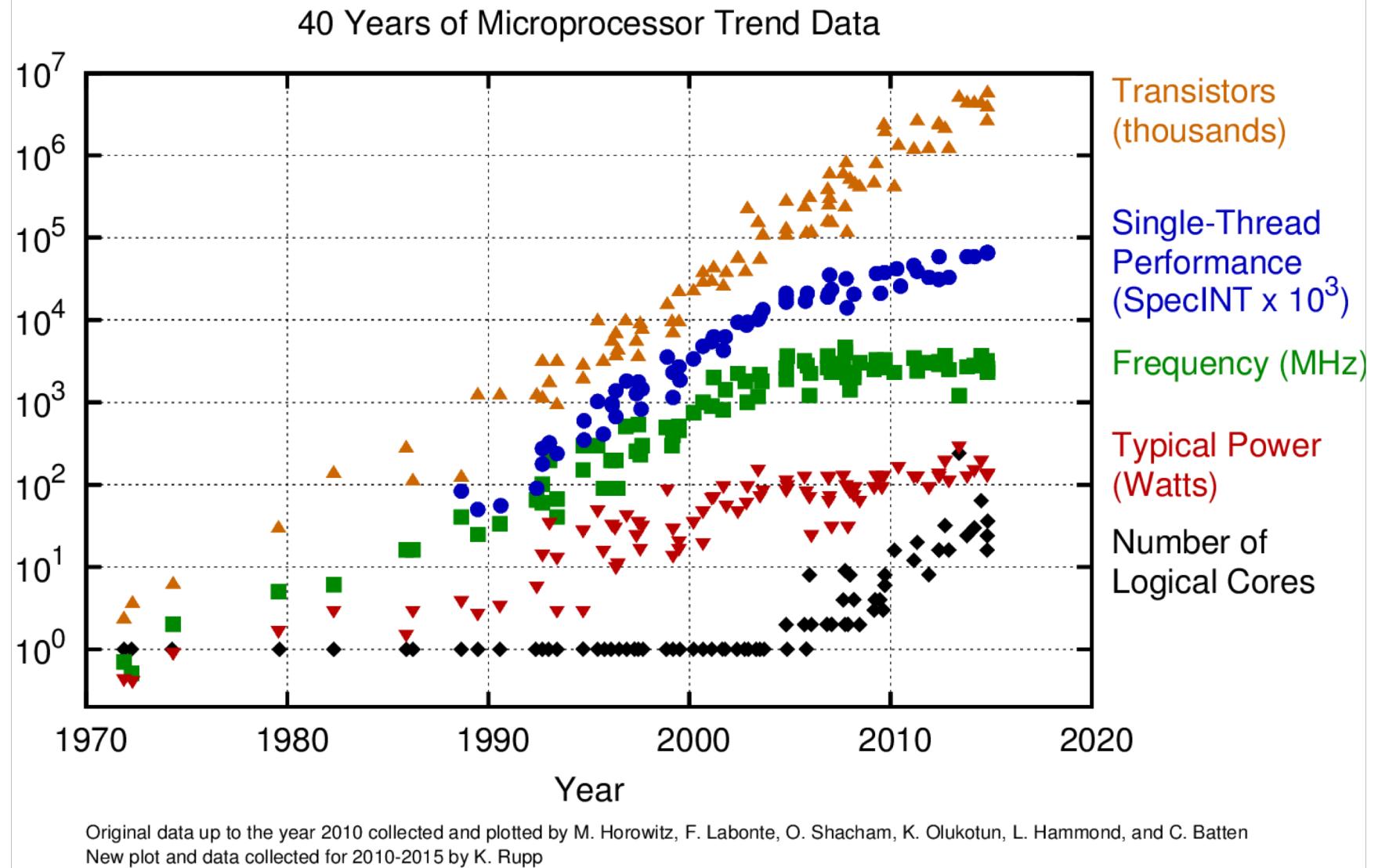
Why Parallel Computing? (2)

- Other reasons might include:
 - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
 - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
 - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Limitations of Serial Computing

- **Limits to serial computing** - both physical and practical reasons pose significant constraints to simply building ever faster serial computers.
- **Transmission speeds** - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
- **Limits to miniaturization** - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- **Economic limitations** - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

The future



Terminology

- **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
 - A single thread of execution operating on multiple data items simultaneously
 - Multiple threads of execution in a single executable
 - Multiple executables all working on the same problem
 - Any combination of the above
- **task**: is the name we use for an instance of an executable. Each task has its own virtual address space and may have multiple threads.

Terminology

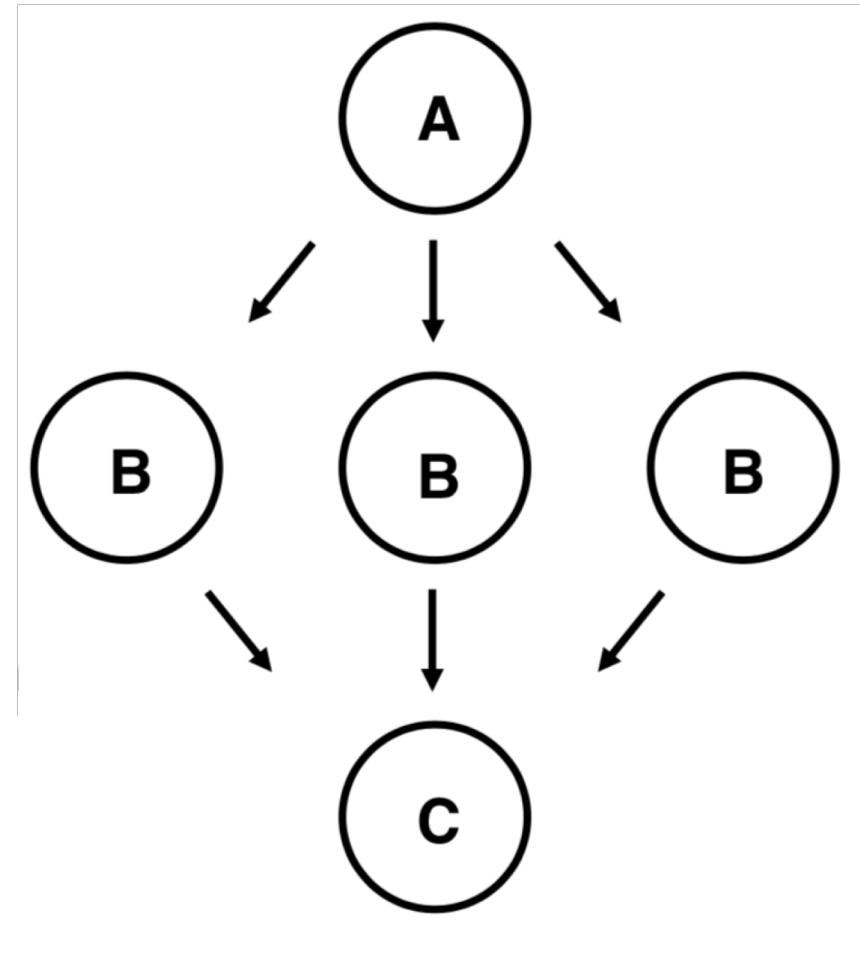
- **node**: a discrete unit of a computer system that typically runs its own instance of the operating system
- **core**: a processing unit on a computer chip that is able to support a thread of execution; can refer either to a single core or to all of the cores on a particular chip
- **cluster**: a collection of machines or nodes that function in someway as a single resource.
- **grid**: the software stack designed to handle the technical and social challenges of sharing resources across networking and institutional boundaries. grid also applies to the groups that have reached agreements to share their resources.

Types of parallelism

- Data parallelism
- Functional parallelism
- Task parallelism
- Pipeline parallelism

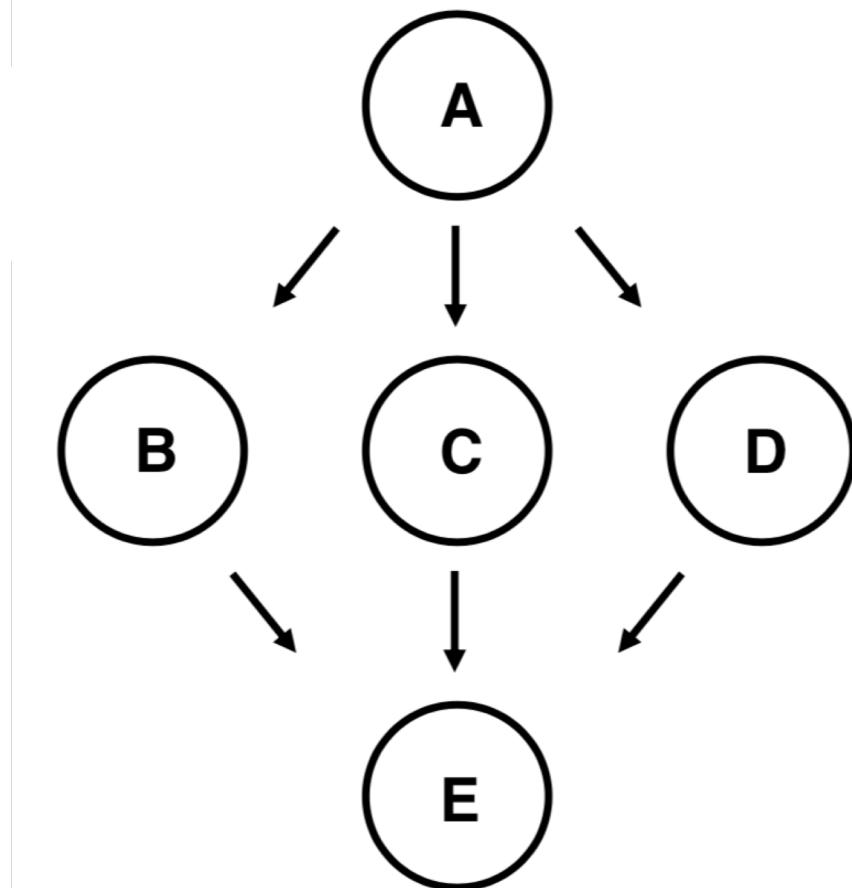
Data parallelism

- Definition: each process does the same work on unique and independent pieces of data
- Examples:
 - 2 brothers mow the lawn
 - 8 farmers paint a barn
- Usually more scalable than functional parallelism
- Can be programmed at a high level with OpenMP, or at a lower level using a message-passing library like MPI.



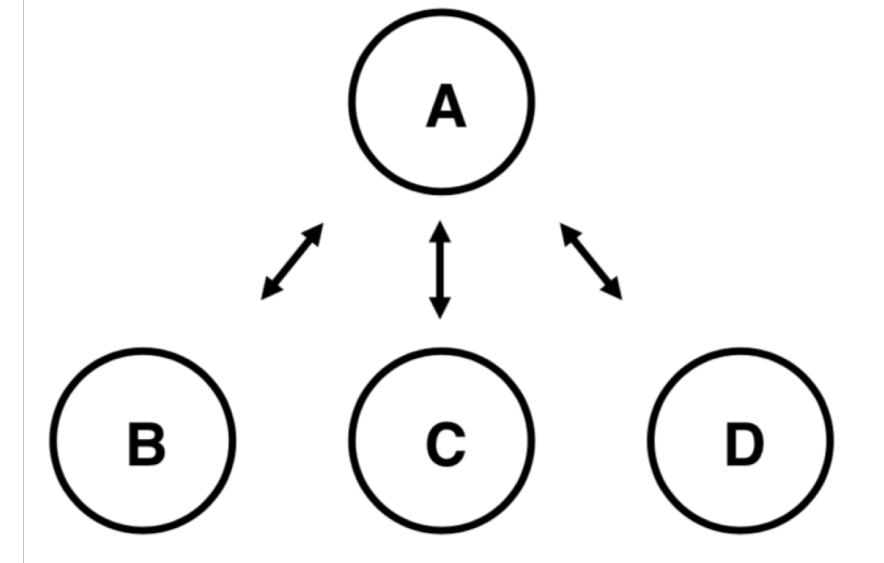
Functional parallelism

- Definition: each process performs a different "function" or execute a different code sections that are independent.
- Commonly programmed with message passing libraries
- Examples:
 - 2 brothers do yard work (1 edges & 1 mows)
 - 8 farmers build a barn



Task parallelism

- Definition: each process perform the same functions but do not communicate with each other, only with a "Master" Process. are often called "Embarrassingly Parallel".
- Examples:
 - Independent Monte Carlo Simulations
 - ATM Transactions

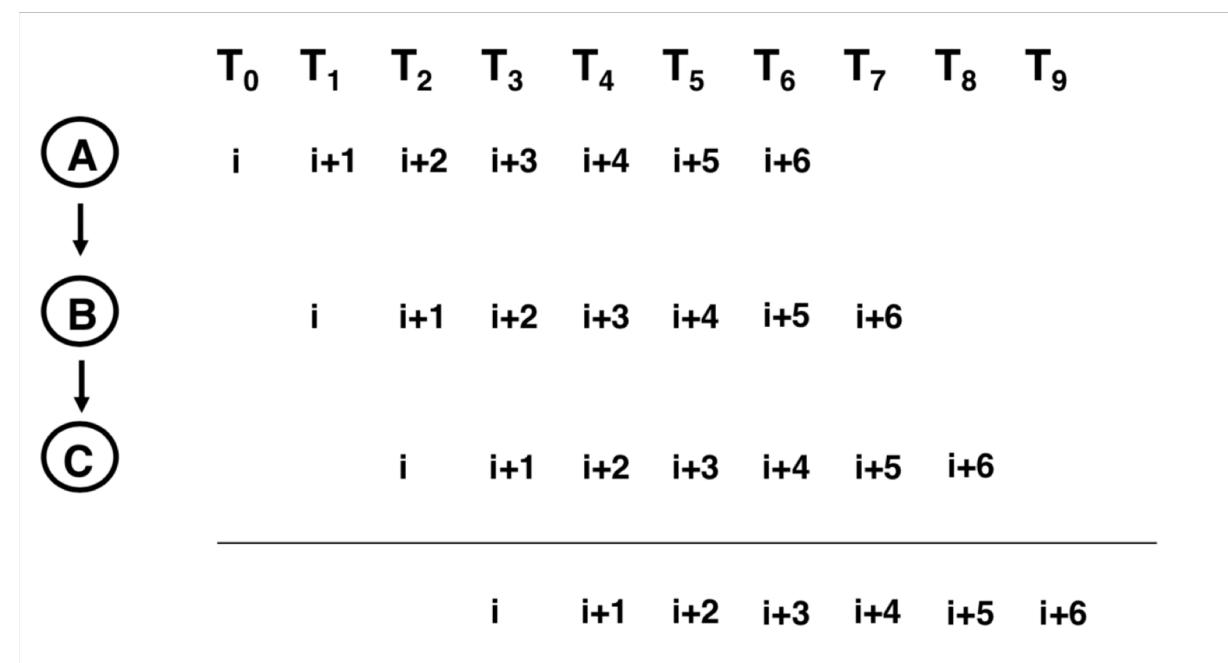


Pipeline parallelism

- **Definition:** each Stage works on a part of a solution. The output of one stage is the input of the next.

(Note: This works best when each stage takes the same amount of time to complete)

- **Examples:** Assembly lines, Computing partial sums

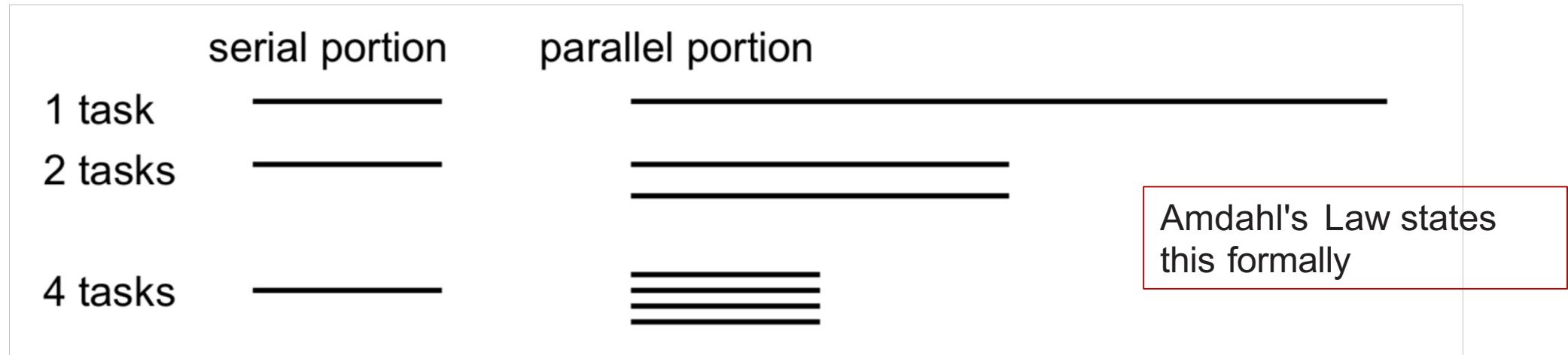


Is it really worth it to go Parallel?

- Writing effective parallel applications is difficult!!
 - Load balance is important
 - Communication can limit parallel efficiency
 - Serial time can dominate
- Is it worth your time to rewrite your application?
 - Do the CPU requirements justify parallelization? Is your problem really 'large'?
 - Is there a library that does what you need (parallel FFT, linear system solving)
 - Will the code be used more than once?

Theoretical Upper Limits to Performance

- All parallel programs contain:
 - parallel sections (we hope!)
 - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness



Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
- Effect of multiple processors on run time

$$t_n = (f_p / N + f_s) t_1$$

where

f_s = serial fraction of code

f_p = parallel fraction of code

N = number of processors

t_1 = time to run on one processor

Limit Cases of Amdahl's Law

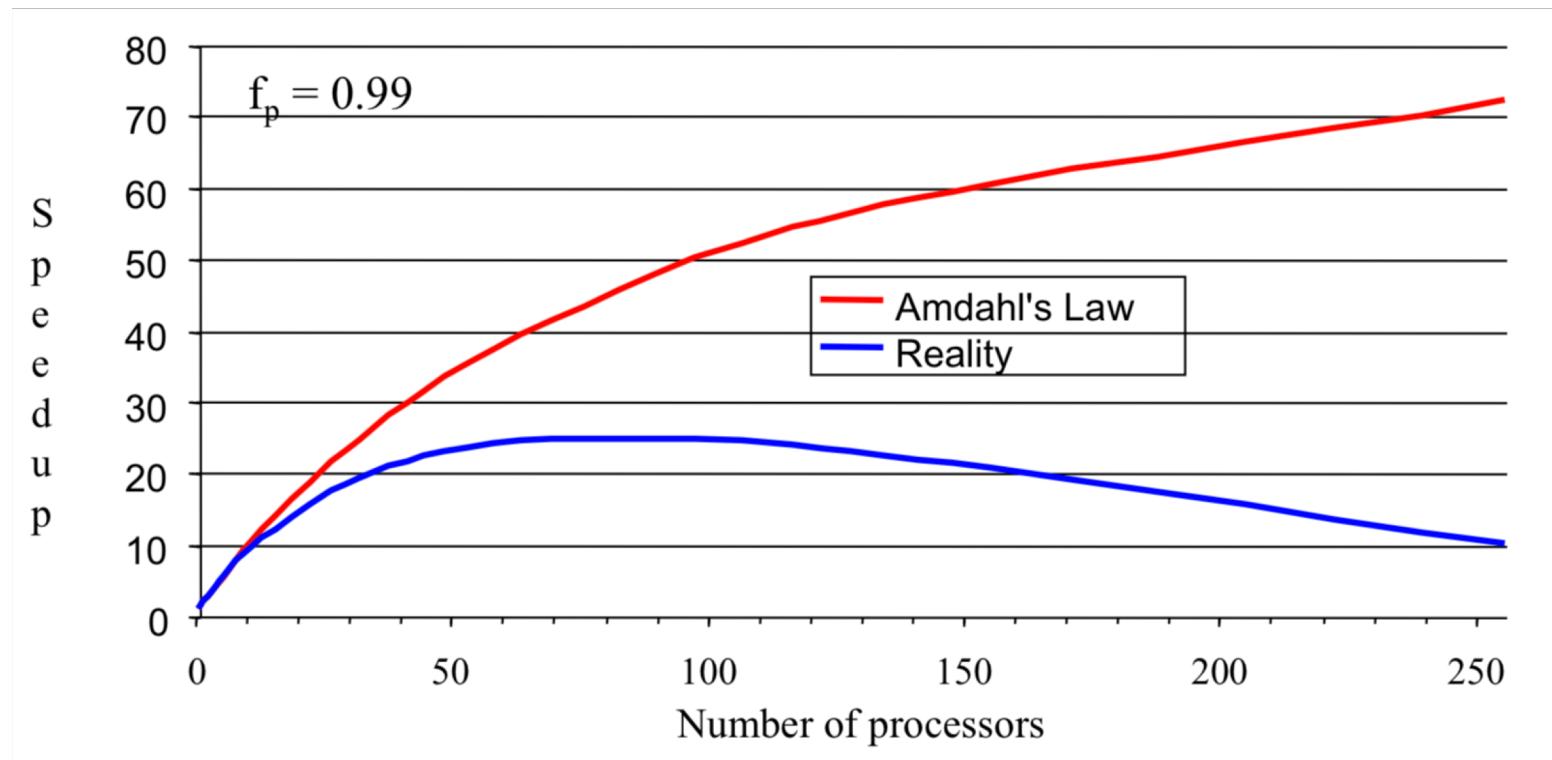
- Speed up formula:
 - $S = 1 / (f_s + f_p/N)$
- where
 - f_s = serial fraction of code
 - f_p = parallel fraction of code
 - N = number of processors
- Case:
 1. $f_s = 0, f_p = 1$, then $S = N$
 2. $N \rightarrow \text{infinity}$: $S = 1/f_s$; if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors

More terminology

- **synchronization:** the temporal coordination of parallel tasks. It involves waiting until two or more tasks reach a specified point (a sync point) before continuing any of the tasks.
- **parallel overhead:** the amount of time required to coordinate parallel tasks, as opposed to doing useful work, including time to start and terminate tasks, communication, move data.
- **granularity:** a measure of the ratio of the amount of computation done in a parallel task to the amount of communication.
 - fine-grained (very little computation per communication-byte)
 - coarse-grained (extensive computation per communication-byte).

Practical Limits: Amdahl's Law vs. Reality

- 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

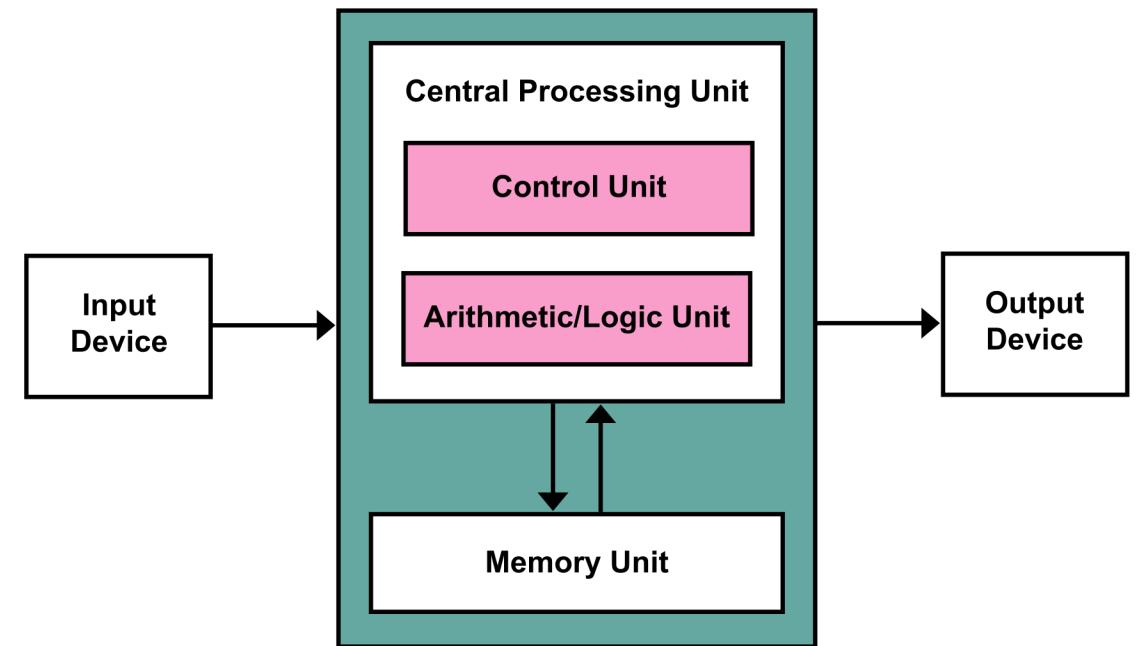


Von Neumann Architecture

- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

Basic Design

- Basic design
 - Memory is used to store both program and data instructions
 - Program instructions are coded data which tell the computer to do something
 - Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.

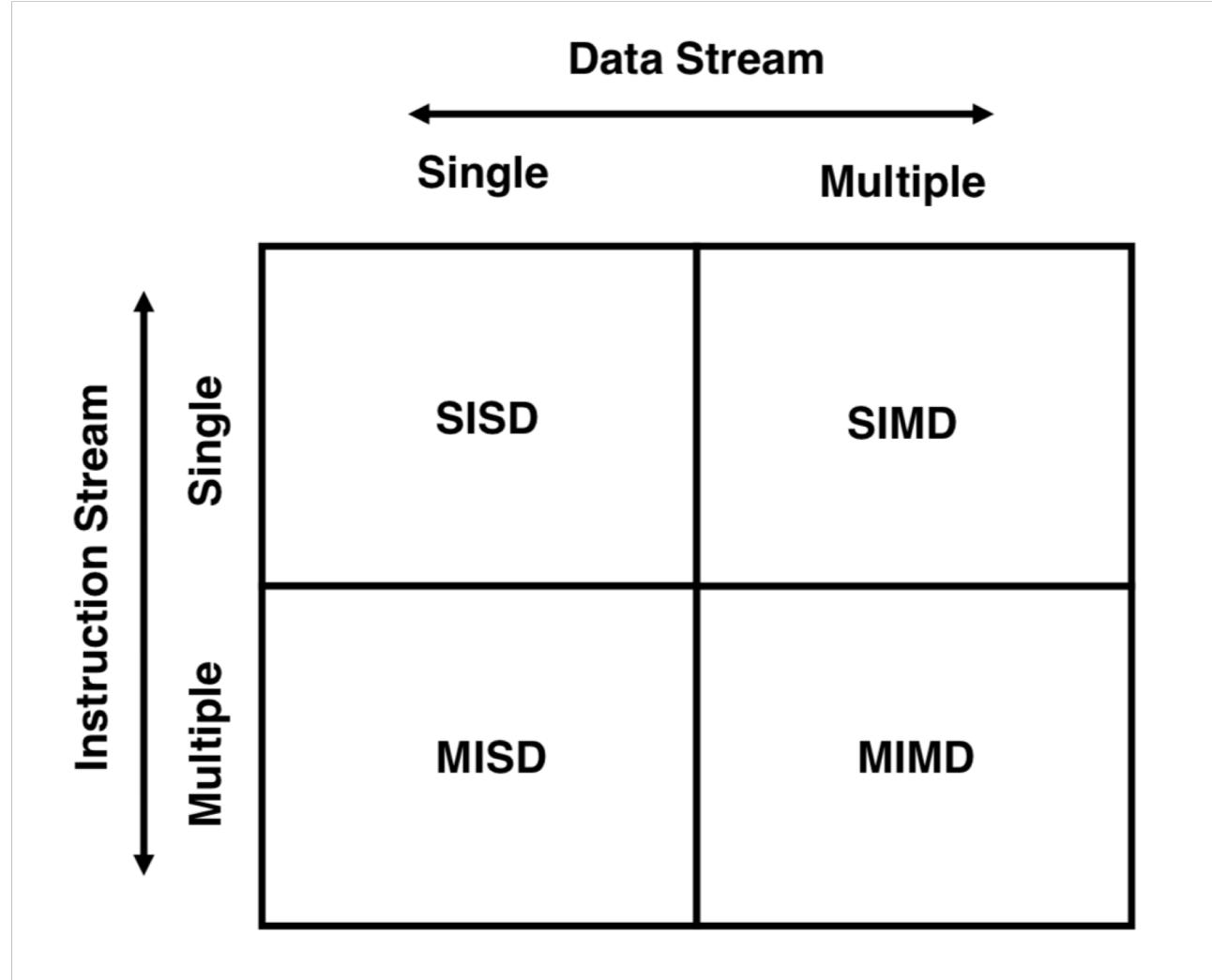


Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

Flynn's Taxonomy

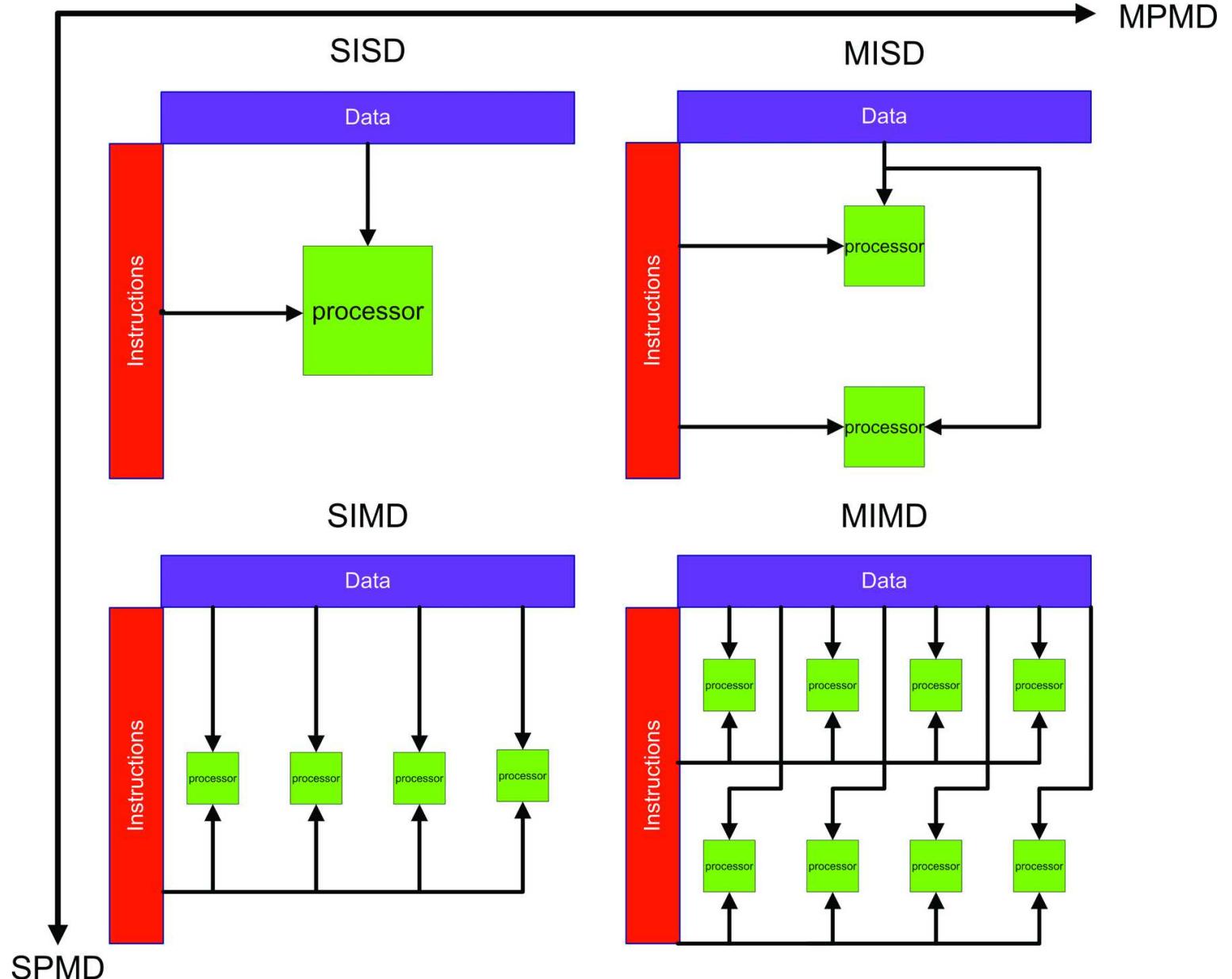
- Classification Scheme for Parallel Computers



Overview of Flynn's Taxonomy

- Single Instruction, Single Data (**SISD**)
 - No parallelism in the execution
 - Old single processor architectures
- Single Instruction, Multiple Data (**SIMD**)
 - Multiple data streams processed with one instruction stream at the same time
 - Typical in graphics hardware and GPU accelerators
 - Special SIMD machines in high-performance computing
- Multiple Instructions, Single Data (**MISD**)
 - Multiple instructions applied to the same data in parallel
 - Rarely used in practice, only for fault tolerance
- Multiple Instructions, Multiple Data (**MIMD**)
 - Every modern processor, compute clusters

Flynn's Taxonomy



Schmeisser et al. 2009

Dell EMC PowerEdge Servers

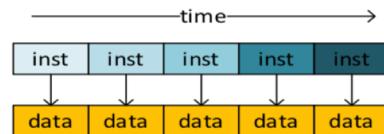
- Next generation accelerators
 - Graphics Processing Unit (GPU) Market to Surpass **67.6 Million units by 2020***
 - Field Programmable Gate Arrays (FPGA) Market worth **\$7.2 Billion by 2022***

CPU vs GPU vs FPGA

CPU - SISD

- Applicable to any programmable environment and language.
- Non-vectorized integer applications

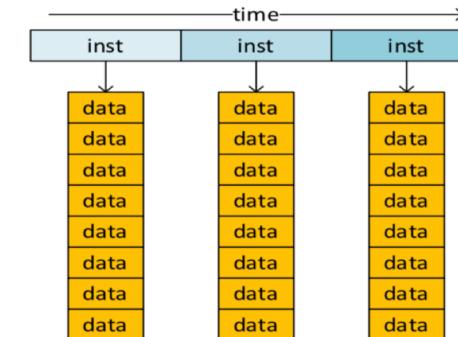
- Remains the core center of computation



GPU - SIMD

- Excels at vectored floating point
- Requires large data to be effective
- Hurt by branches or exceptions – "if" statements

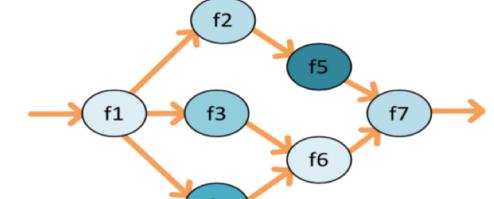
- Remains focused on a subset of high-performance problems
- Not going away; limited breadth



FPGA - MIMD

- Many independent instructions can operate in parallel and with either small amounts or large amounts of data
- Excellent streaming with IO devices

- Greater breadth than GPUs
- Expect higher prevalence for enterprise applications than GPUs over time



DELL EMC /World

Types of Parallel Computers (Memory Model)

- Nearly all parallel machines these days are multiple instruction, multiple data (MIMD)
- A much more useful way to classify modern parallel computers is by their memory model
 - shared memory
 - distributed memory

