Concurrent and Distributed Programming (CSC1101)

Basics of Concurrent, Parallel & Distributed Processing

2024/2025

Graham Healy

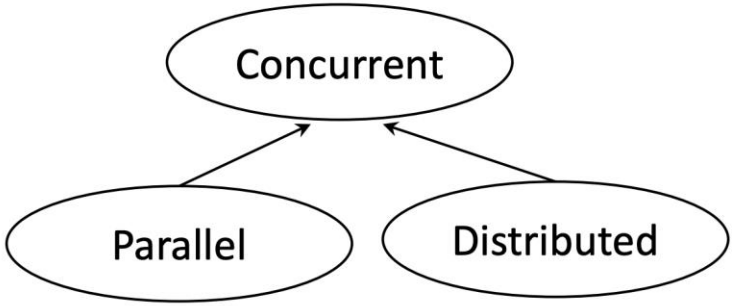
Concurrent vs Parallel vs Distributed

Concurrent programs: different parts of the program/algorithm/problem can be executed out-of-order (or in partial order) without affecting the final outcome.

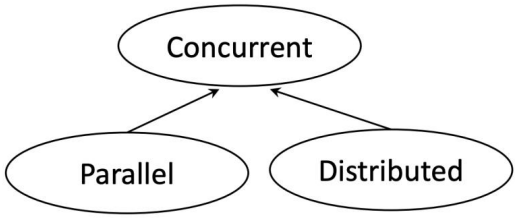
Parallel programming: the simultaneous use of multiple resources (e.g., processors) to solve a problem.

Distributed system: a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.

All parallel and distributed programs must be concurrent.



Concurrent vs Parallel vs Distributed



- Both leverage available resources & boost performance
  - Hence much overlap between the two
- All distributed systems must make use of some form of concurrent programming
- At its simplest ...
  - Concurrent programming deals with ensuring correctness despite computation order
    - E.g., computational processes synchronising/communicating with each other
  - Distributed computing additionally must deal with failure
    - E.g., network down, node not responsive/down, etc.

Concurrent

Concurrent Programming is about facilitating the performance of multiple computing tasks at the same time. This could be through:

Multi-tasking: allows several programs or processes to access the CPU

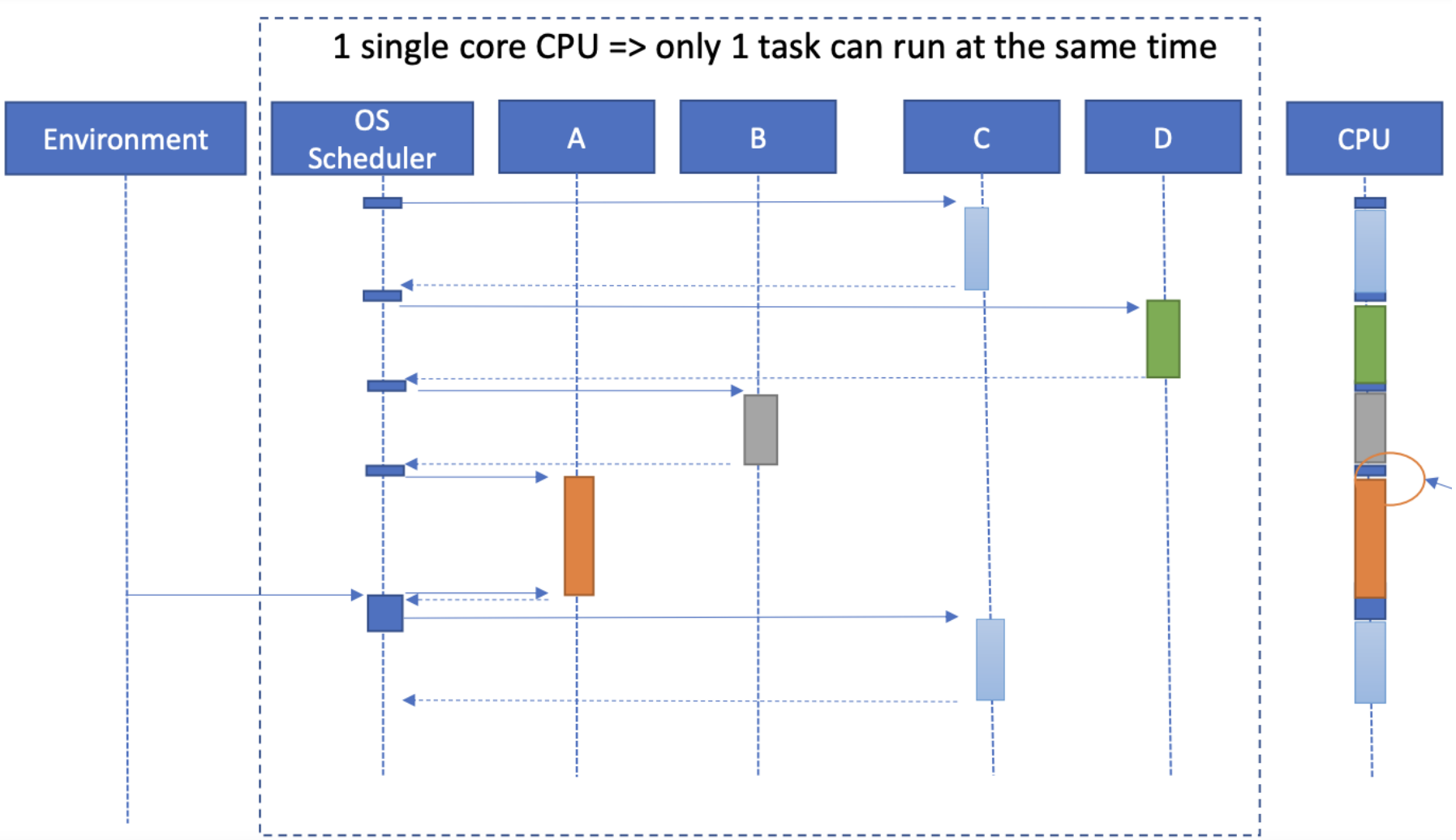
- E.g., using mouse, watching Netflix, updating a spreadsheet and scanning your PC for viruses at the same time
- This permits: intensive I/O processing and with effective signal handling, resources being shared among multiple tasks.

Multi-threading: implementation of multiple threads of computation in a single process/program

- E.g., print a draft of the document while continuing to in the document.
- Without multi-threading, User Interfaces (UIs) would be very slow (system only handles one action at a time)

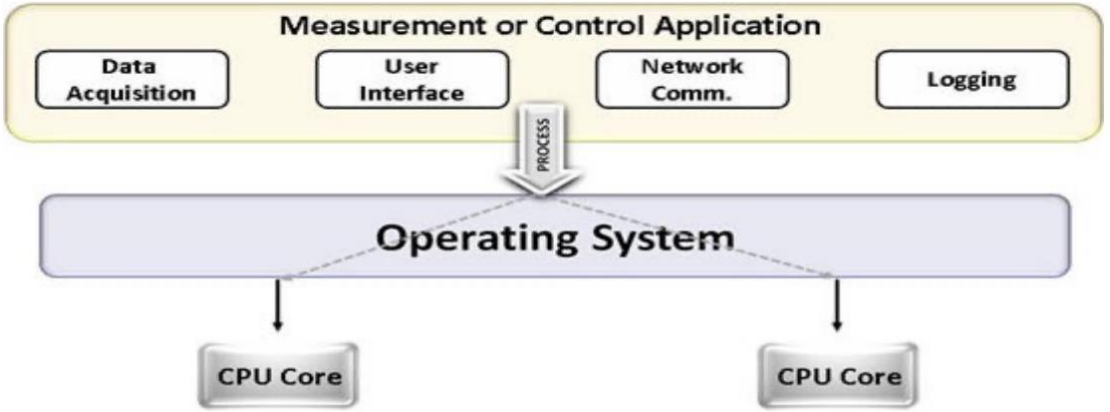
Tasks A, B, C, D, ...

Multi-tasking



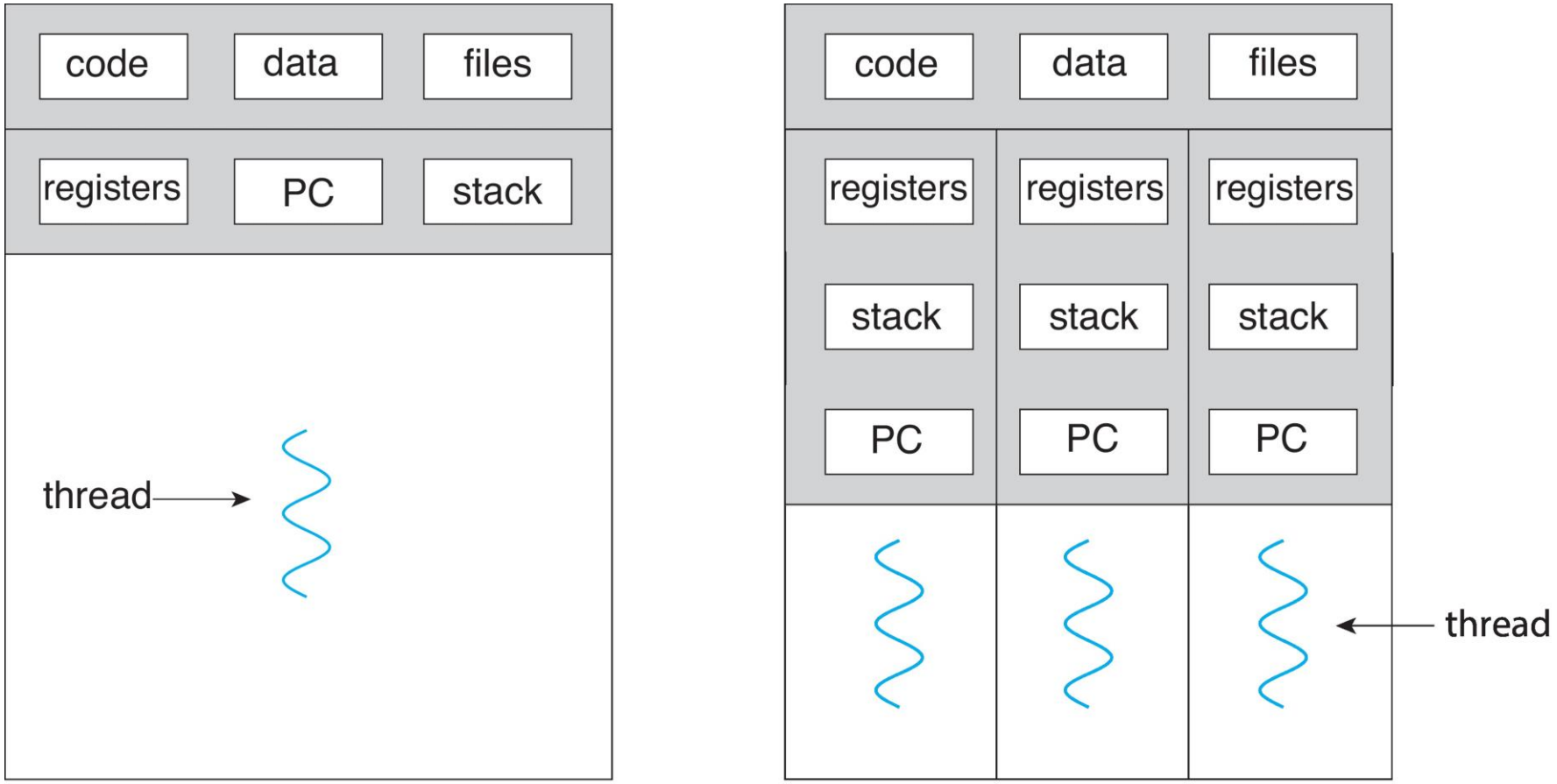
Multi-threading Definition

- **Multi-threading**: extends multitasking to application (process)-level,
  - subdivides operations in one application (process) into individual threads.
  - (Previously we only saw multi-tasking at process level)
- Each thread within an application (process) can potentially run in parallel.
  - They are lightweight processes
- OS scheduler splits processing time among different applications **and** among each thread within an application.



Dual-core system enables multithreading

Concurrency, Threads and Processes

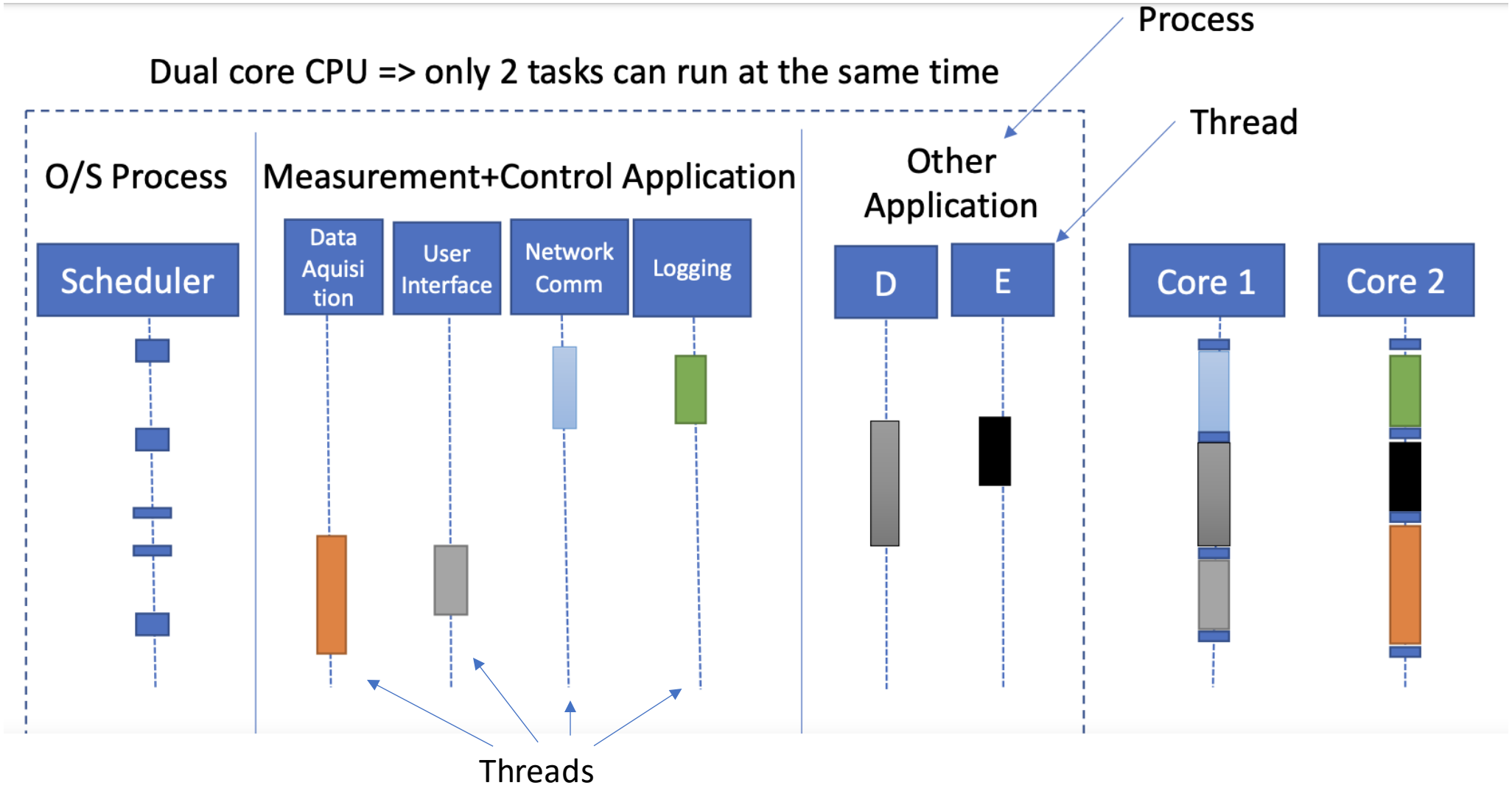


single-threaded process

multithreaded process

From: Silberschatz, Abraham, and Peter Baer Galvin. "Operatings System Concepts." (2018).

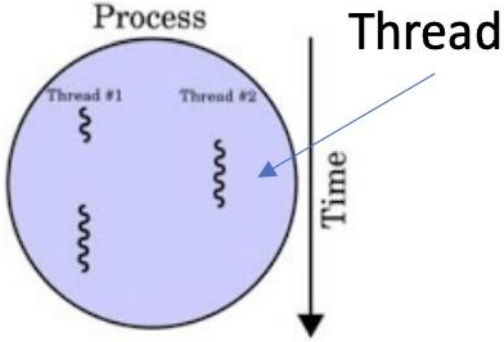
Multi-threading



Concurrency, Threads and Processes

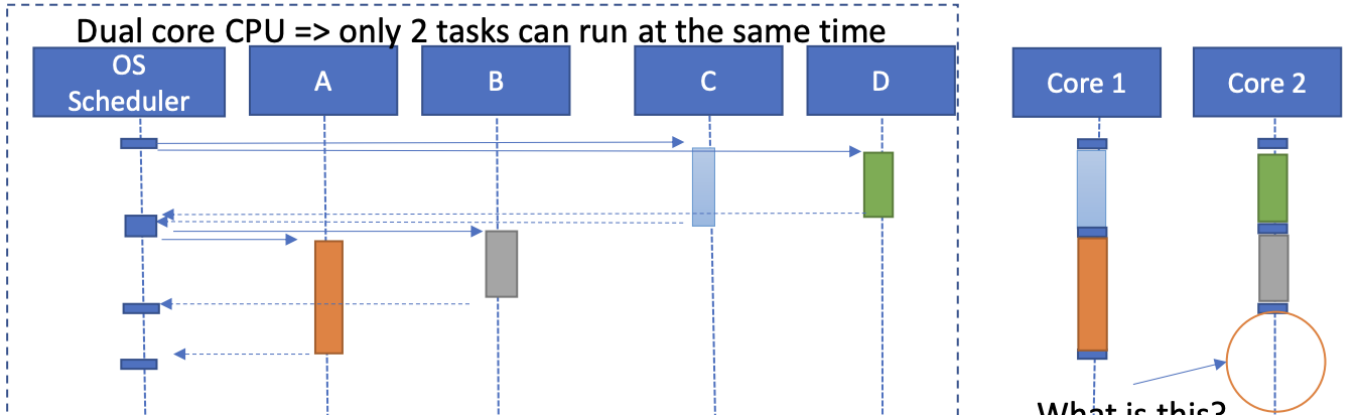
- Concurrency is the ability to run several parts of a program or several programs at the same time.
- A modern multi-core computer has several CPU's or several cores within one CPU.

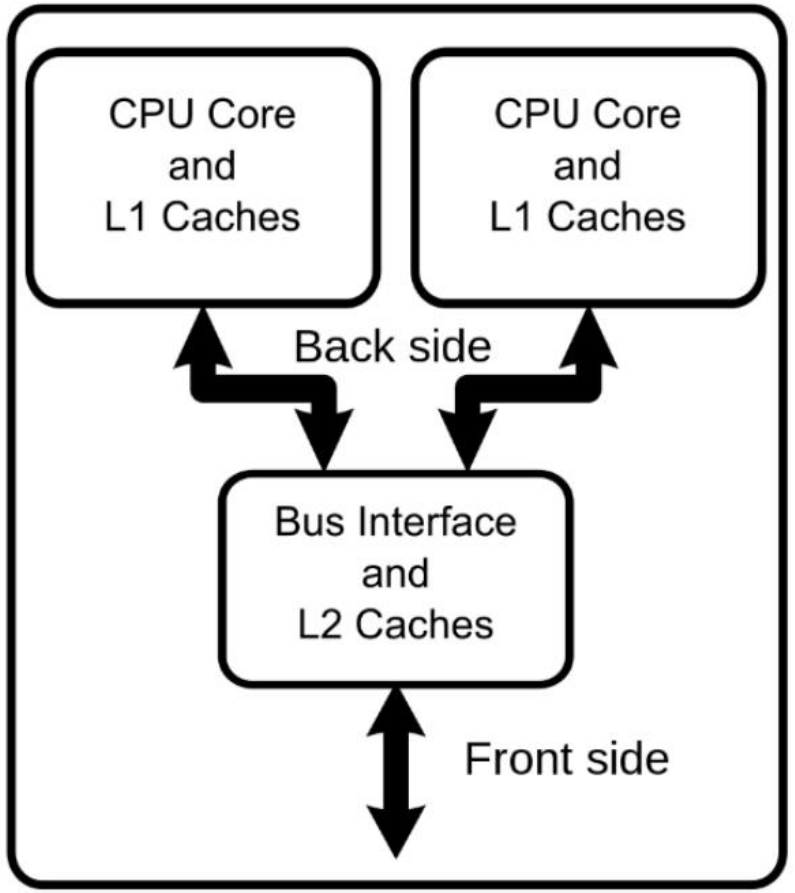
- We distinguish between processes and threads:
  - **Process**: runs independently and isolated of other processes (multitasking)
    - Cannot directly access shared data in other processes.
    - Process resources allocated to it via OS, e.g. memory, CPU time.
  - **Threads**: (or lightweight processes)
    - Own call stack but can access shared data.
    - Every thread has its own memory cache.
    - Thread reads shared data, stores it in its own memory cache.



Multi-core

- **Multi-Core**: multiple cores on one chip, multitasking OSs can truly run many tasks in parallel. O/S must support this (most do).
- Multiple compute engines work independently on different tasks.
- **OS Scheduling** dictates which task runs on the CPU Cores.





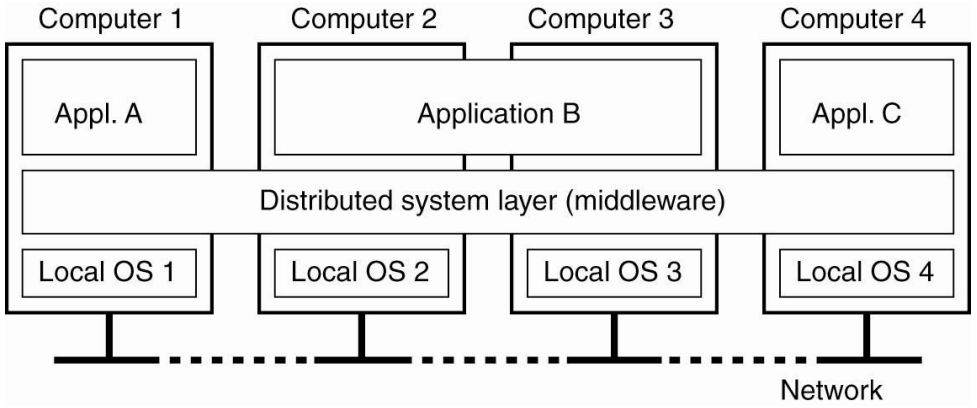
https://commons.wikimedia.org/w/index.php?curid=11894458

Distributed

Distributed System is a collection of independent computers that appears as a single coherent system to its users.

Wherever/whenever users interact with a distributed system, interaction should be uniform (all the same) & consistent (up to date)

- There are many types of Distributed Systems based on:
  - Device types: grid, desktops, sensors/IoT, ...
  - Centralized/Decentralized: Client/Server, Peer-to-Peer, Mesh networks, CDNs, ..
- Any many considerations:
  - Device reliability
  - Locations (or even relocation) at runtime
  - Data representations
  - Possible replication of objects at different locations
  - ....(later)



What About Parallel?

Mainly focused on strategies to break problems up (and reassemble them) so we can process each part independently.

- With parallelisation:
  - We can use different hardware for each sub-problem
  - We can do lots of work simultaneously
  - However, we need to consider concurrency, as the order of execution and completion may vary compared to a serial program

Intro to Concurrent Processing / Programming

Intro to Concurrent Processing

(today and tomorrow)

- Definitions
- Properties of Concurrent Systems
- Concurrent Decomposition
- Architectures for Concurrency
- Concurrent Scalability and Performance

Concurrency (External View Definition)

- Concurrent Programs: when different parts of the program/algorithm/ problem can be executed:
  - out-of-order or in partial order
  - without affecting the final outcome

- But this definition focuses on the internal program structure, not on
  - what capabilities this grants the program

We can provide a better definition ...

Concurrent Program Capabilities

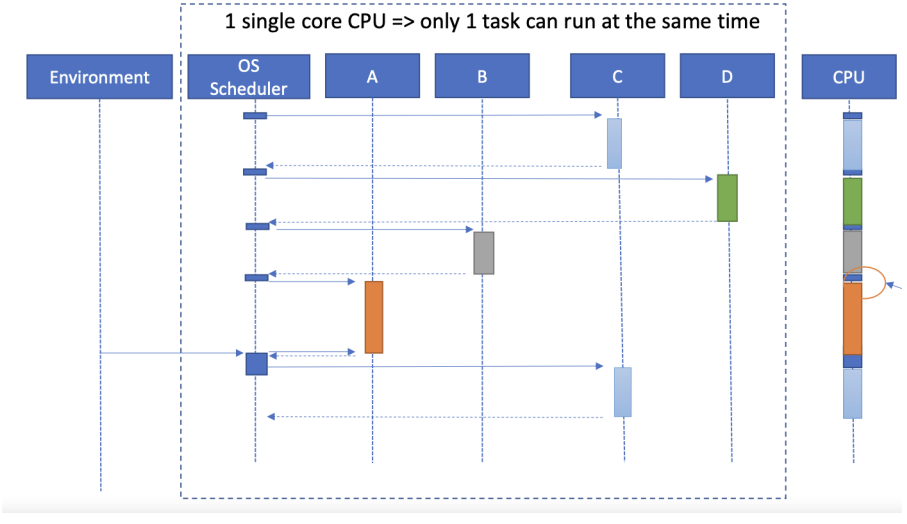
- Handle multiple inputs/outputs at the same time
- Perform multiple tasks at the same time
- Serve multiple clients at the same time
- Share resources between tasks
- Enable distribution of the program
- Improve performance
  - By enabling parallel execution (given suitable hardware)
  - By reducing wait time of multiple clients
  - By managing bottlenecks at shared resources

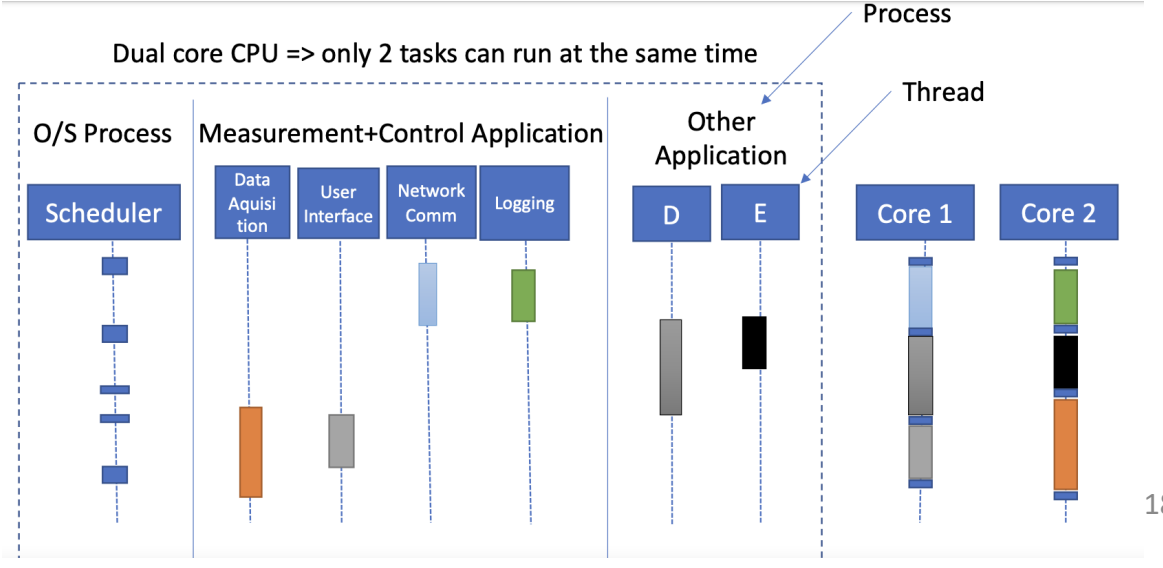
- But there is a cost to making program concurrent (need new rules and checks in code) – the cost is complexity

What do we mean by “at the same time”?

- Concurrency = just means “in the same time period”
- Parallel = actually executing at the same time

- i.e., applications can run concurrently on a single CPU (multi-tasking) but it doesn’t mean they are running in parallel (at the same time)





A New Definition….

Concurrency is a property of systems in which:

1. Several computations can be in **progress** at the same time, and
2. Potentially (need to) interact with each other.

The computations may then be executed on:

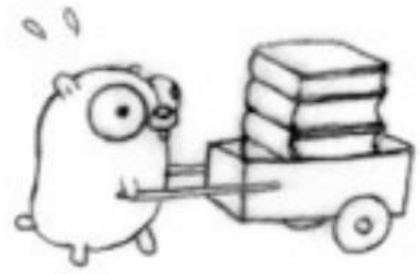
- multiple cores in the same chip (parallel),
- pre-emptively time-shared threads on the same processor (concurrent), or
- physically separated processors (parallel or distributed)...

Questions

- Can you have more threads than CPUs/Cores?
- Who decides how threads/processes are allocated to CPUs/Cores?
- Who decides how the application is broken into threads?
- Why are processes usually safe from concurrency issues?

A Simple Example Problem to Make Things More Concrete

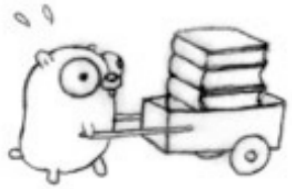
- Job to do: Move a pile of obsolete language manuals to the incinerator.



- With only one gopher this will take too long.

A Simple Example With Gophers (cont'd)

- Maybe more gophers…..

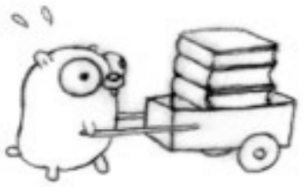


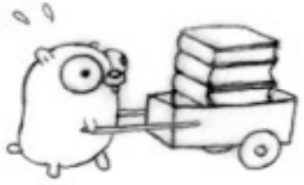
- More gophers are not enough; they need more carts.
  =>Adding CPUs/processes/threads with no way to allocate tasks does nothing

More Gophers

- More gophers and more carts

You've to wait until I'm finished...

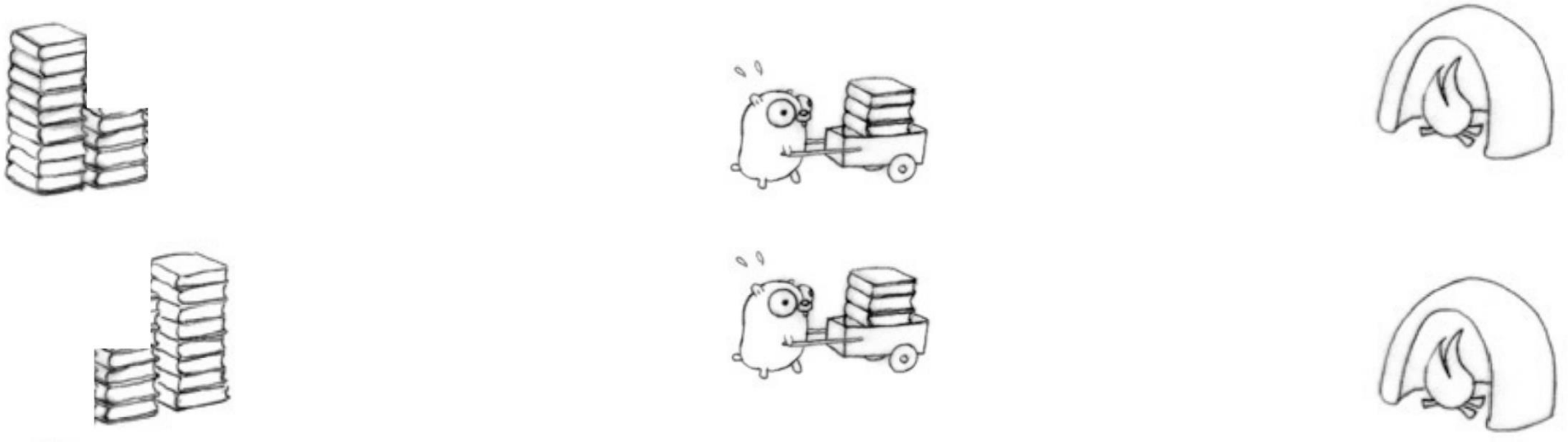




- Faster, but gives rise to bottlenecks at pile, incinerator.
- Now we need to synchronize the gophers.
- A message (i.e., communication between gophers) will do.

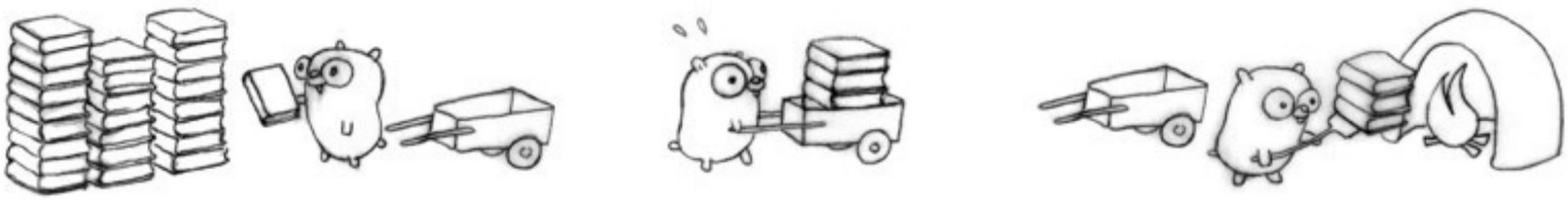
Doing the same amount of work with more Gophers (data decomposition)

- Split the piles of books



- Can finish twice as fast if no additional coordination required
- i.e., concurrent composition of two gopher procedures
- Full speedup only applies to embarrassingly parallel problems

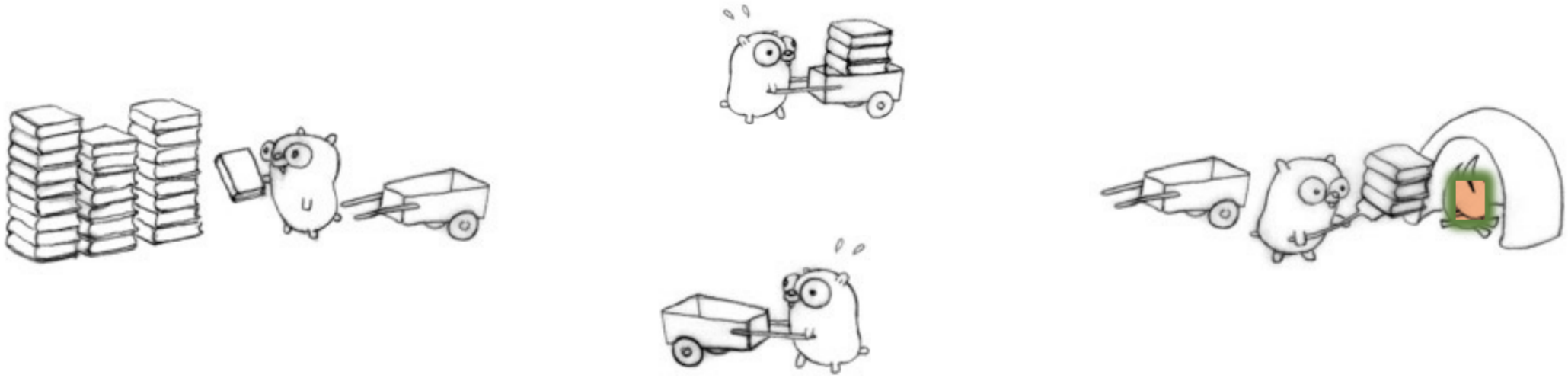
More Gophers: Another Design by splitting the tasks (functional decomposition)



- Three gophers in action, but with likely delays
- Each gopher is an independently executing procedure, plus coordination costs (communication)

Even More Gophers: Finer-grained concurrency

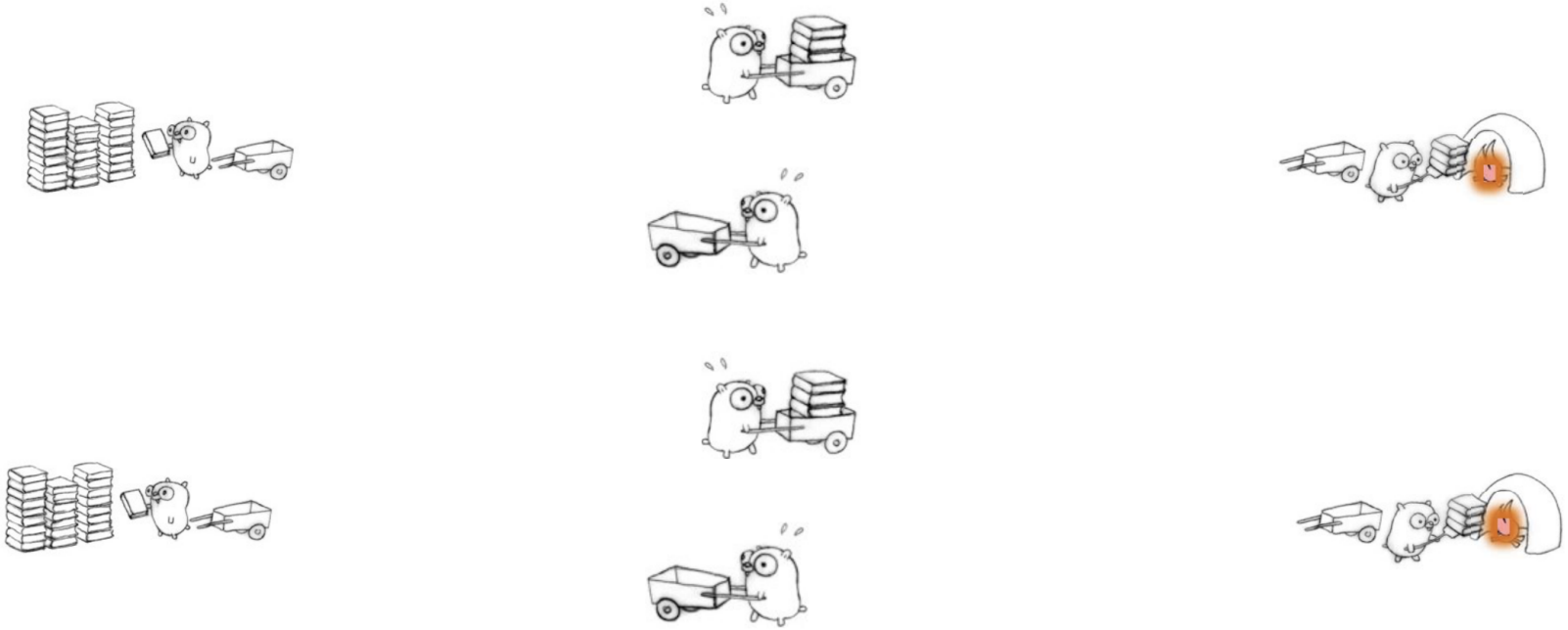
- Add another gopher procedure to return empty carts (split a task, add a thread).



- 4 gophers in action for better flow, each doing a simple task.
- If we arrange everything right (implausible but not impossible)= 4 times faster than original 1-gopher design.

Even More Gophers (cont'd): Another design

- Now parallelize on other axis; the concurrent design makes it easy: 8 gophers, all busy! (combination of functional and data decomposition)



- Or maybe no parallelization at all!
- Remember even if only 1 gopher is active at a time (zero parallelism), it's still a correct & concurrent solution.

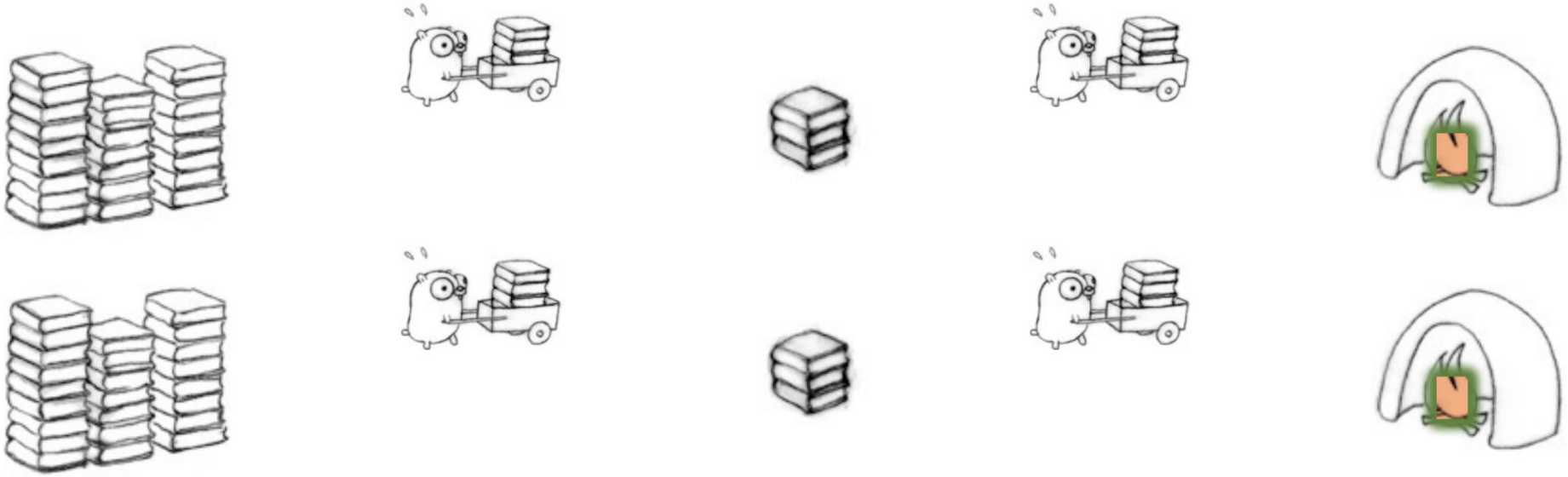
Even More Gophers (cont'd): Another design

- Here's another way to structure the problem (functional decomposition).

- Two gopher procedures, plus a staging pile (buffer/queue).
  - Queues provide a mechanism for communicating data between components of a system

Even More Gophers (cont'd): Another design

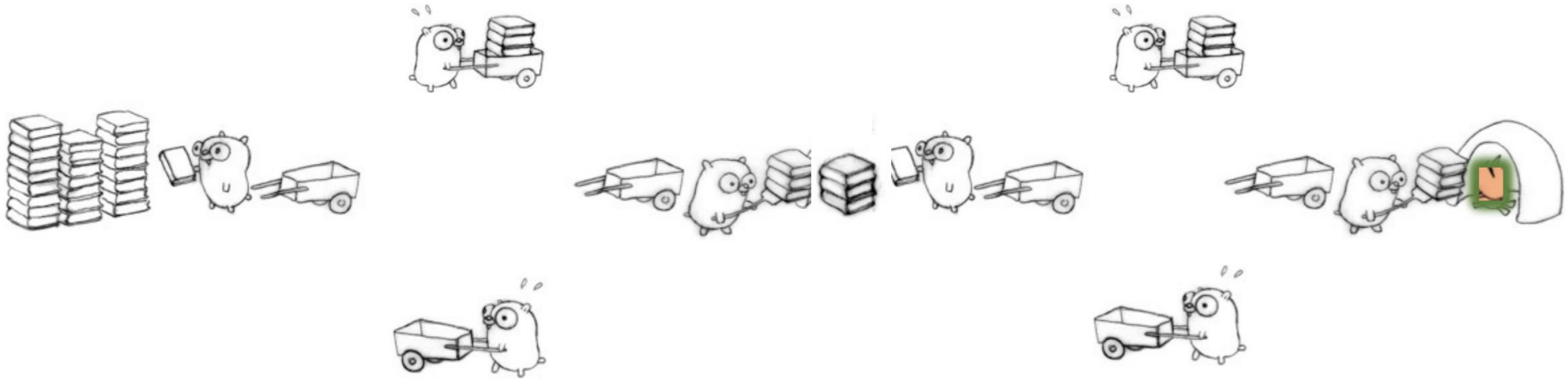
- Parallelize this in the usual way (combination of data and functional decomposition):



- i.e. can now run more concurrent procedures to get more throughput.

Even More Gophers (cont'd): A Different Way…

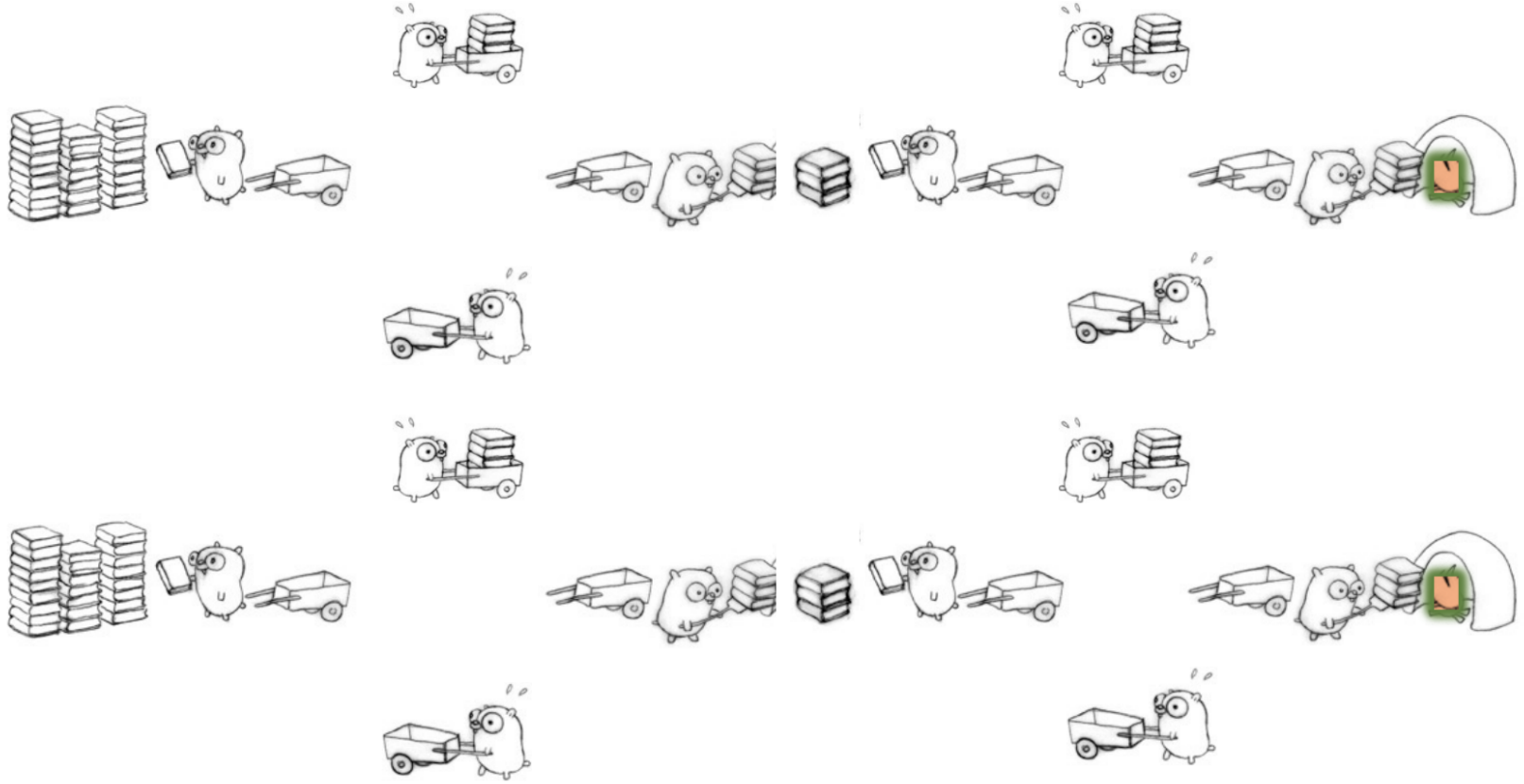
- Bring a staging pile to the multi-gopher concurrent model: (Functional decomposition only)



- i.e., everywhere threads/processes collaborate is a potential queue

Even More Gophers (cont'd): A Different Way…

- Add more decomposition to enable more distribution or parallelization:



The Lesson from All This…

- Many ways to break the processing down
  - Best strategy depends on task, data and platform
  - That's concurrent design
- Once the problem is broken down:
  - Opportunities for parallelization falls out of this
  - but we must assure/maintain correctness
- In computer science, for example, you create a concurrent design for a scalable web service with serving web content by substituting:
  - book pile => web content
  - gopher => CPU
  - cart => marshalling, rendering, networking, ...
  - incinerator => proxy, browser, or other consumer

A Clear(er) Analogy of Concurrency

- ***Concurrency*** is about dealing with lots of things at once.
- ***Parallelism*** is about doing lots of things at once.

These are not the same, but they are related.

- ***Concurrency*** is about structure, *parallelism* is about execution.
- ***Concurrency*** provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

- Example:
  - Concurrent: using MS Word, mediaplayer.
  - Parallel: calculating a Vector dot product, cells being updated in excel

Program Decomposition

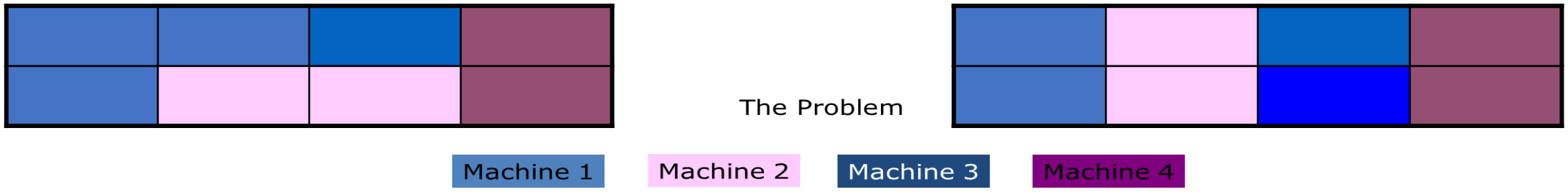
Three methods to decompose problem into smaller processes for parallel execution: *Functional Decomposition*, *Domain Decomposition*, or combination

Functional Decomposition

- Decompose *problem into different processes* to allocate to processors for simultaneous execution
- Good when no static structure or fixed number of calculations

Data Decomposition

- Partition *problem data* & distribute to processors for simultaneous execution
- Good where e.g.:
  - Static data (solve large matrix)



Simple example of concurrency problem with 2 threads....

Example

```
i = 0
```

Concurrent Thread A (trying to get to 10)

```
while (i < 10)
    i = i + 1;
    print “A won!”;
```

Concurrent Thread B (trying to get to -10)

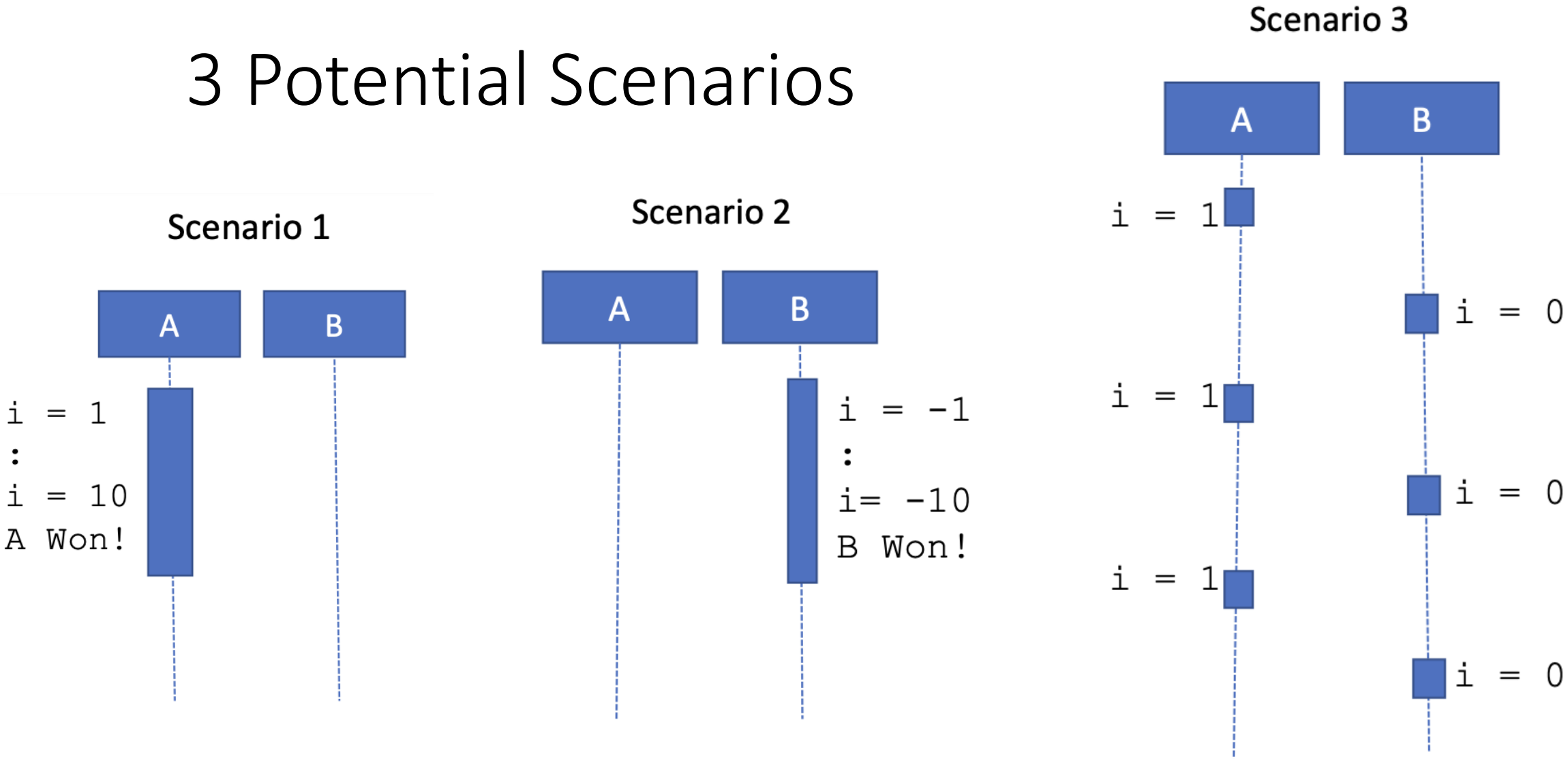
```
while (i > -10)
    i = i - 1;
    print “B won!”;
```

Some framework application

```
A.start(); B.start();
```

What are the possible outcomes if you run this code?

3 Potential Scenarios

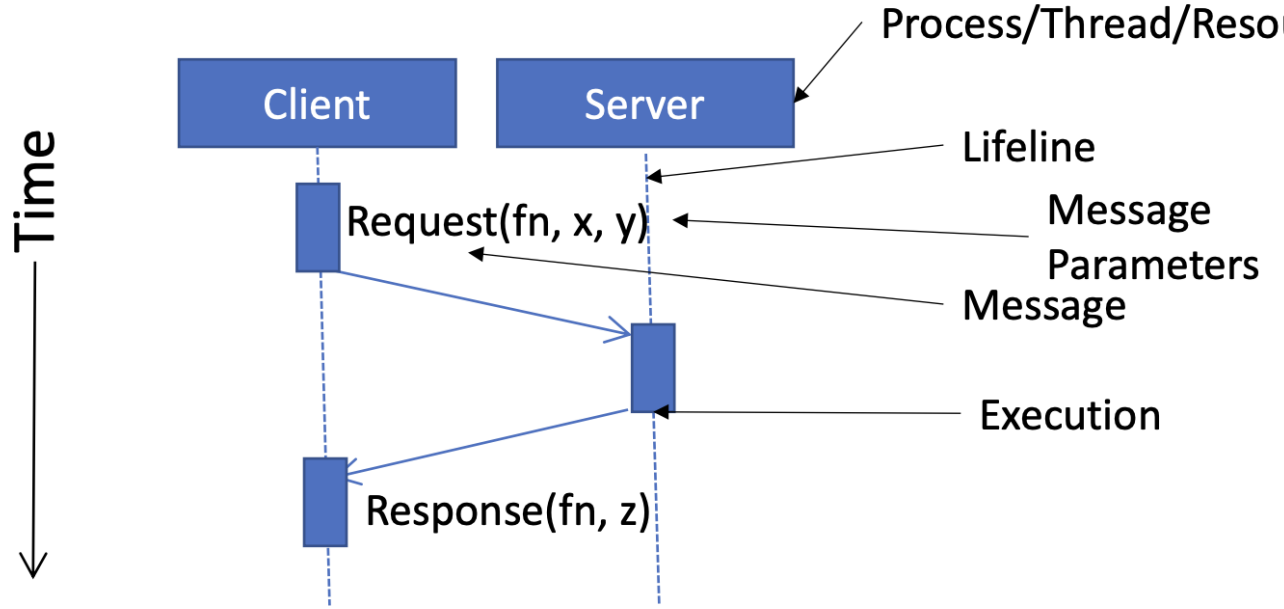


Murphy's Law (Expecting the Unexpected): The OS scheduler will try to do things in the worst possible way for your program.

UML sequence diagram notation: https://www.uml-diagrams.org/sequence-diagrams.html

Aside: UML Sequence Diagrams

- Show object/entity/process/thread interactions arranged in time sequence
- Very useful for describing and analyzing the behavior of concurrent systems



UML sequence diagram notation: https://www.uml-diagrams.org/sequence-diagrams.html

(Benign) Concurrency

- When do we not have to worry about it?
  - No shared data or communication
    - E.g., A program has 3 threads, each working on their own data
- Read-only data
  - E.g., A program has 3 threads reading the same shared data
    - No chance that 1 thread will modify the data and another will read the wrong data
    - E.g., global constants in your code, immutable objects, etc

(some) Strategies for Managing Concurrency

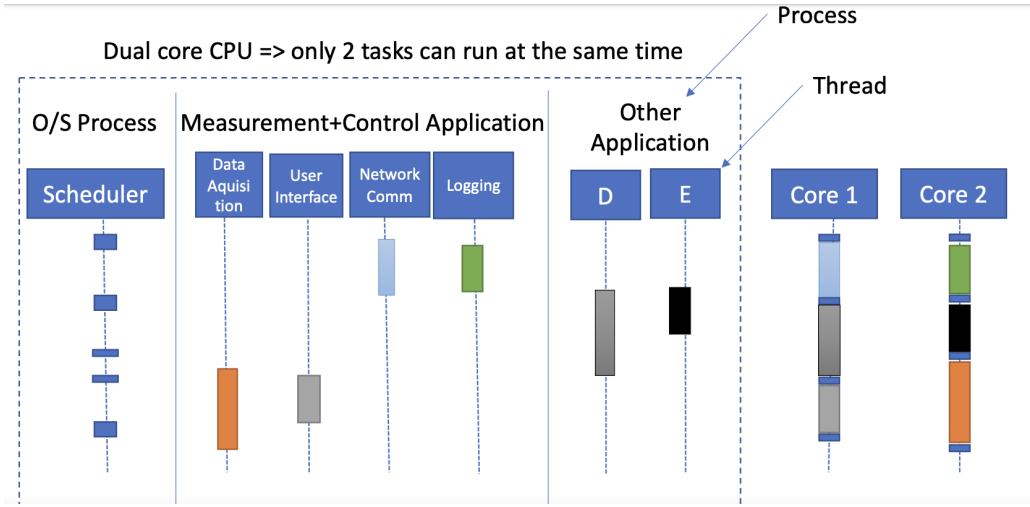
- Synchronisation mechanisms
  - Allow programs to write rules for concurrency e.g. locks, mutexes, semaphores, monitors, critical sections, etc
- Atomicity
  - Ensure no other thread changes data while it's running
- Conditional Synchronisation
  - Ensure code in different threads runs in the correct order e.g. condition variables, barriers
- Deadlock Prevention and Avoidance
  - Detecting / preventing deadlock

Potential Concurrency Problems

- **Deadlock**: 2+ threads stop and wait for each other (forever).
- **Livelock**: 2+ threads continue to execute but make no progress toward the goal (spinning).
- **Starvation**: some thread gets deferred forever (especially with ordering constraints…that we used to avoid deadlock).
- **Lack of fairness**: not all threads get a turn to make progress.
- **Race condition**: some possible interleaving of results in an undesired computation result. This is caused by an unprotected shared object/memory.
- **Heisenbugs**: Software bugs that seem to disappear when you study them (due to timing/state changes in the system)

... it's important to be clear on who owns data and at what time!

Race Conditions



- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Architectural Classification Systems

Von Neumann Architecture

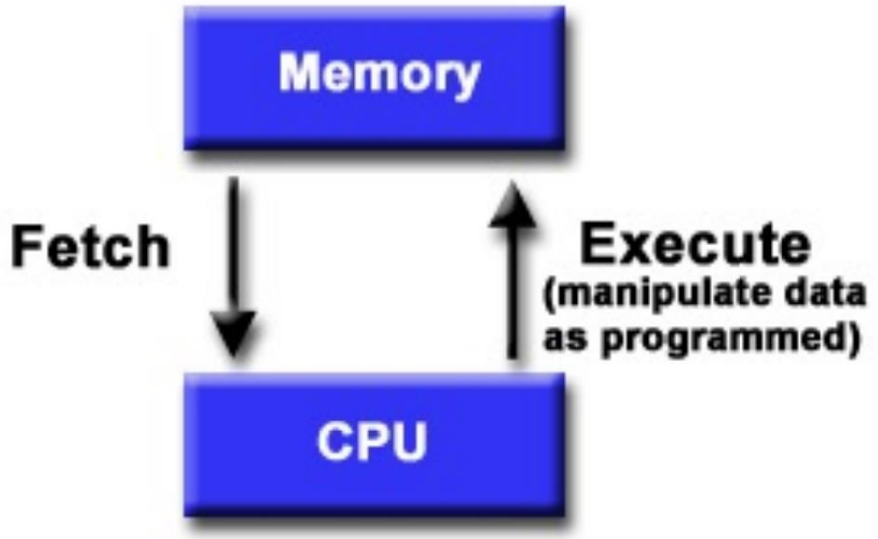
- For 50+ years, virtually all computers have followed a common machine model known as the von Neumann computer.
  - Named after the Hungarian mathematician John von Neumann.

- A von Neumann computer uses the **stored-program** concept.

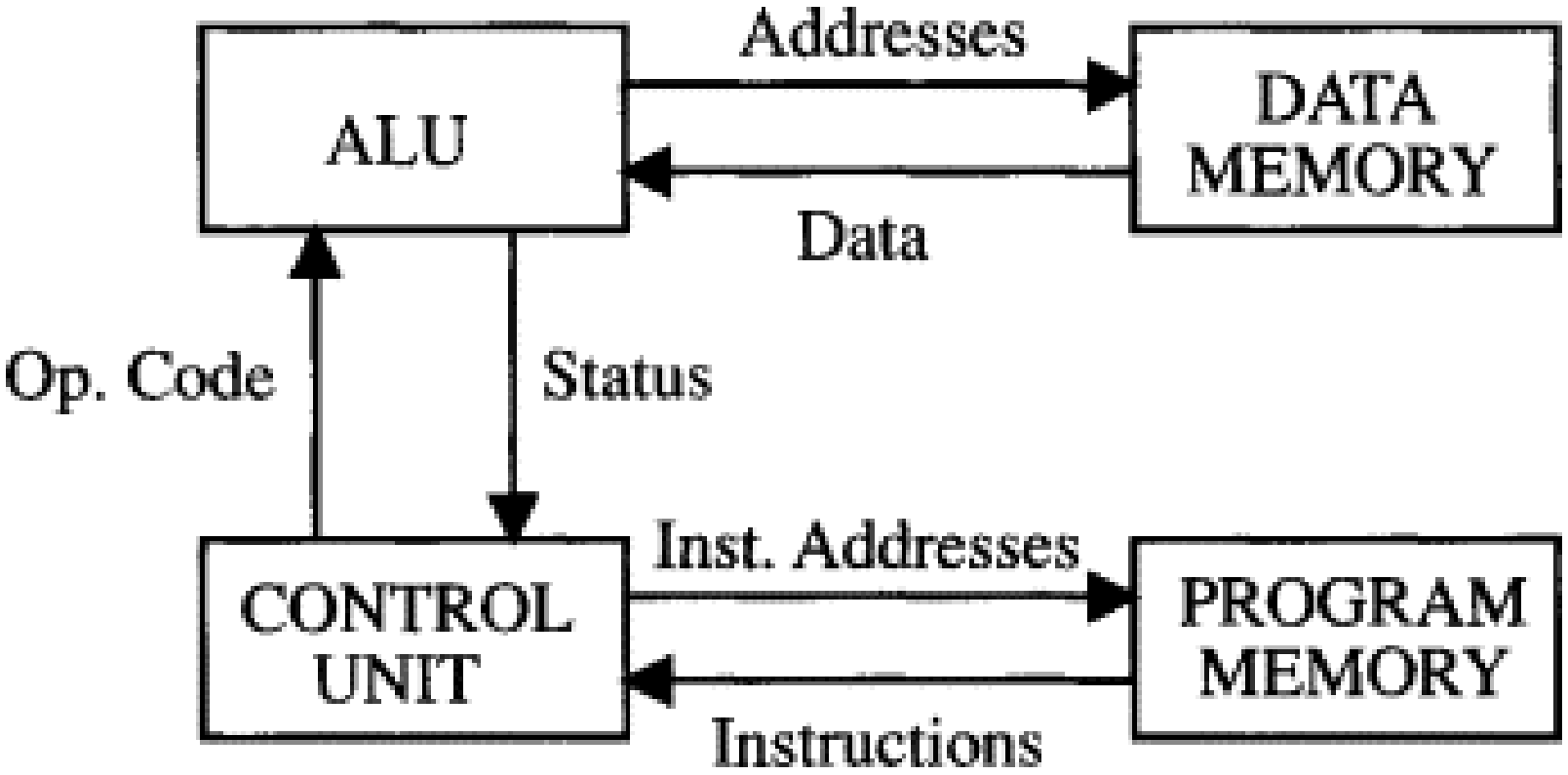
- The CPU executes a stored program that specifies a sequence of read and write operations on the memory

Basic Design

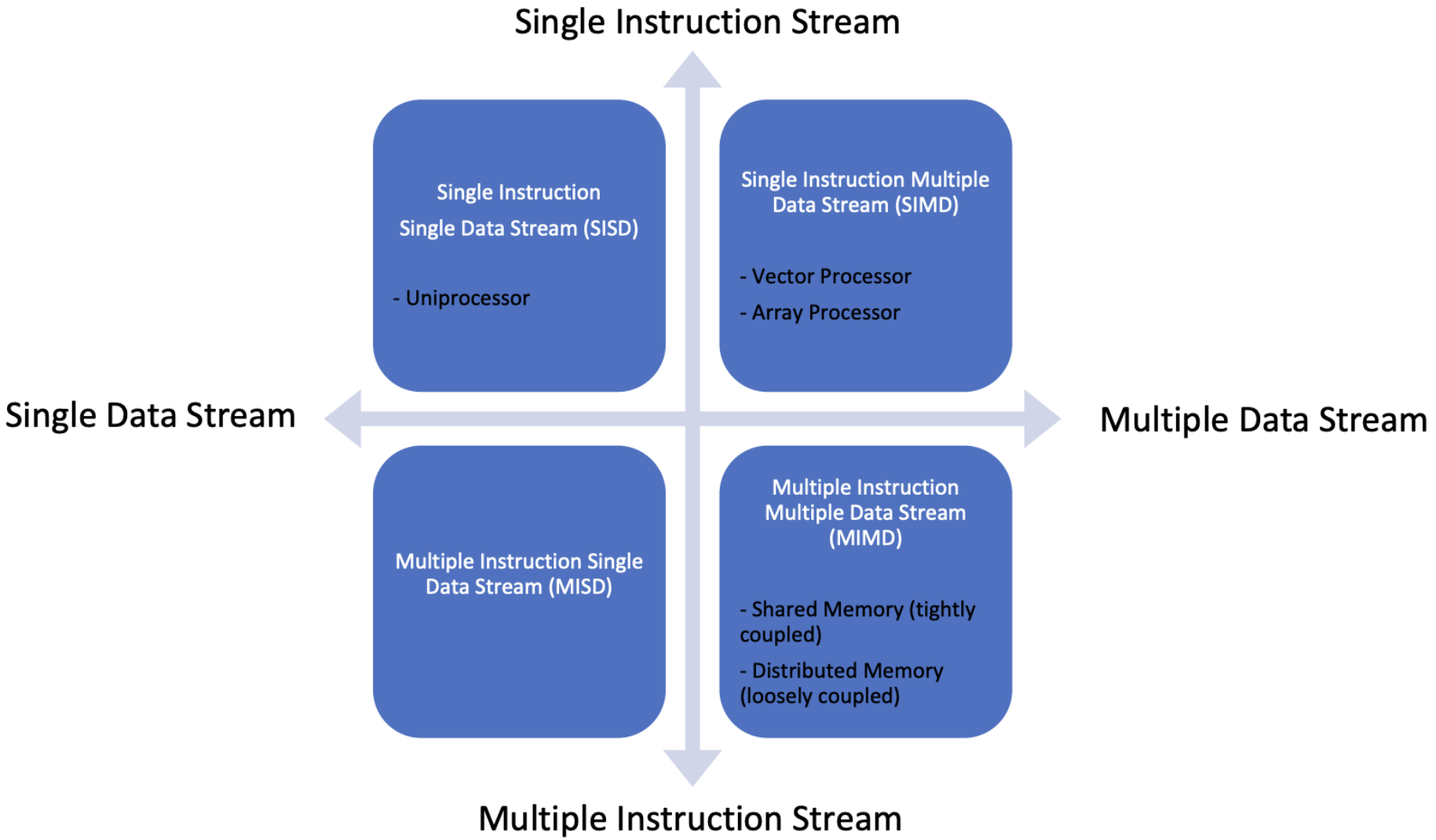
- Memory is used to store both program instructions and data
- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then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



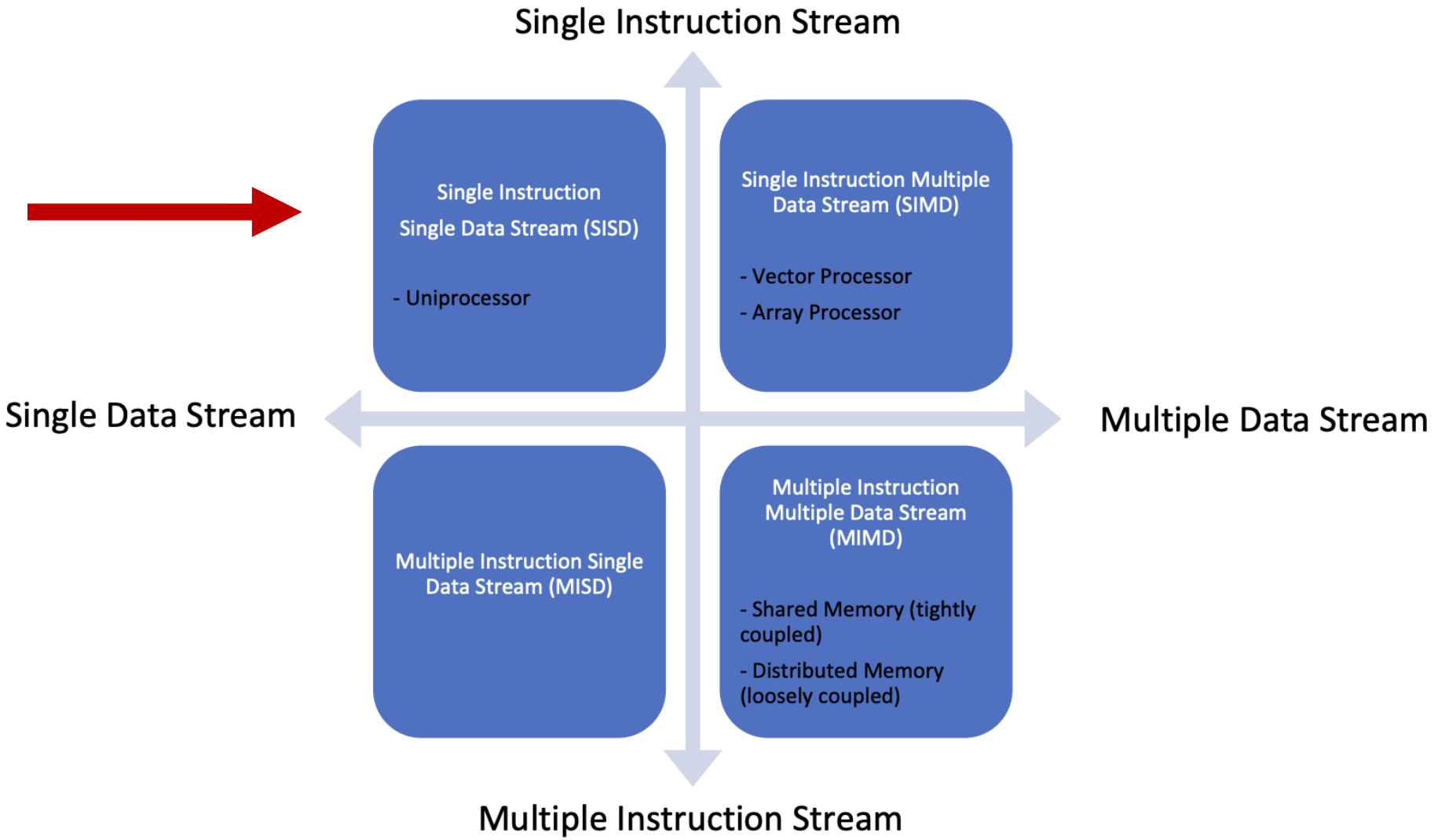
Basic Design



Computer Architecture Taxonomy - Flynn's Taxonomy

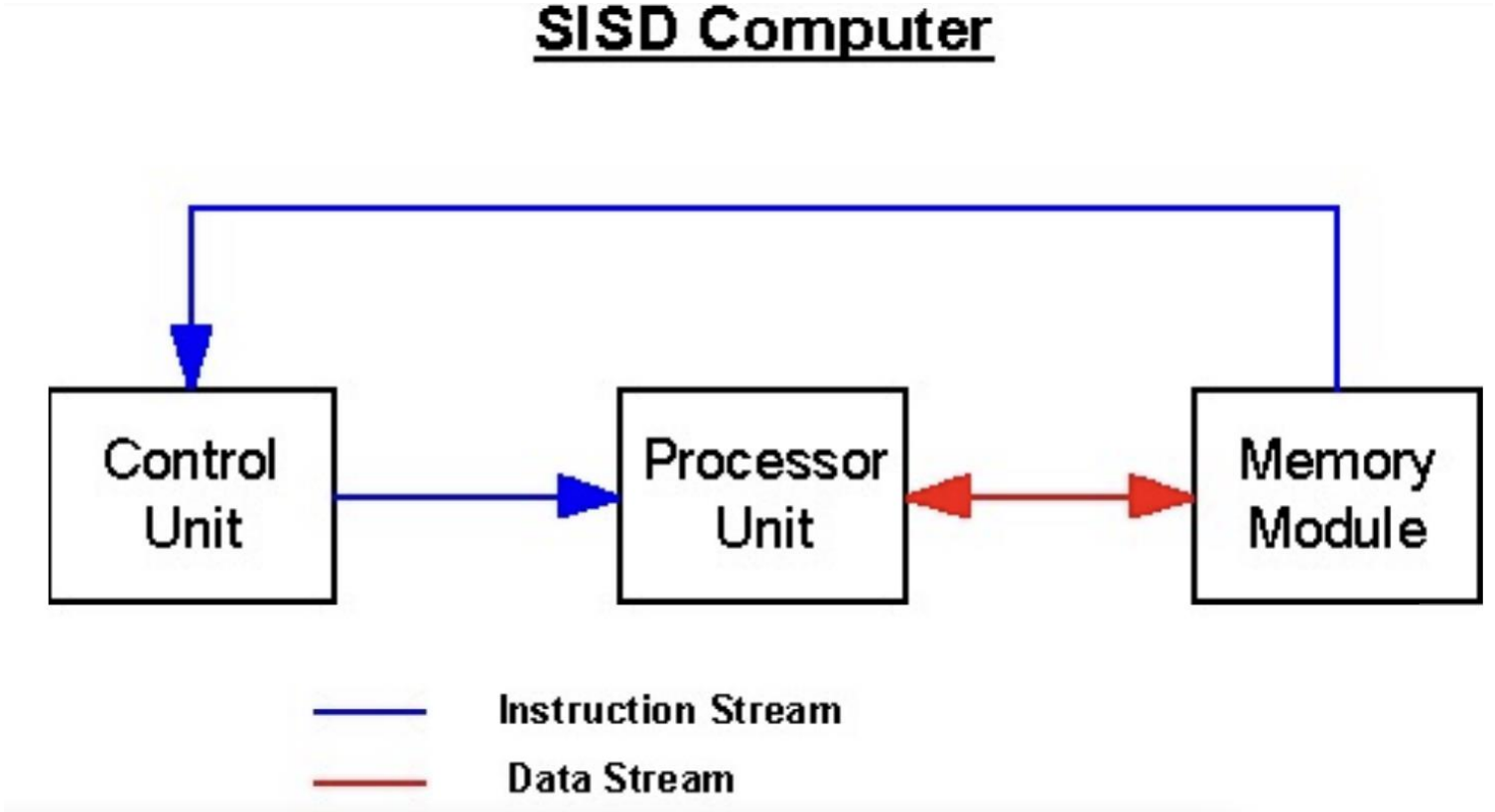


Computer Architecture Taxonomy - Flynn's Taxonomy



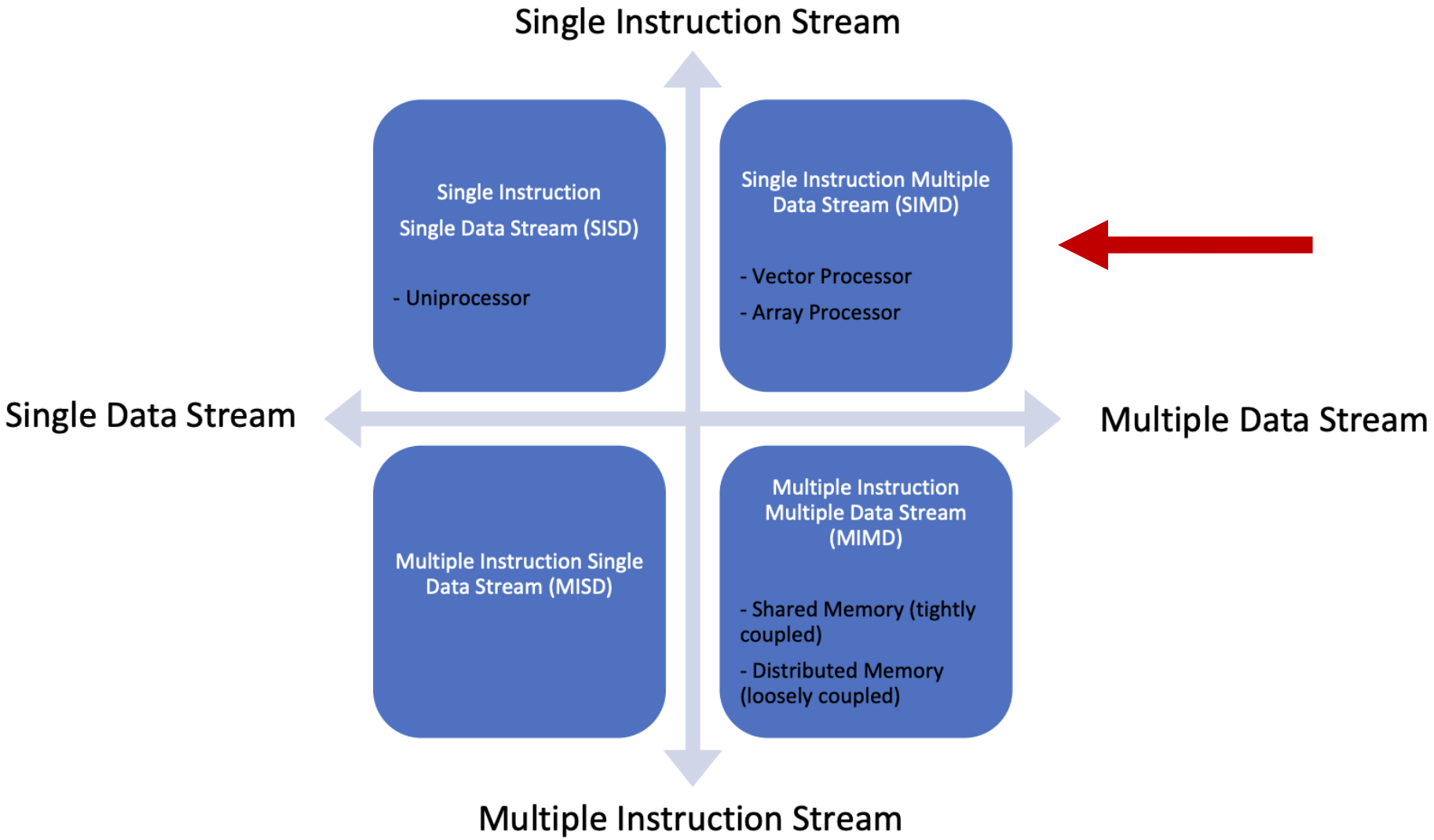
Single Instruction Single Data (SISD)

- Single processor
- Single instruction stream
- Data stored in single memory
- Uni-processor
- Old but still around



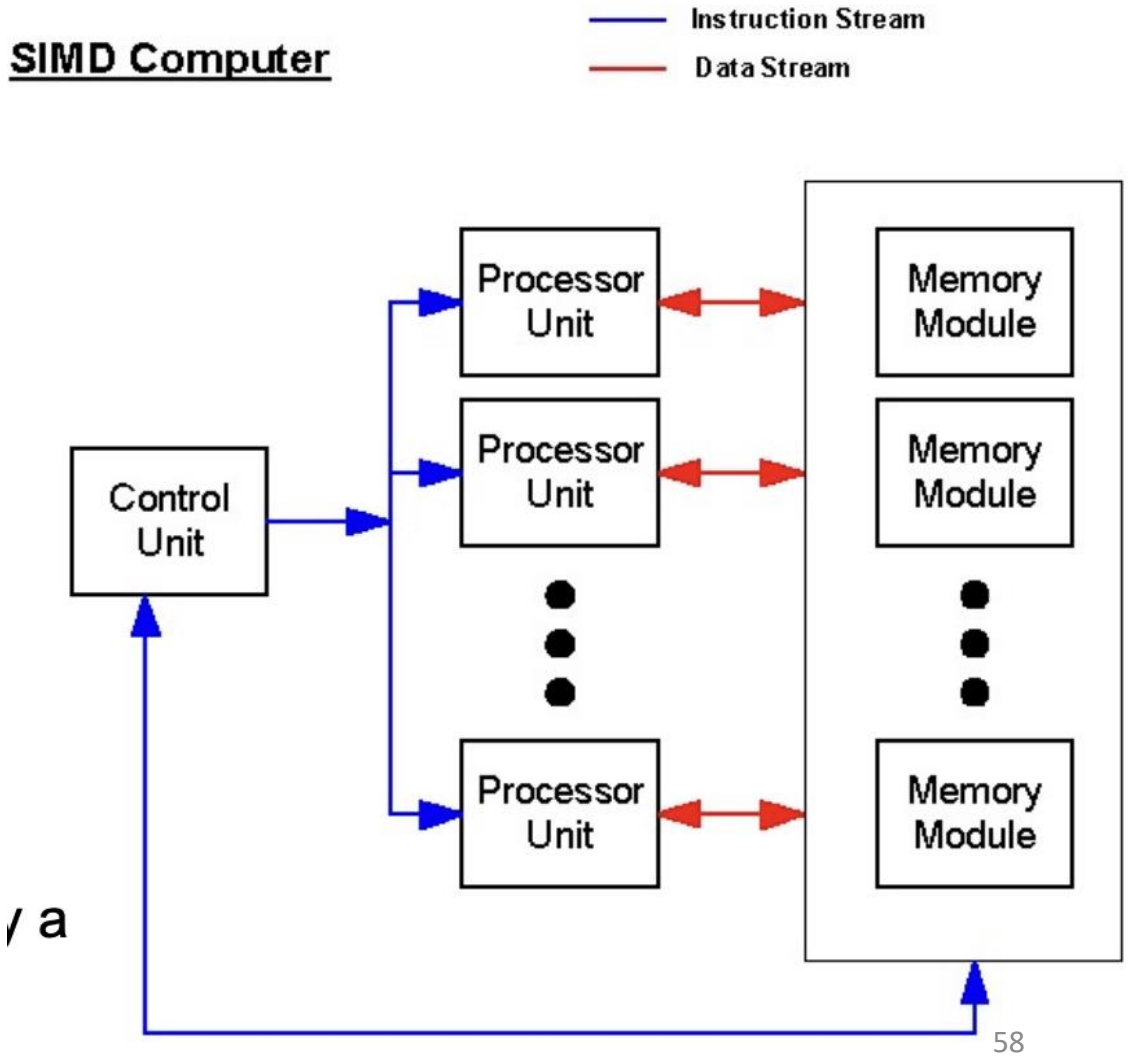
These still exist, although the term has become less popular ...

Computer Architecture Taxonomy - Flynn's Taxonomy

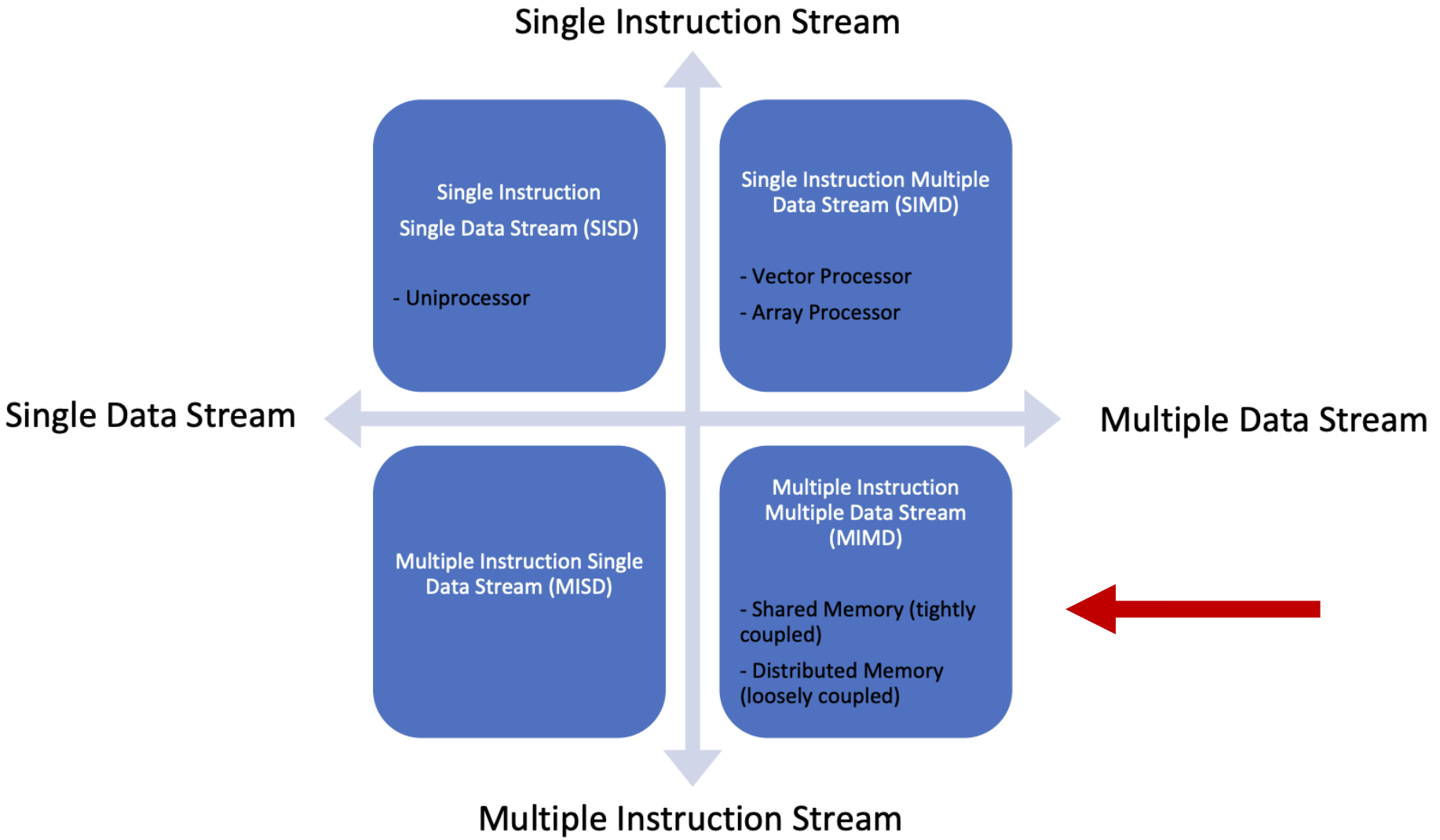


Single Instruction Multiple Data (SIMD)

- Single machine instruction controls simultaneous execution of same instruction on many data items
- Number of processing elements each with associated data memory
- Each instruction executed across data by different processor units
- Vector & array processors (for graphics)
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.

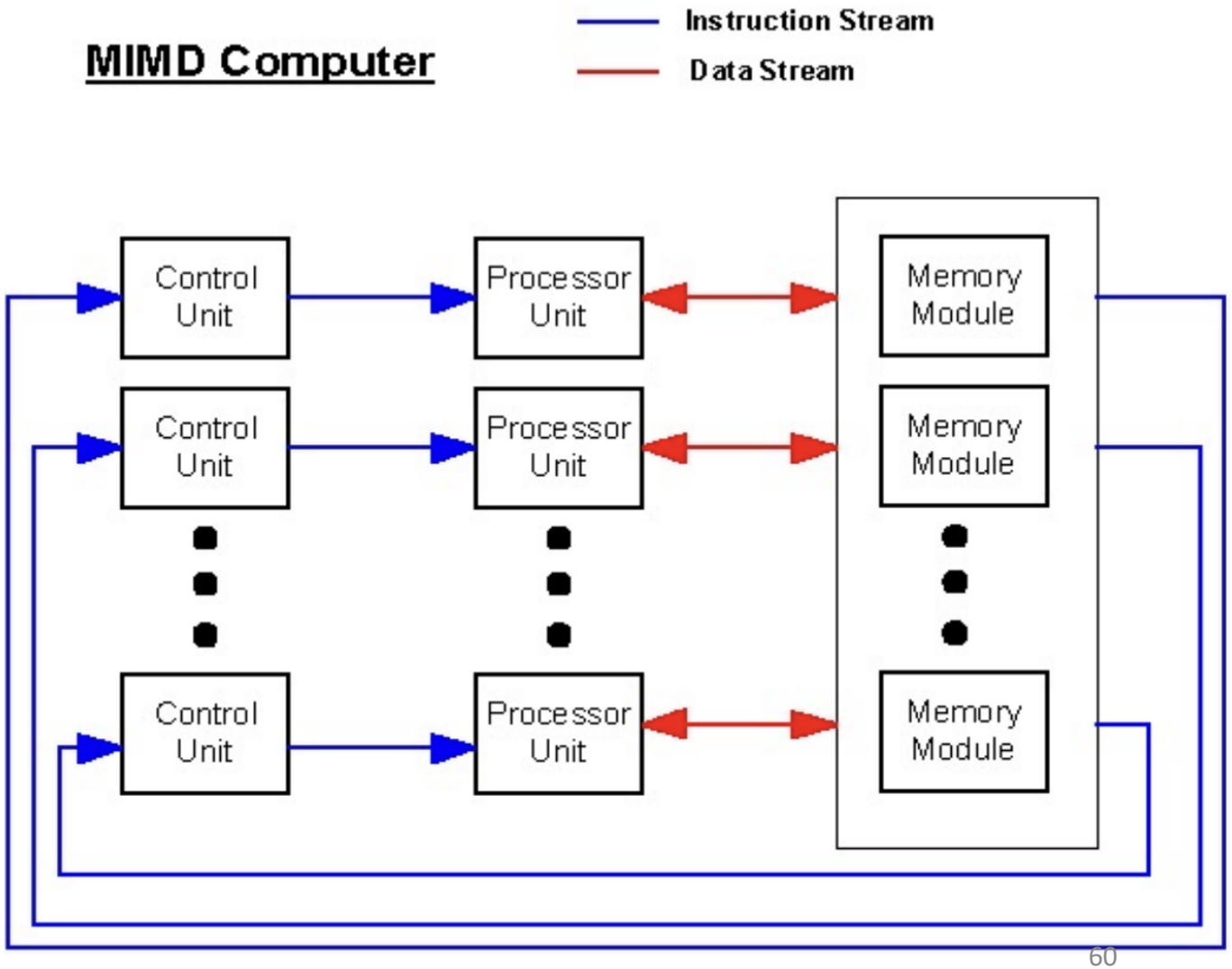


Computer Architecture Taxonomy - Flynn's Taxonomy

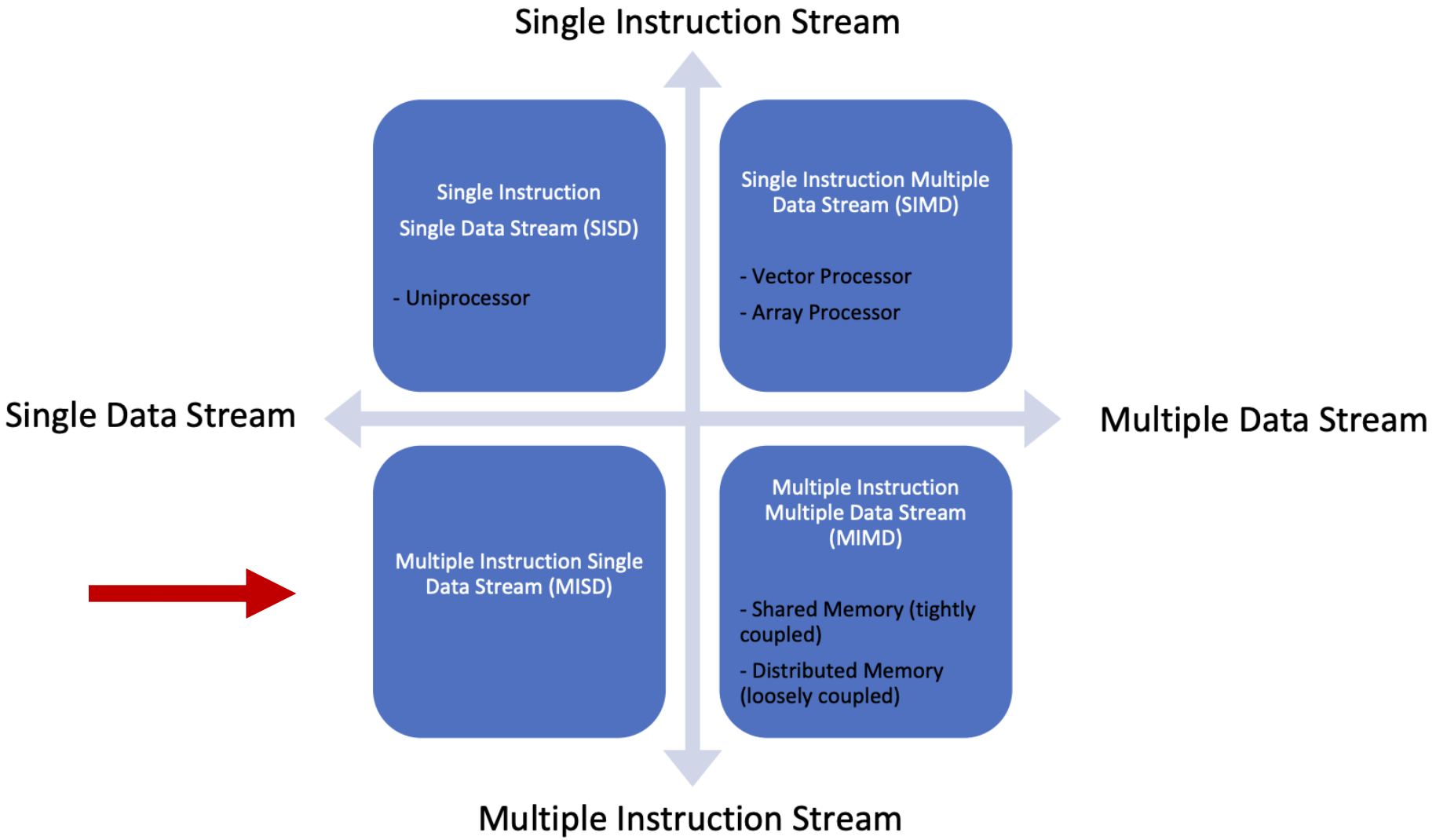


MIMD Multiple Instruction Multiple Data

- Set of processors
- Simultaneously execute different instruction sequences on different data
- SMPs, clusters & NUMA systems (more later)
- Supercomputers use MIMD with SMPs for specific tasks.

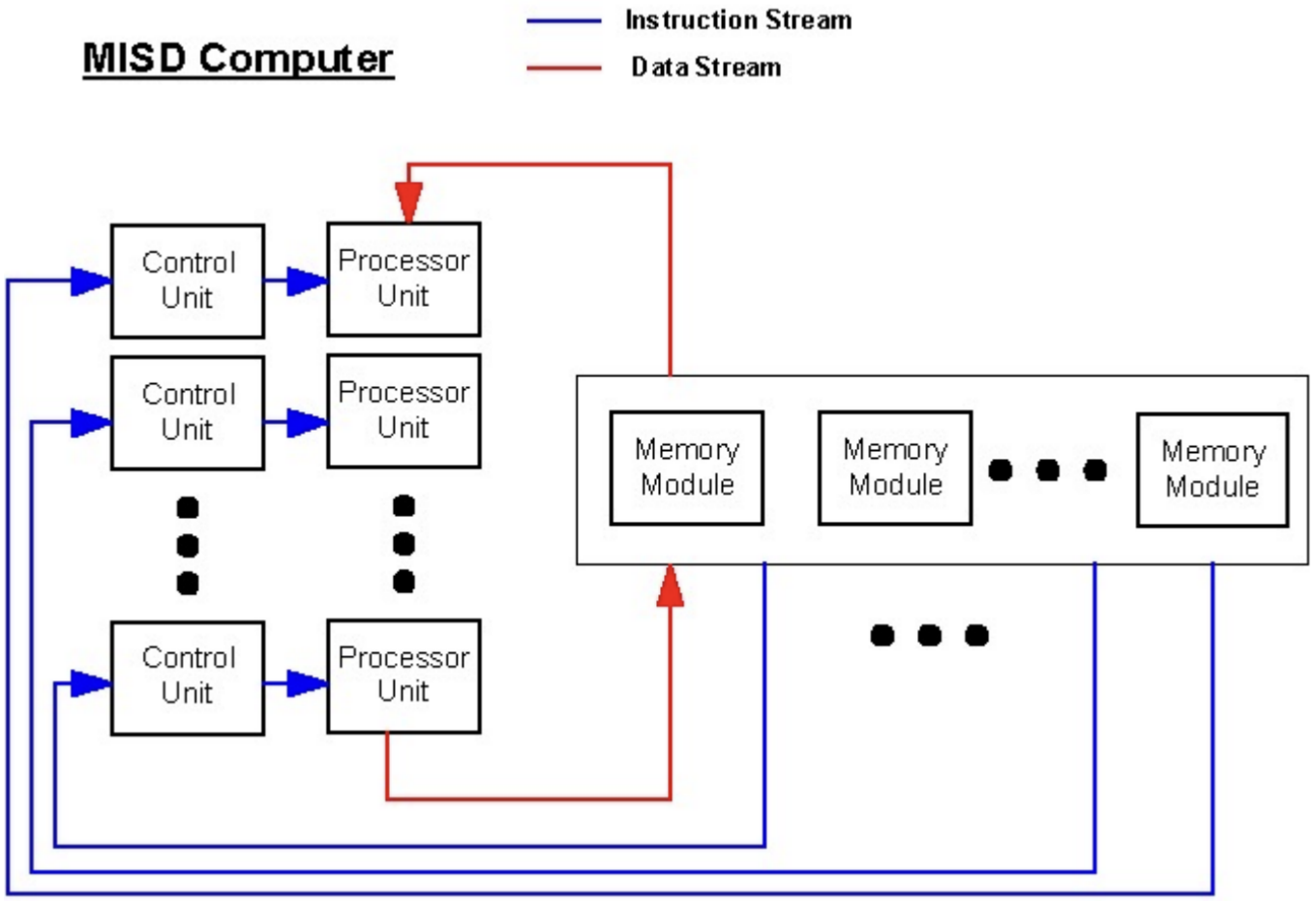


Computer Architecture Taxonomy - Flynn's Taxonomy



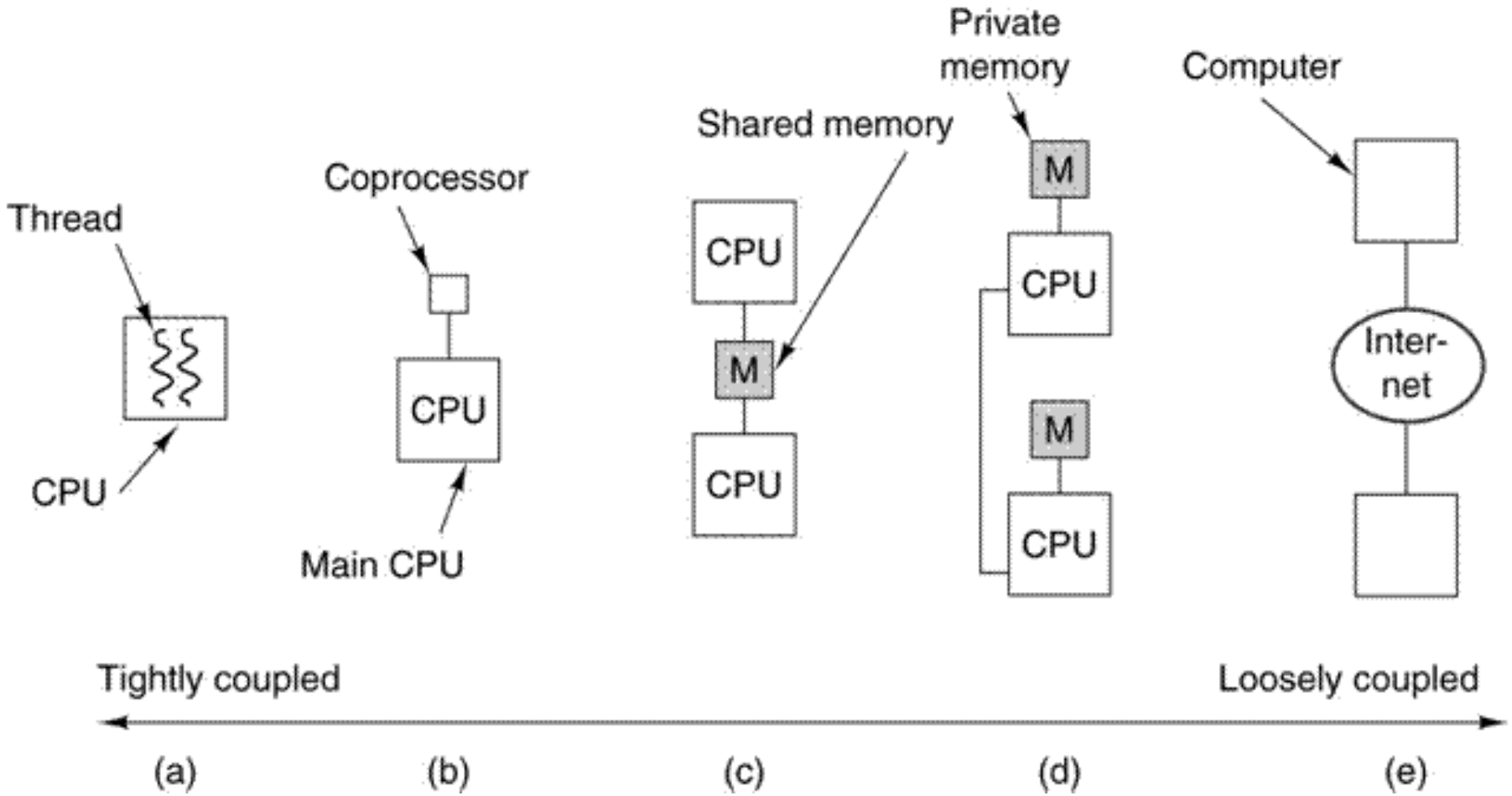
MISD Multiple Instruction Single Data

- Multiple instructions – one data stream
- Each processor executes different instruction sequence
- E.g., for fault tolerance on space shuttle/avionics system
- Mainly theoretical



- Few actual examples of this class of parallel computer have ever existed.
- This does not mean that MISD is dead though. All we are waiting on is a paradigm shift that allows MISD to become a viable reality • E.g., quantum computing?

Parallelism Levels



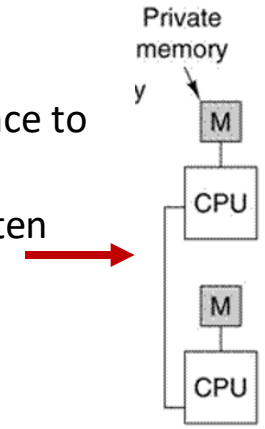
More on MIMD (Parallel Computing)

Architectures further classified by method of processor communication / interdependence.

Tight Coupling:

- **Symmetric Multi-Processing (SMP)**
  - Processors share memory & communicate via that shared memory
  - Fair access time to given area of memory between processors

- **Non-uniform Memory Access (NUMA)**
  - Varying access times to memory regions with distance to processor
  - Good only for particular workloads, e.g. data are often associated strongly with certain tasks or users

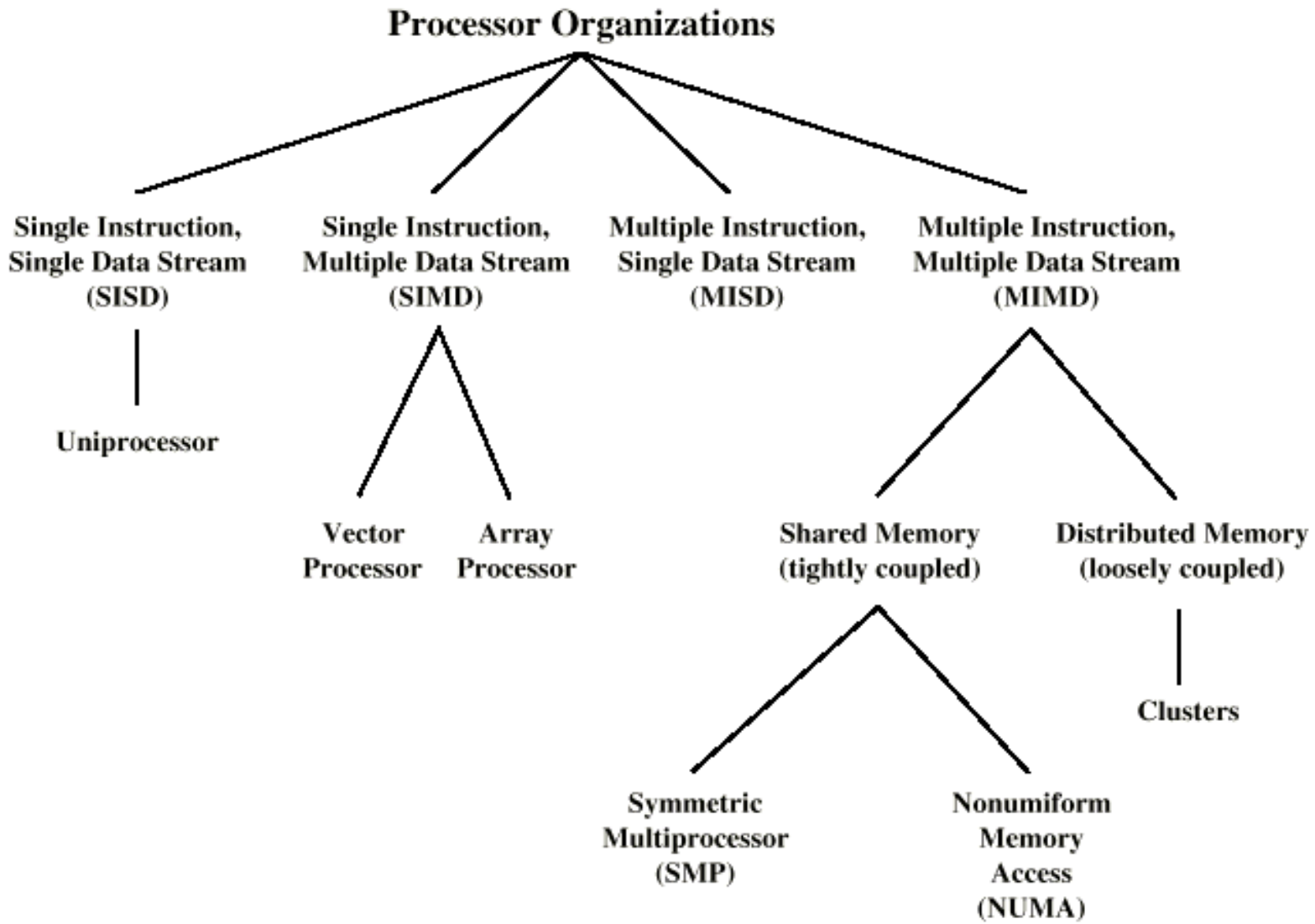


...

Loose Coupling: e.g., Clusters

- Collection of independent nodes (e.g. SMPs)
- Interconnected to form a cluster
- Often used as unified resource/ different users using partitions
- Communication via fixed path or network connections
- Alternative to SMP giving high performance & high availability
- E.g. batch jobs

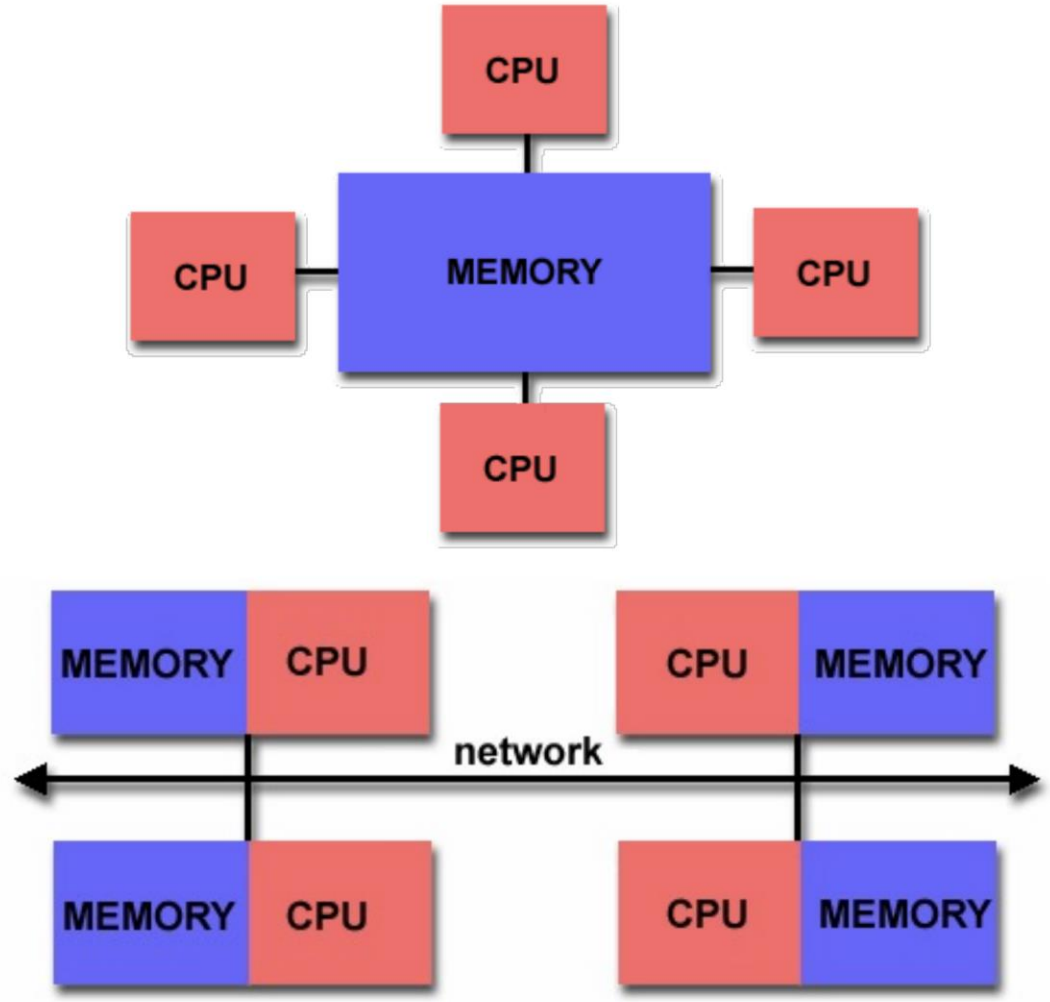
Computer Architecture Taxonomies for Concurrency



Memory architectures

- There are only three memory architectures
  - Really, there are two
  - the third is a combination of the first two!

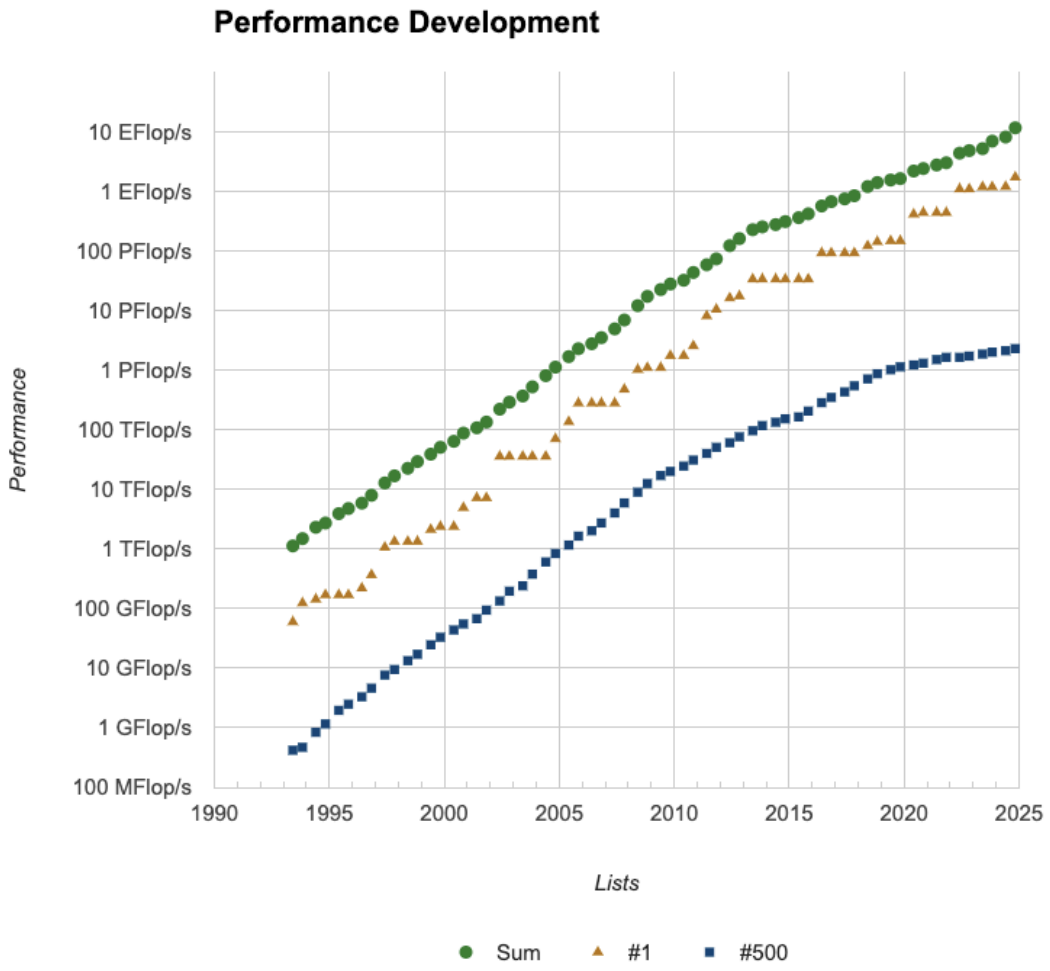
- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory



Pros and Cons (Distributed Memory)

- Advantages
  - Memory is scalable with number of processors.
    - Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages:
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization (tightly coupled just may be tightly coupled)
  - Like Non-uniform memory access (NUMA) times

TOP 500 (November 2024)



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory	9,066,176	1,353.00	2,055.72	24,607
3					38,698
4					
5					8,461
6					29,899

top500.org

Performance, Scalability: Metrics & Barriers

The Ideal vs. The Real World…

Ultimately, we would like system **throughput** to be directly proportional (and **latency** inversely proportional) to the number of CPUs**.**

Unfortunately, this ‘perfect’ scenario is impossible to realise for various reasons:

- Poor Design (how problem is broken down & solved);
- Code Implementation (I/O, inefficient use of memory…);
- Some parts of the code are inherently sequential
- Operating System overhead / Coordination overhead of concurrent code
  - e.g. to avoid Race Conditions;
- Etc., etc., …

Metrics for Performance

- Time spent on a task (i.e. *latency, units* [time, T])
  - Note: not just for communication tasks i.e. can talk of latency of task processing
- Rate at which tasks are completed (*throughput*, units T^{-1}).

- *Power* consumed on a calculation
- *Platform cost* required for the computation
- How effectively computational power used in parallel program (*Efficiency*)?

…

Performance: Latency vs Throughput

- **Latency** = how long to execute a unit of work
  - e.g., how long to cook a dinner in a restaurant (from order to plate)

- **Throughput** = how many units of work a system can do per unit time
  - e.g., how many dinners are served per hour in the restaurant

Scalability

- Performance:
  - How much **faster** can a given problem be solved with N workers instead of one?
  - How much **more work** can be done with N workers instead of one?

- *Strong Scaling*: same problem size, add more workers/processors
  - **Goal**: Minimize time to solution for a given problem
- *Weak Scaling*: same work per worker, add more workers/processors
  - **Goal**: solve larger problems.

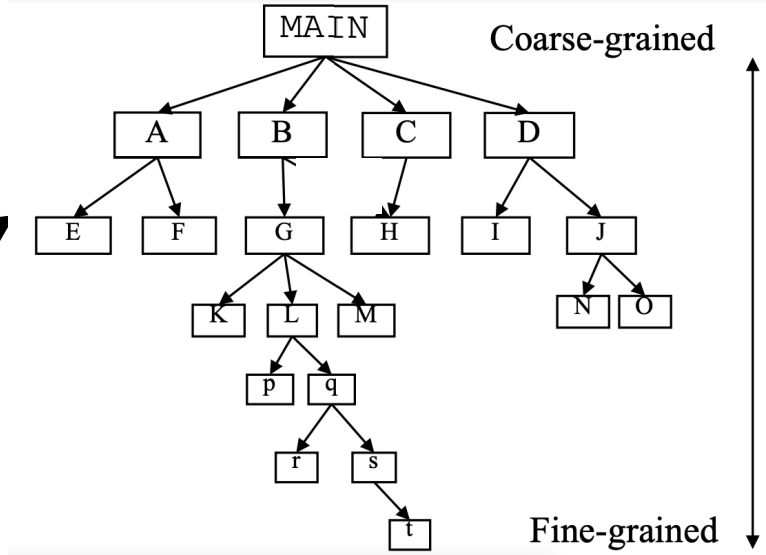
Barriers to Scalability

Classify applications on how often their subtasks must synchronize/inter-communicate:

Fine-grain parallelism - *exhibited if application subtasks must communicate often e.g. multiple times per second*

- Small amounts of computational work between communication stages
- Low computation to communication ratio
- Less opportunity for performance enhancement
- High communication bandwidth

- **Coarse-grain parallelism** - *if they don't communicate as often*
  - Many bigger tasks
  - More opportunity for performance increase
  - Harder to balance efficiently (load balancing)



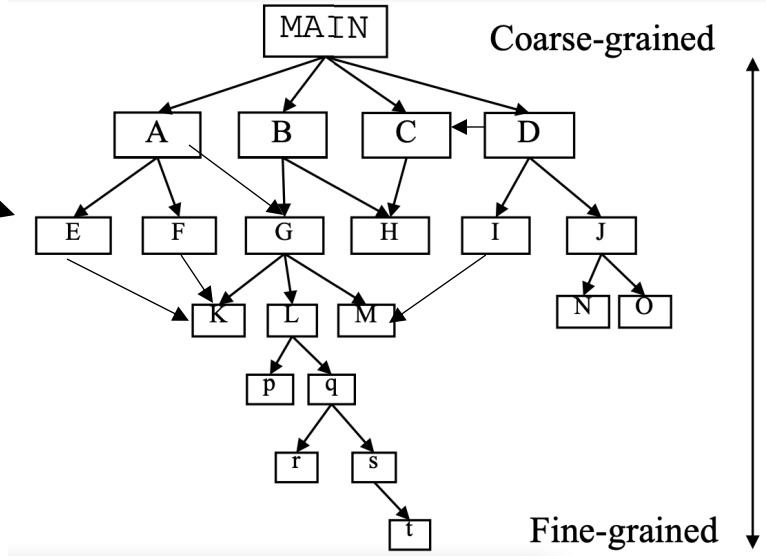
- **Embarrassing parallelism**, *if they rarely/never have to communicate*.
  - Ideal case

Barriers to Scalability

Classify applications on how often their subtasks must synchronize/inter-communicate:

Fine-grain parallelism - *exhibited if application subtasks must communicate often e.g. multiple times per second*

- Small amounts of computational work between communication stages
- Low computation to communication ratio
- Less opportunity for performance enhancement
- High communication bandwidth



- **Coarse-grain parallelism** - *if they don't communicate as often*
  - Many bigger tasks
  - More opportunity for performance increase
  - Harder to balance efficiently (load balancing)

- **Embarrassing parallelism**, *if they rarely/never have to communicate*.
  - Ideal case

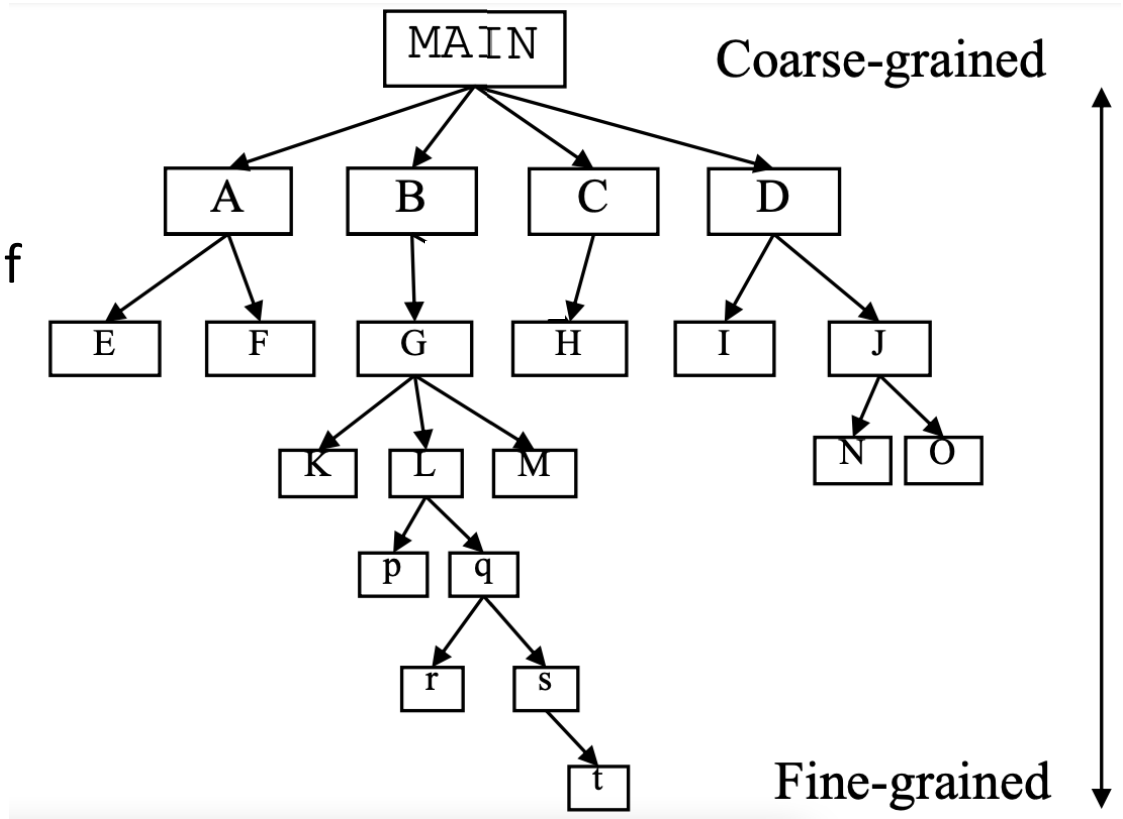
Barriers to Scalability

Fine-grain parallelism (typically loop level)

- (generally) Does not require deep knowledge of the code
- Many loops must be parallel for decent speedup
- Potentially many **synchronization points** (at end of each parallel loop)

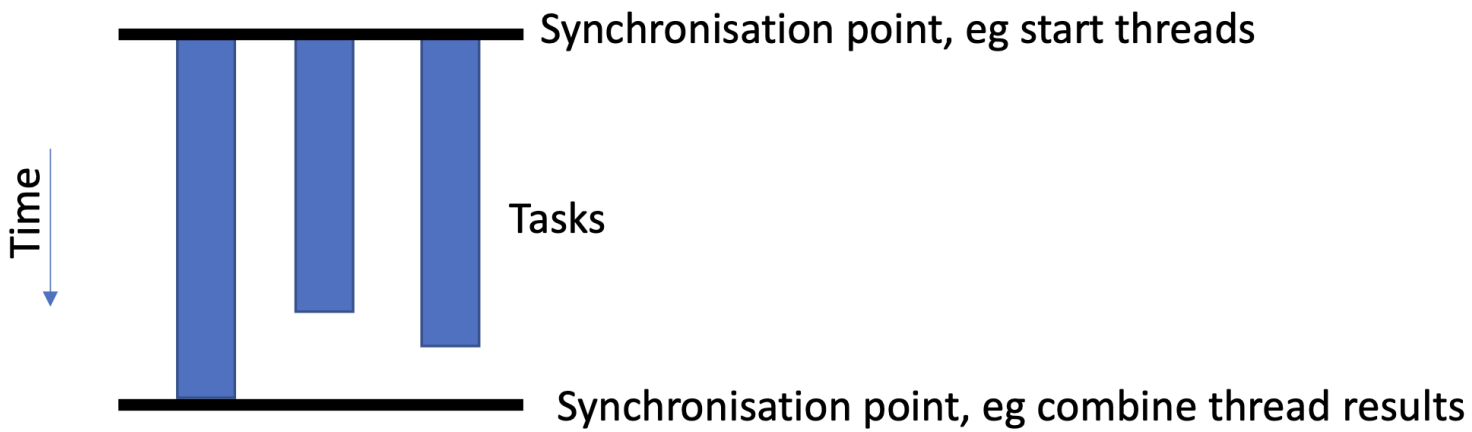
Coarse-grain parallelism

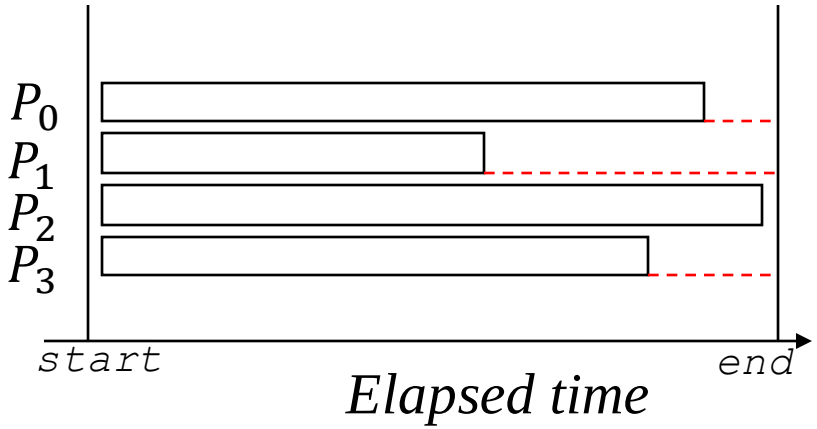
- Make larger loops parallel at higher call-tree level potentially in-closing many small loops
- More code is parallel at once
- Fewer **synchronization points**, reducing overhead
- Needs deeper knowledge of code
- Snag: load balancing



Barriers to Scalability: Load Balancing Problem

- Tasks that finish first need to wait for the processor with the largest amount of work to finish =>idle time





- Unequal work load distribution leads to idle processors, others work too much
- Coarse-grain parallelization, can lead to **load imbalance**

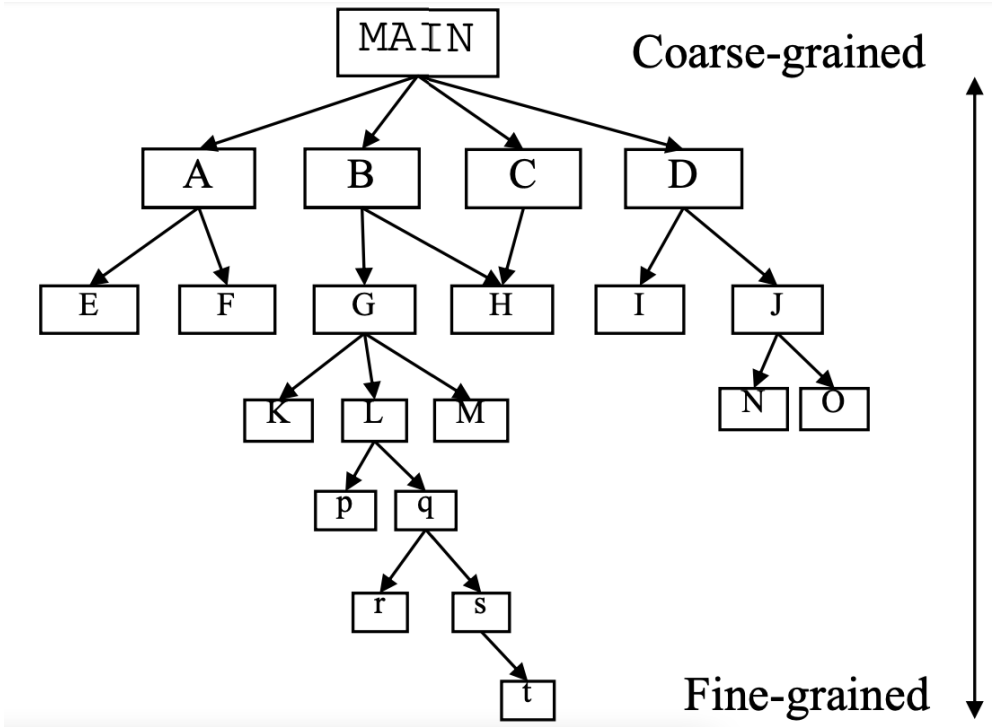
Barriers to Scalability: Synchronisation Points

Are there ordering constraints on execution?

- Depends on how you cut up data or allocate tasks to processors

- Potentially many synchronisation points
  - (at end of each parallel loop)

- If too many small loops have been made parallel, synchronization overhead will compromise scalability.



Amdahl’s Law for Single & Multi-Core Systems

Amdahl's Law

Amhdal's law identifies performance gains from adding additional cores / processors to an application that has both serial and parallel components

- *S* is serial portion
- *N* processing cores

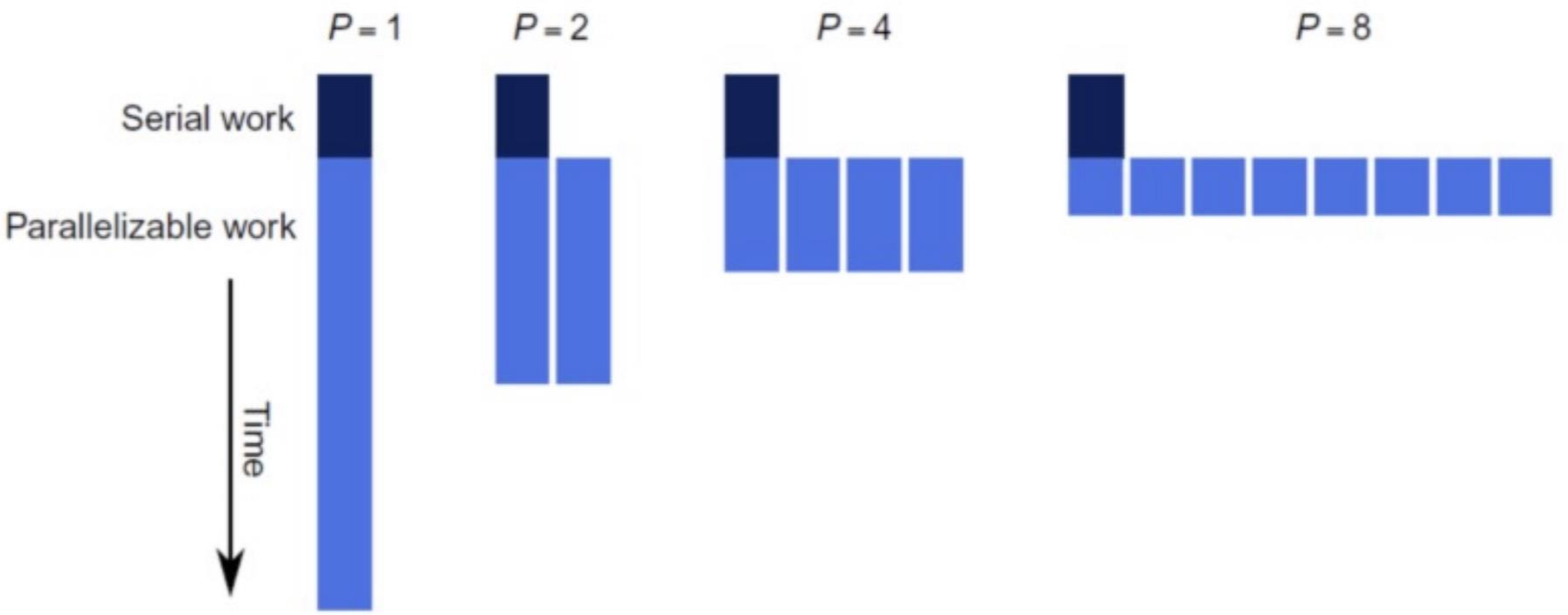
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

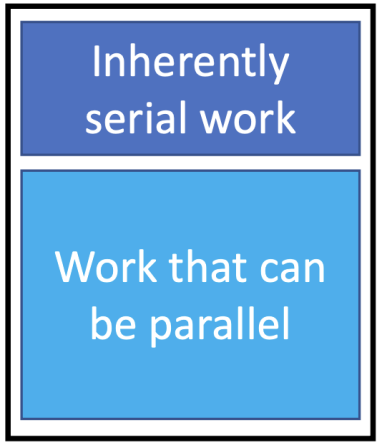
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

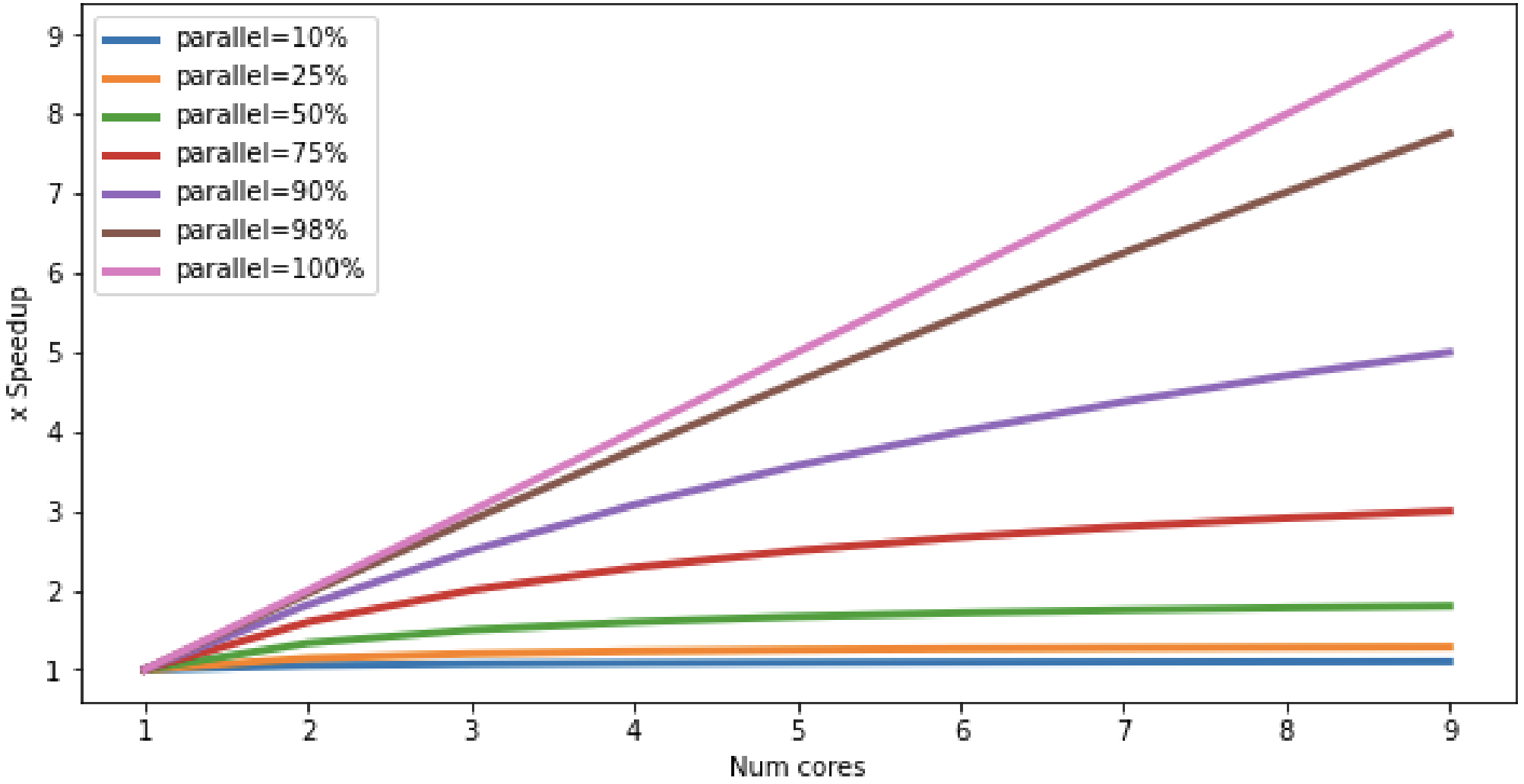
Amdahl's Law

This can be better seen in the following schematic:

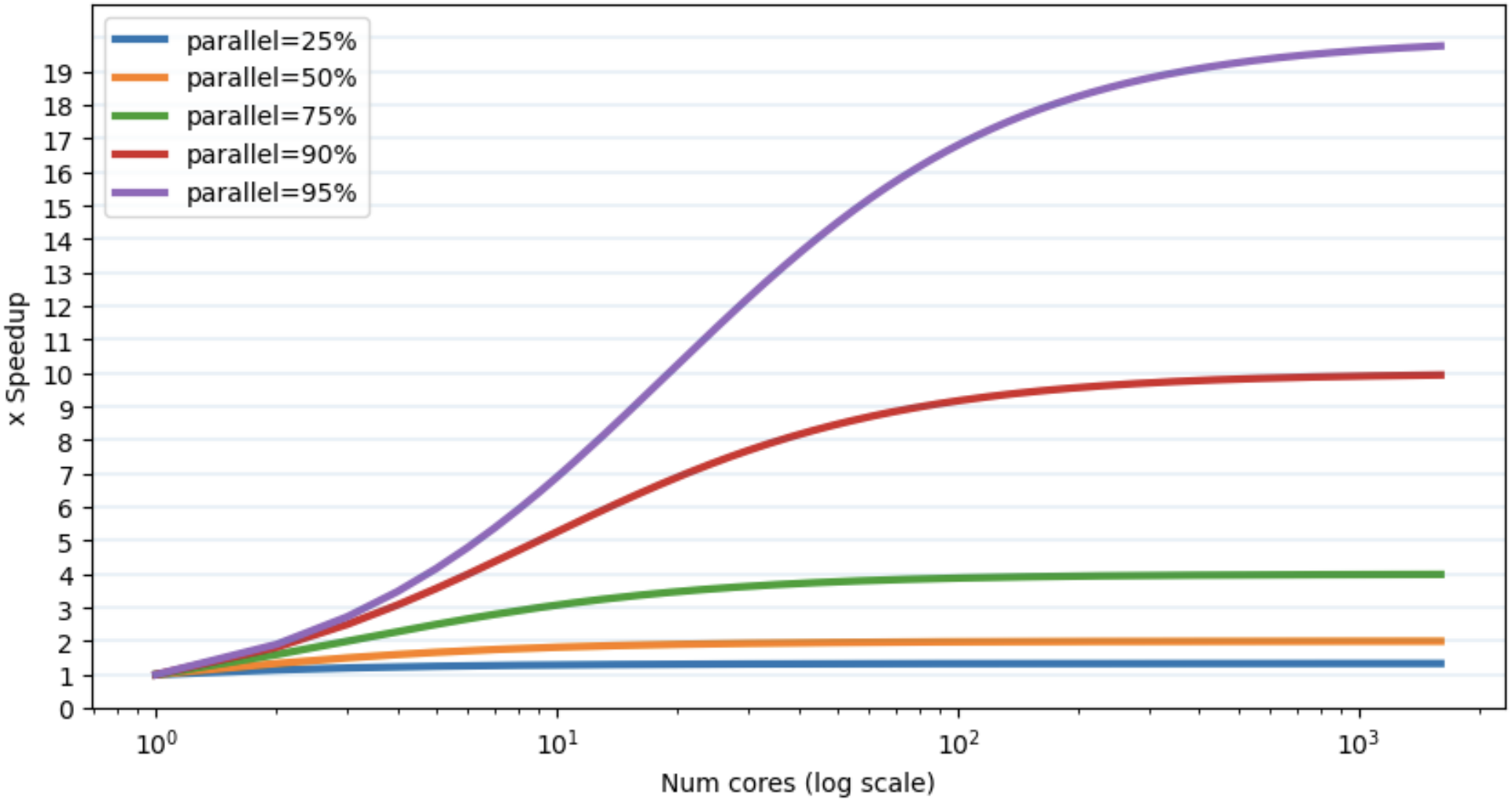




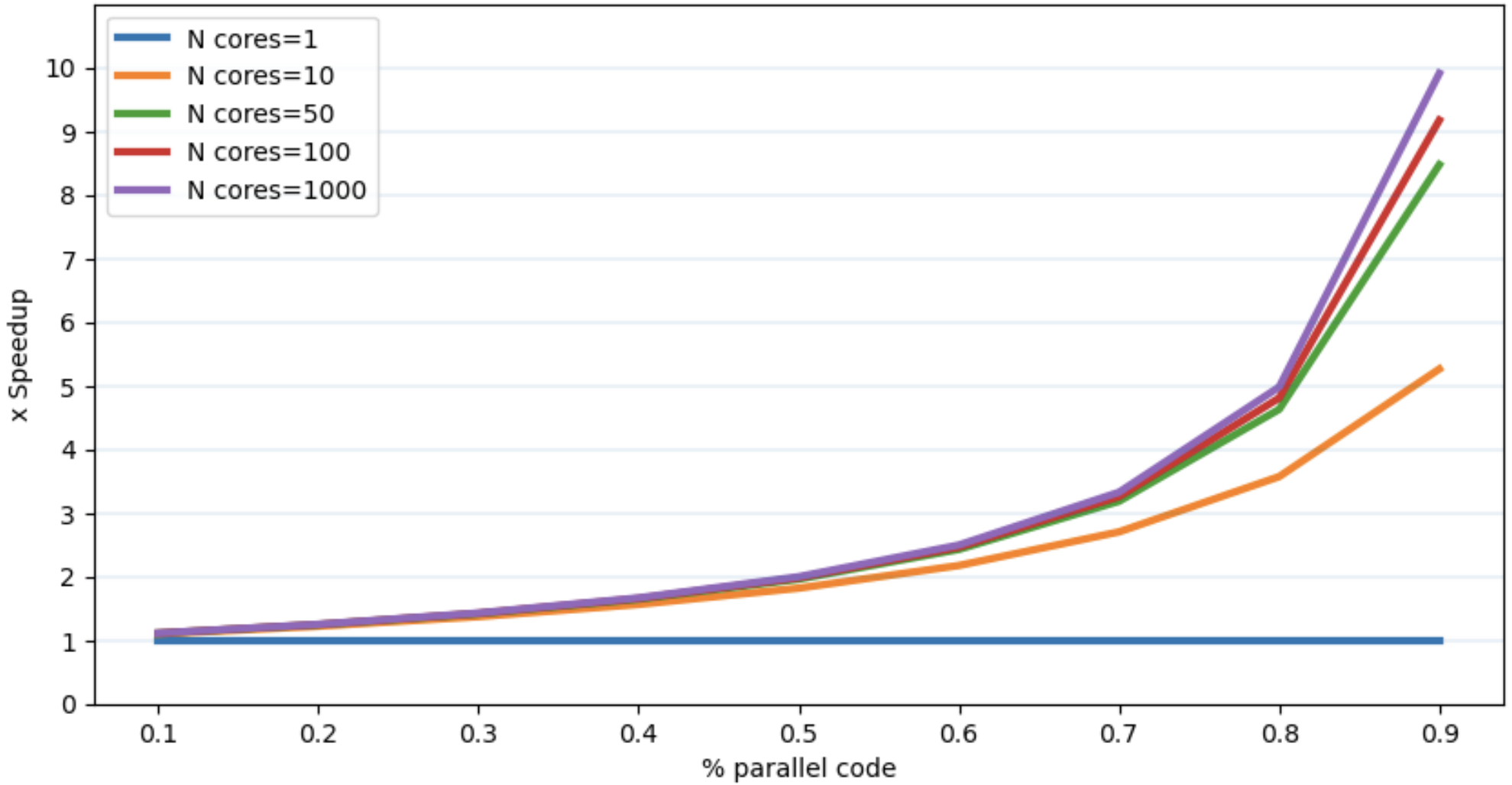
Amdahl's law


parallel=10%
parallel=25%
parallel=50%
parallel=75%
parallel=90%
parallel=98%
parallel=100%
x Speedup
Num cores
1
2
3
4
5
6
7
8
9

Amdahl's law


parallel=25%
parallel=50%
parallel=75%
parallel=90%
parallel=95%
x Speedup
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
10^0
10^1
10^2
10^3
Num cores (log scale)

Amdahl's law


N cores=1
N cores=10
N cores=50
N cores=100
N cores=1000
x Speedup
0
1
2
3
4
5
6
7
8
9
10
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
% parallel code

Amdahl's Law

The Sandia Experiments

- **Karp prize** for first program to achieve a speed-up of 200 or better.
- In 1988, Sandia reported speed-up > 1,000 on a 1,024 processor system
- This was on 1,024 processor system used on three different problems.
- How is this possible?

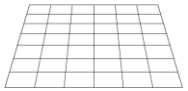
For example: Calculations on a 2D Grid

Regular Problem Size Timings:

- Grid Calculations: 85s (85%)
- Serial fraction: 15s (15%)

Double Problem Size Timings:

- 2D Grid Calculations: 680s (97.84%)
- Serial fraction: 15s (2.16%)



Assumptions:

- Amdahl assumed serial fraction **s**, independent of problem size.
- However, as problem size increases inherently serial parts of program stay same (or increase more slowly than problem size).
- So, when problem size n increases (↑)
  - ratio of serial parts **s(n)** decreases (↓)
  - and potential Speedup increases (↑)

So, Amdahl's law should have been:

$$Speedup \leq \frac{1}{s(n)}$$

Gustafson-Barsis' Law

- Gustafson-Barsis' Law:
  - Argues that it is rare to have a fixed problem size when N (num of processors) increases (i.e., strong scalability)
  - Therefore, speedup should be measured by scaling the problem to number of processors (weak scalability), not by fixing problem size.

- Gustafson-Barsis' Law shows the limits of weak scalability:

$$S' = \alpha + (1 - \alpha)\, N$$

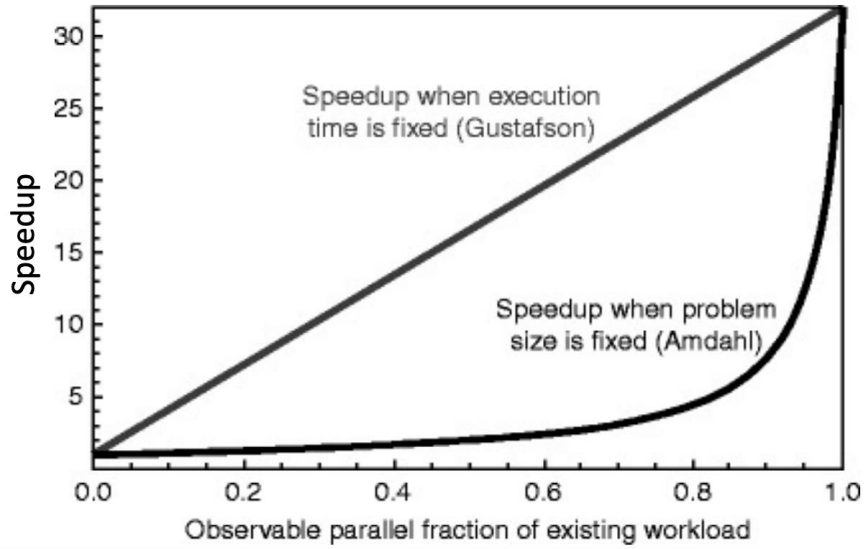
Where:

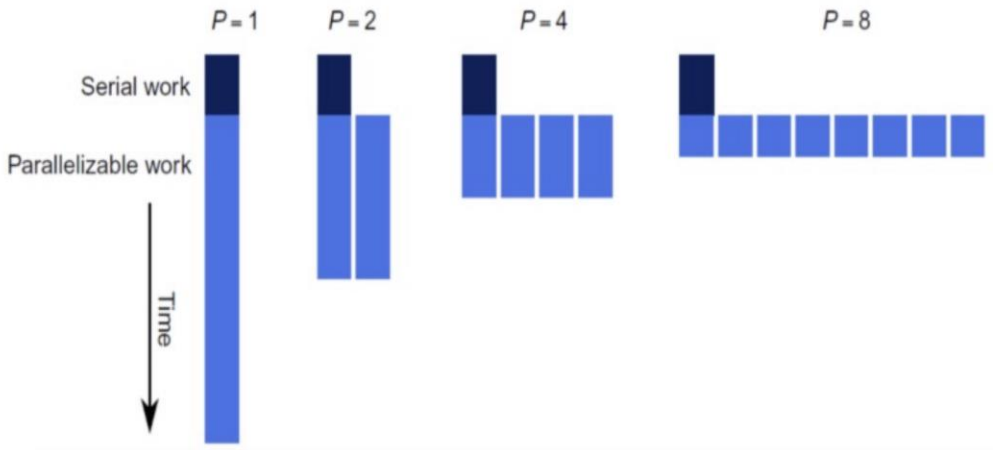
S: Speedup measure by Amdahl's Law

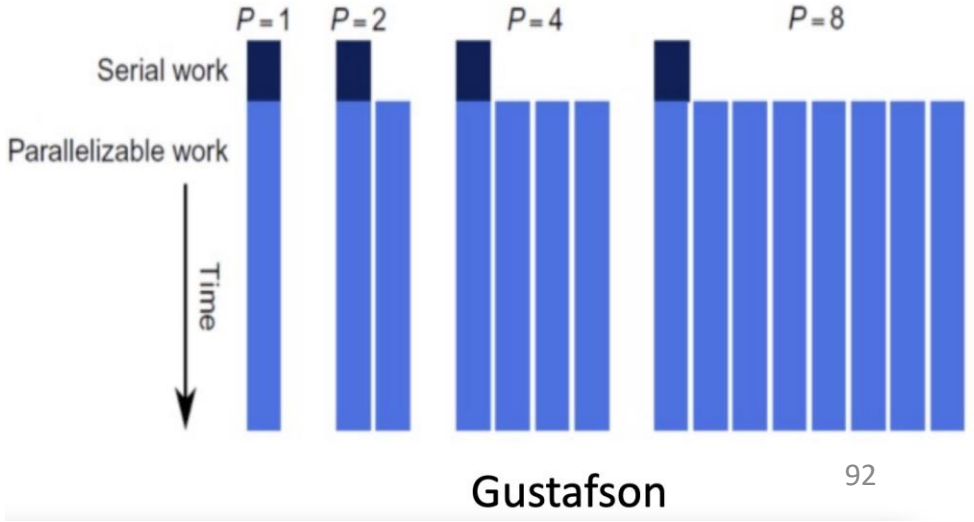
S' is the Scaled Speedup
α is serial proportion
N = Number of processors

Gustafson-Barsis' Law

- Illustrations of Gustafson's Law
  - Considers speedup in terms of throughput - execution time is fixed
  - i.e., more parallel code allows more work to be done **in the same time**







Take Home Message

- **Concurrency** is about dealing with lots of things at once. **Parallelism** is about doing lots of things at once.
- Terms such as Multi-tasking, Multi-core, Multi-threading (both implicit & explicit) important in concurrent systems.
- Flynn's classification is established but essential architectural classification system in concurrency.
- When coding in parallel Functional/Task & Domain/Data Decomposition should be considered.
- Performance of systems defined by **throughput**, **latency**, ...
- Ability of programs to scale is important but many barriers exist.
- Amdahl's law is a simple way to account for some of these barriers.