# Computer Organization and Architecture

Lecture – 26

Nov 16th , 2022

# Chapter 6

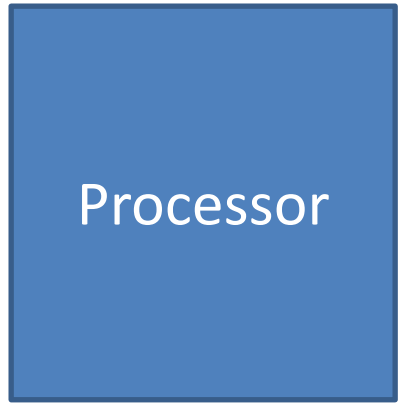
## Parallel Processors from Client to Cloud

# Performance

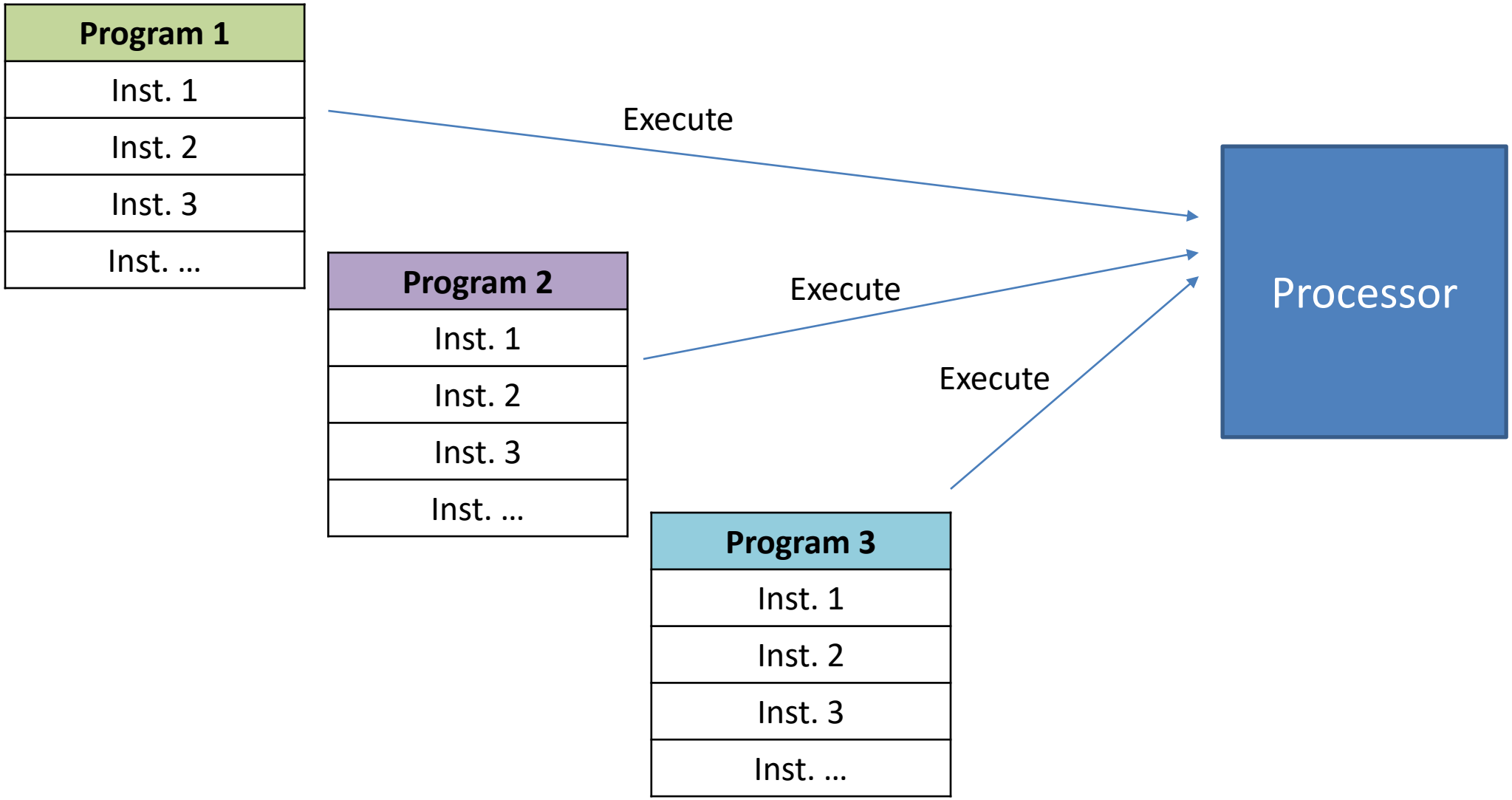
| Program 1 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

| Program 2 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

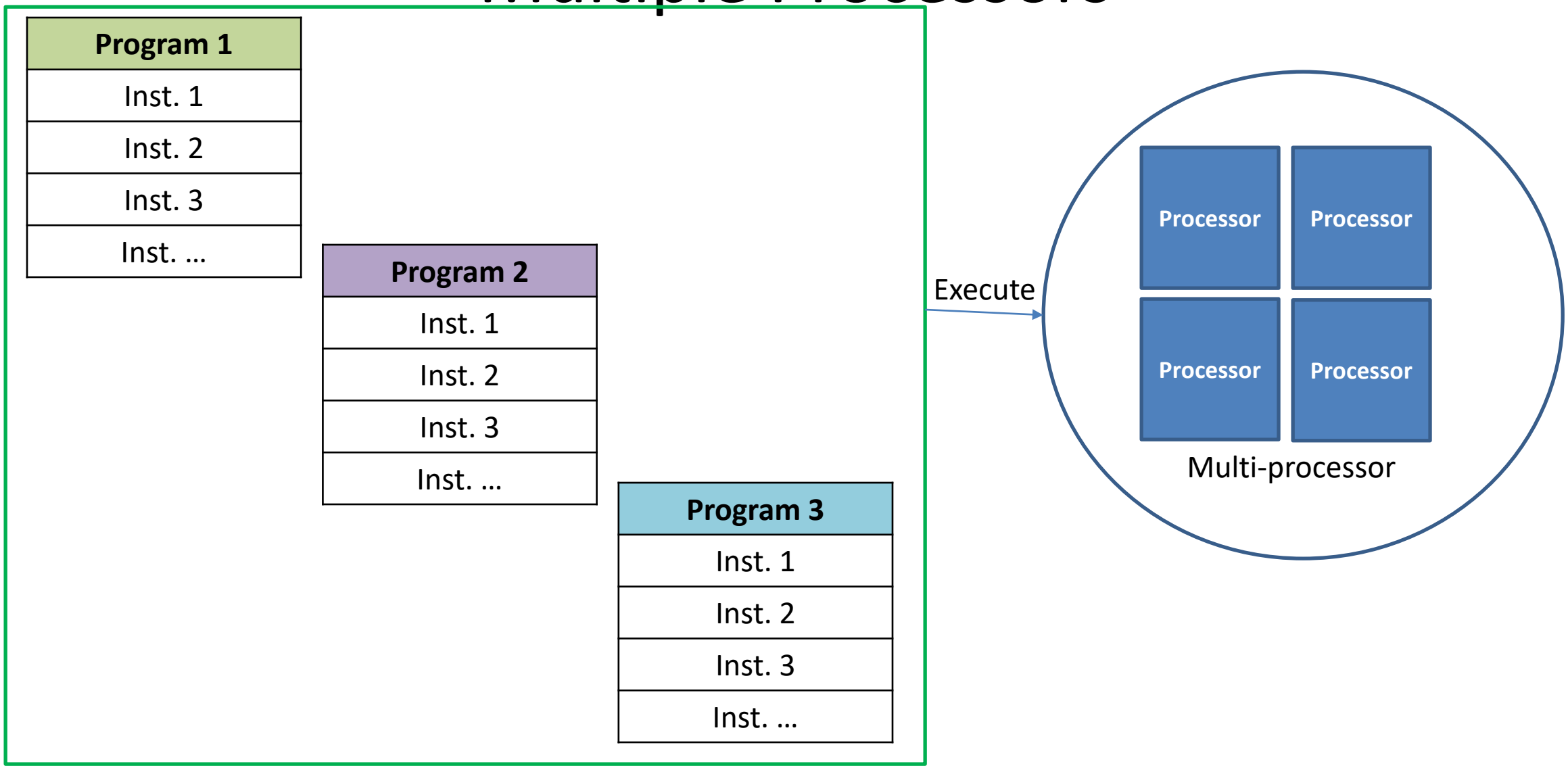
| Program 3 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |



# Performance



# Multiple Processors



# Introduction

- Goal: connecting multiple computers to get higher performance

# Sequential Vs. Concurrent Vs. Parallel

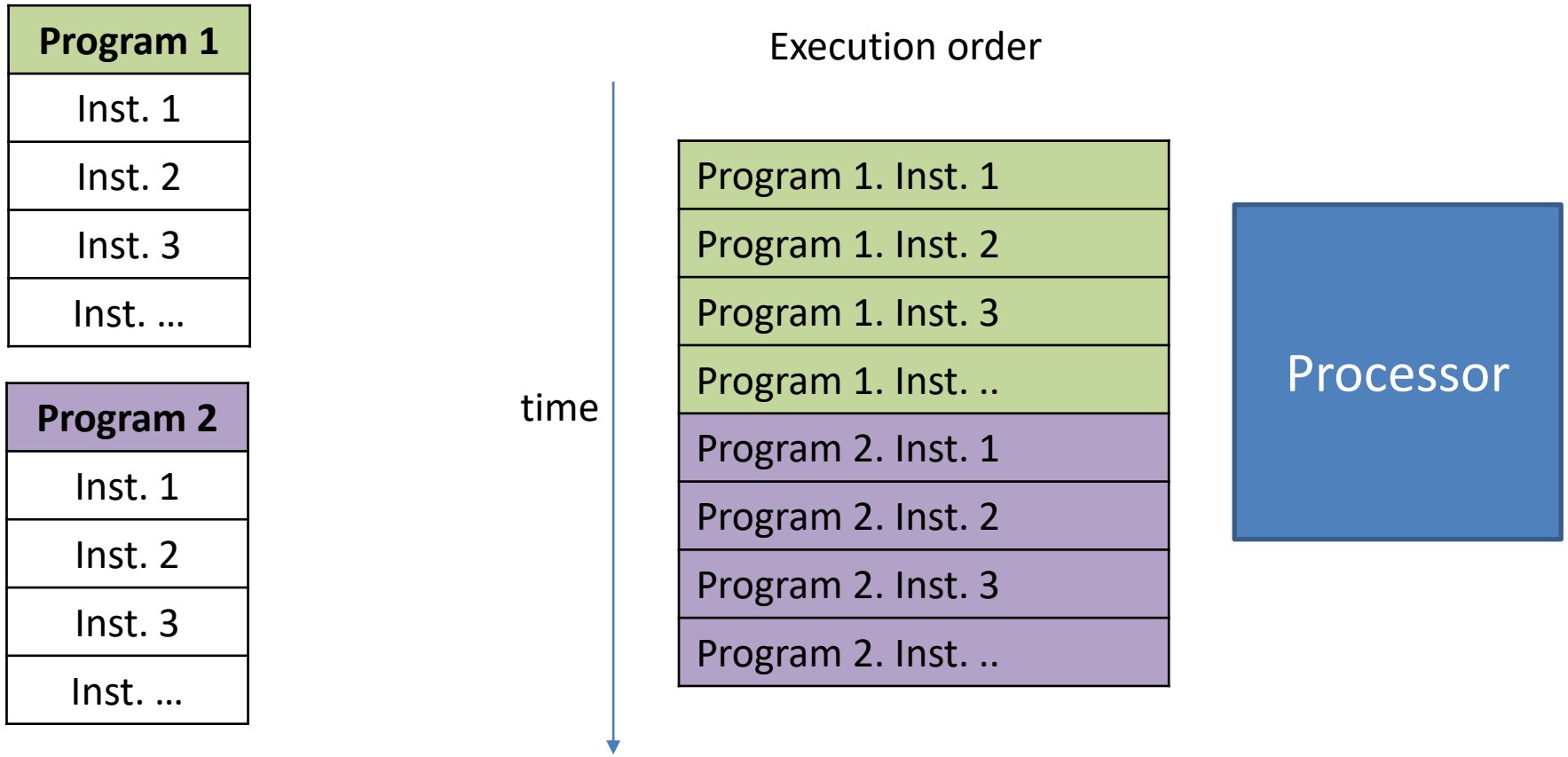
# Sequential Vs. Concurrent Vs. Parallel

| Program 1 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

| Program 2 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

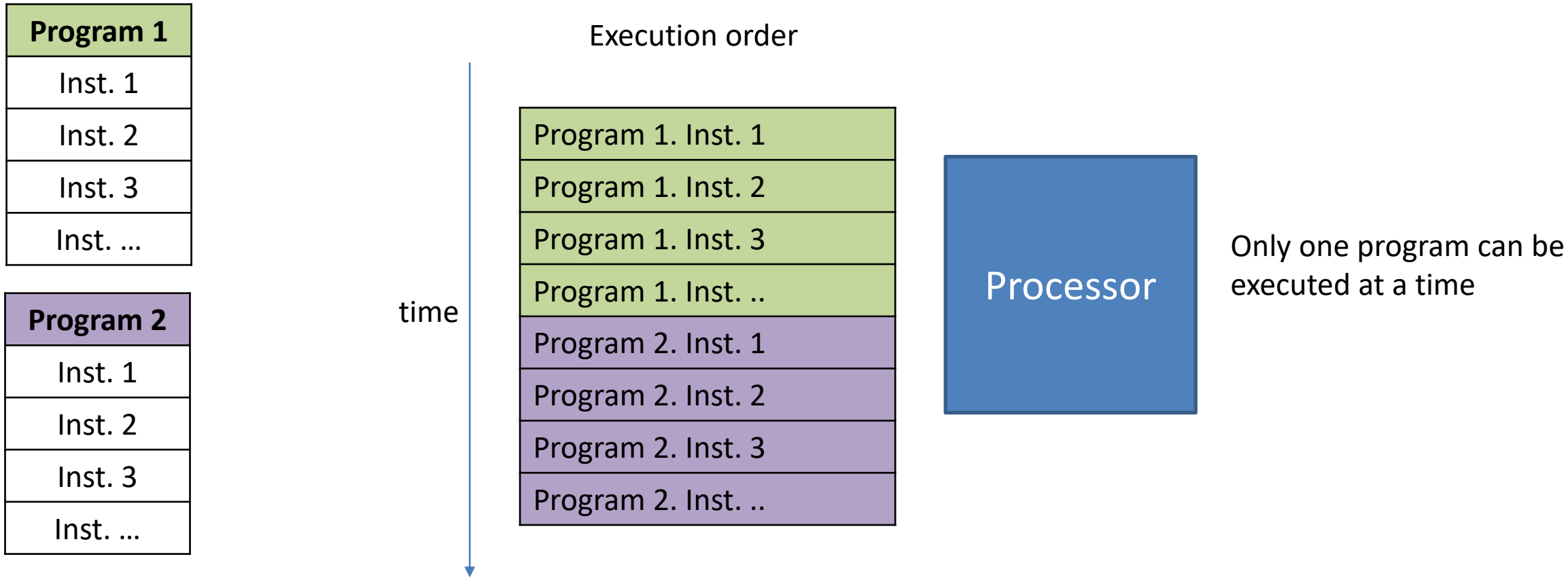

**Sequential**

# Sequential Vs. Concurrent Vs. Parallel




**Sequential**

# Sequential Vs. Concurrent Vs. Parallel




**Sequential**

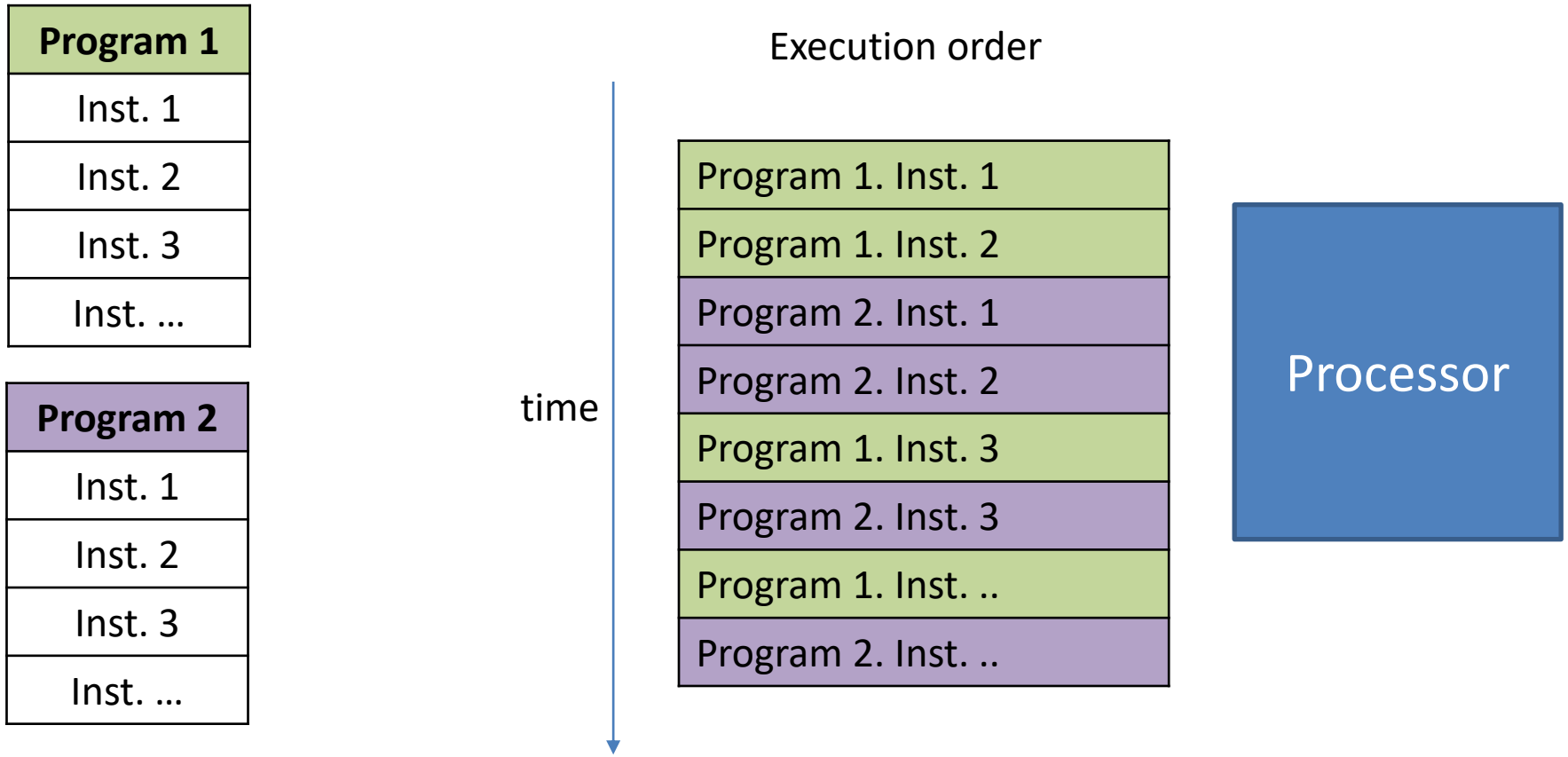
# Sequential Vs. Concurrent Vs. Parallel

| Program 1 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

| Program 2 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

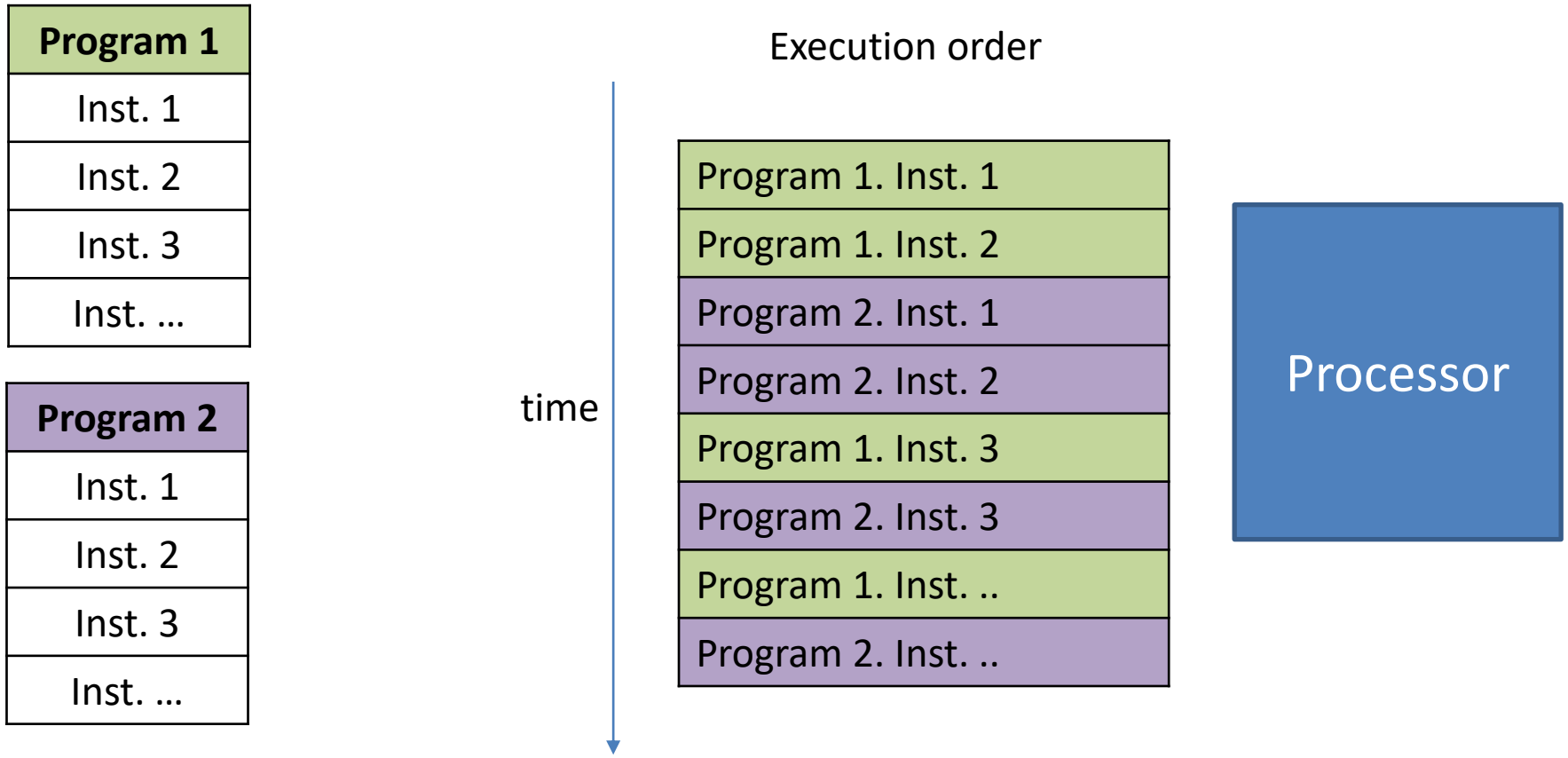

**Concurrent**

# Sequential Vs. Concurrent Vs. Parallel




**Concurrent**

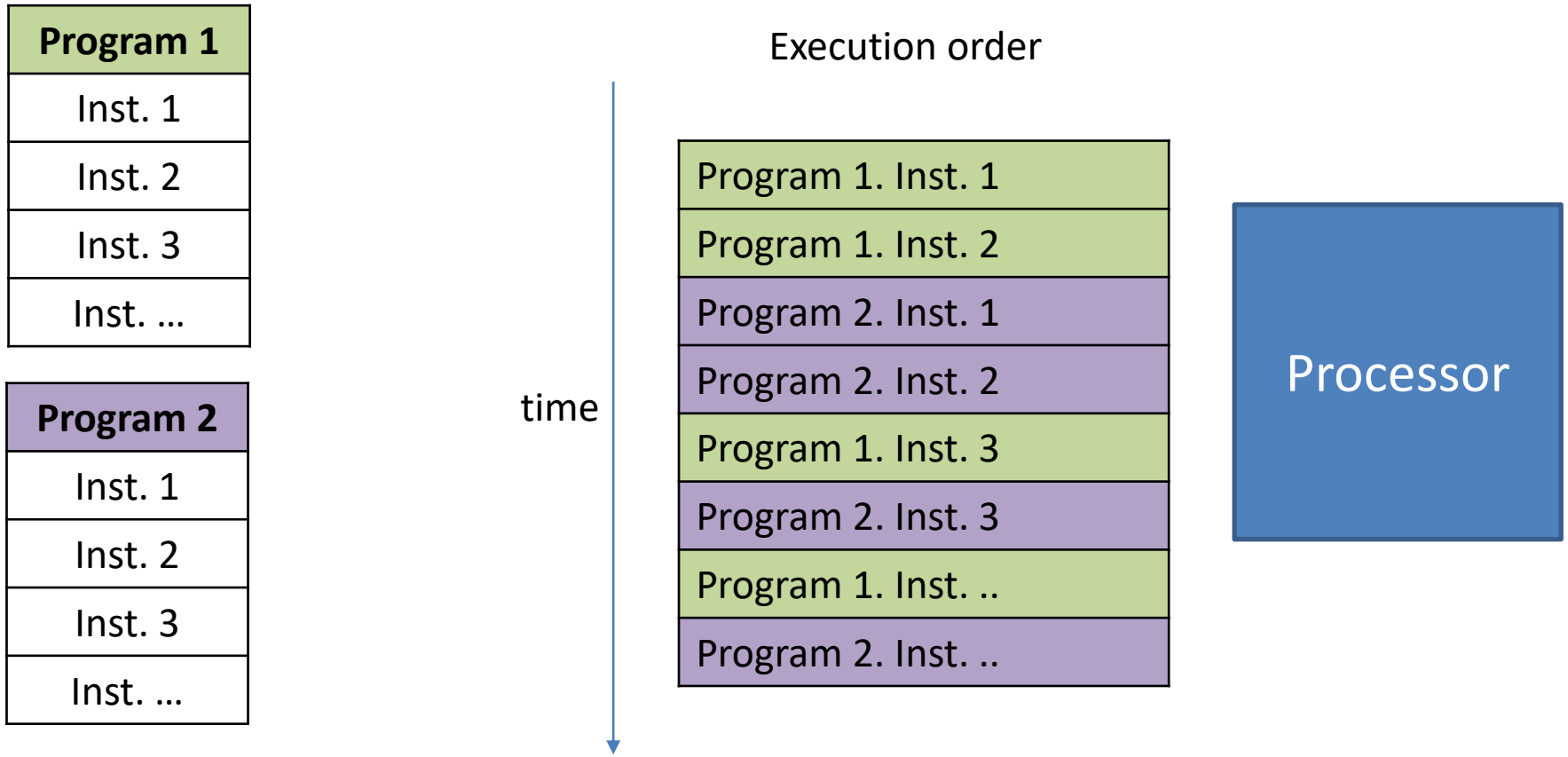
# Sequential Vs. Concurrent Vs. Parallel




1. Instructions from two programs are executed concurrently
2. Creates an illusion that the two programs are running in parallel, event though only one program is executed at a given instance of time.
3. Allows us to execute multiple programs simultaneously.

**Concurrent**

# Sequential Vs. Concurrent Vs. Parallel



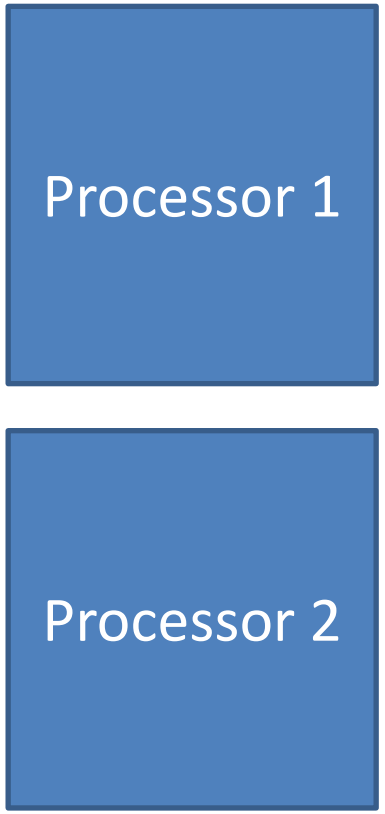

**Concurrent**

1. Instructions from two programs are executed concurrently
2. Creates an illusion that the two programs are running in parallel, event though only one program is executed at a given instance of time.
3. Allows us to execute multiple programs simultaneously.
4. Can improve performance, if program 1 stalls execute program 2

# Sequential Vs. Concurrent Vs. Parallel

| Program 1 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |

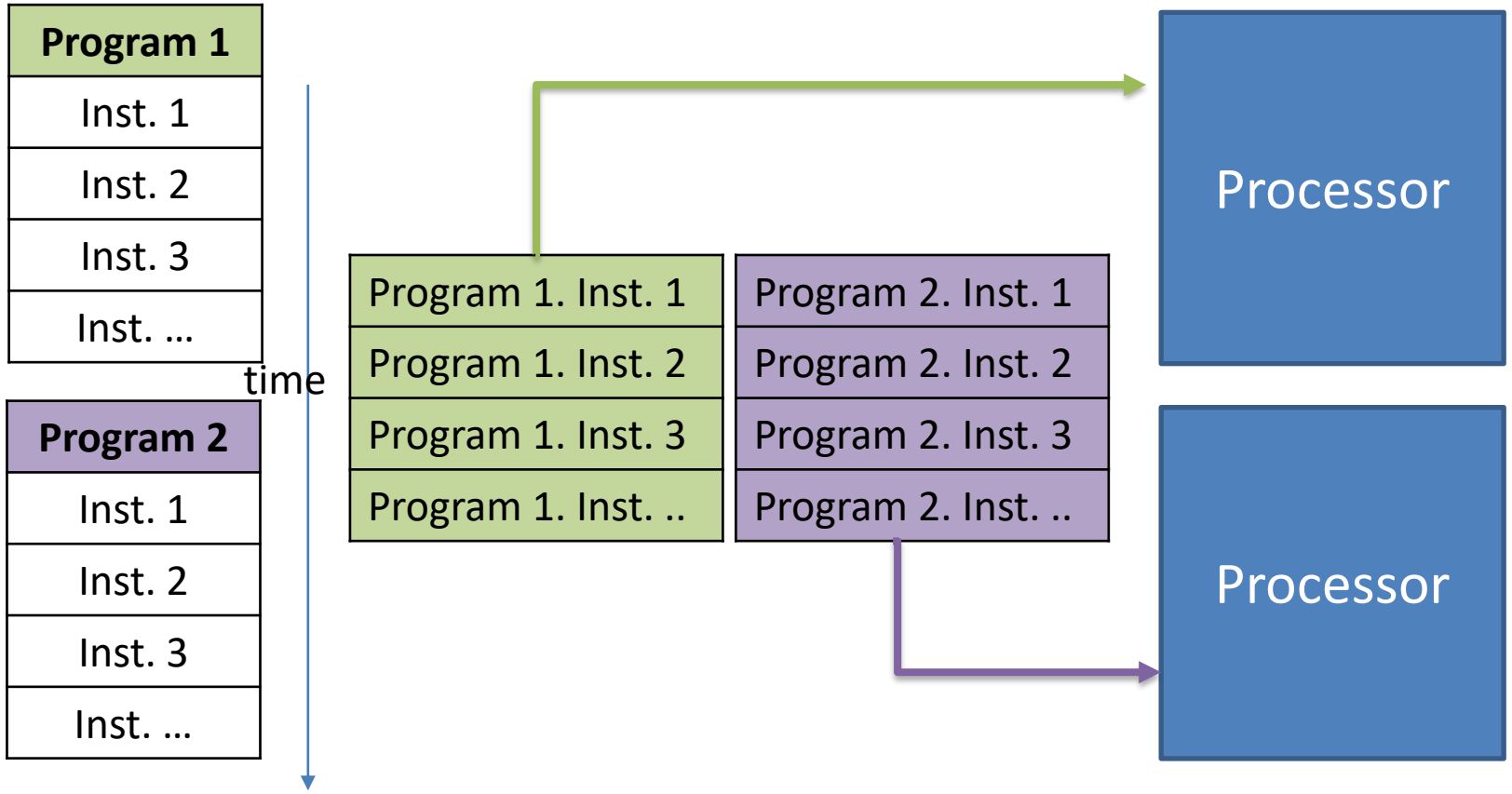
| Program 2 |
| --- |
| Inst. 1 |
| Inst. 2 |
| Inst. 3 |
| Inst. … |




1. Multiple processors or mulit-core processors are available.

**Parallel**

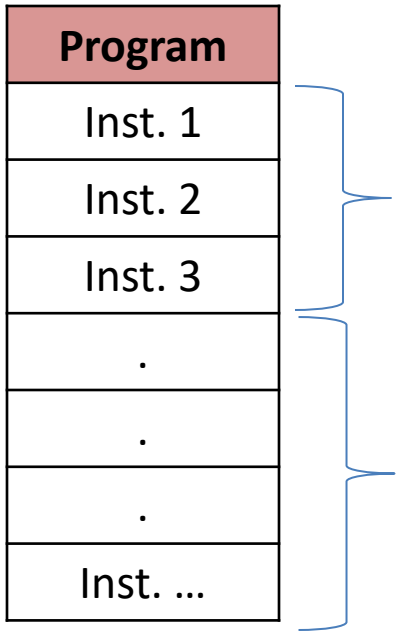
# Sequential Vs. Concurrent Vs. Parallel

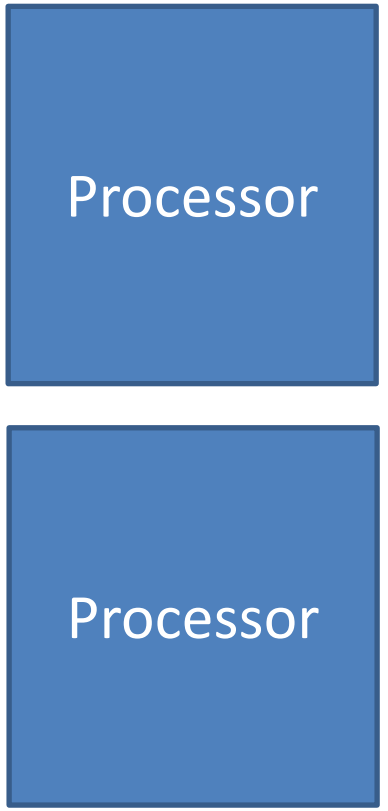



**Parallel**

1. Multiple processors or mulit-core processors are available.
2. Program 1 is executed by processor 1 and program 2 is executed by processor 2.
3. Two tasks executed in parallel, referred to as **task-level parallelism**.

# Sequential Vs. Concurrent Vs. Parallel

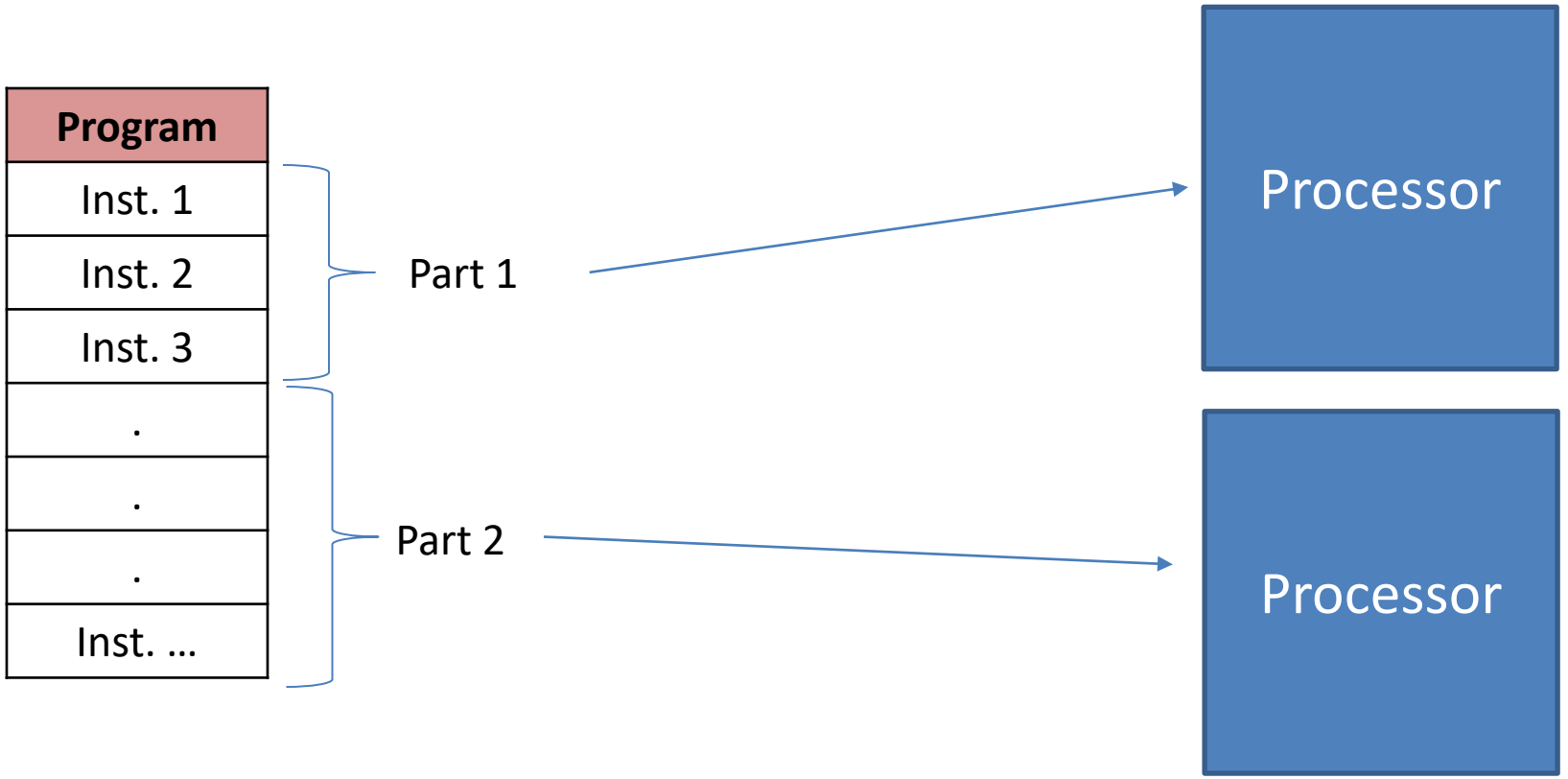



1. Multiple processors or mulit-core processors are available.
2. Program 1 is executed by processor 1 and program 2 is executed by processor 2.
3. Two tasks executed in parallel, referred to as **task-level parallelism**.
4. Segment program into multiple chunks

**Parallel**

# Sequential Vs. Concurrent Vs. Parallel




**Parallel**

1. Multiple processors or mulit-core processors are available.
2. Program 1 is executed by processor 1 and program 2 is executed by processor 2.
3. Two tasks executed in parallel, referred to as **task-level parallelism**.
4. Segment program into multiple chunks, and execute each chunk on different processor
5. One task executed on multiple processors simultaneously, referred to as a **parallel processing program**

# Realizing Concurrency and Parallelism

- Concurrency:

# Realizing Concurrency and Parallelism

- Concurrency:
  - OS can perform scheduling
  - Hardware Multi-threading
- Parallel execution
  - Task level parallelism

# Realizing Concurrency and Parallelism

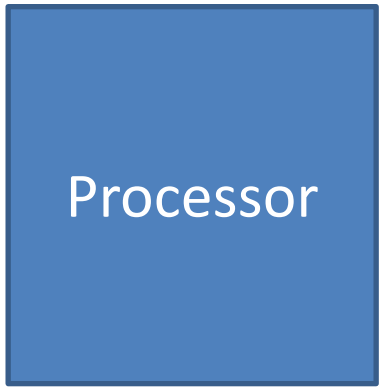
- Concurrency:
  - OS can perform scheduling
  - Hardware Multi-threading
- Parallel execution
  - Task level parallelism
    - Task scheduling (assign task to individual processes)
  - Parallel process programming

# Example

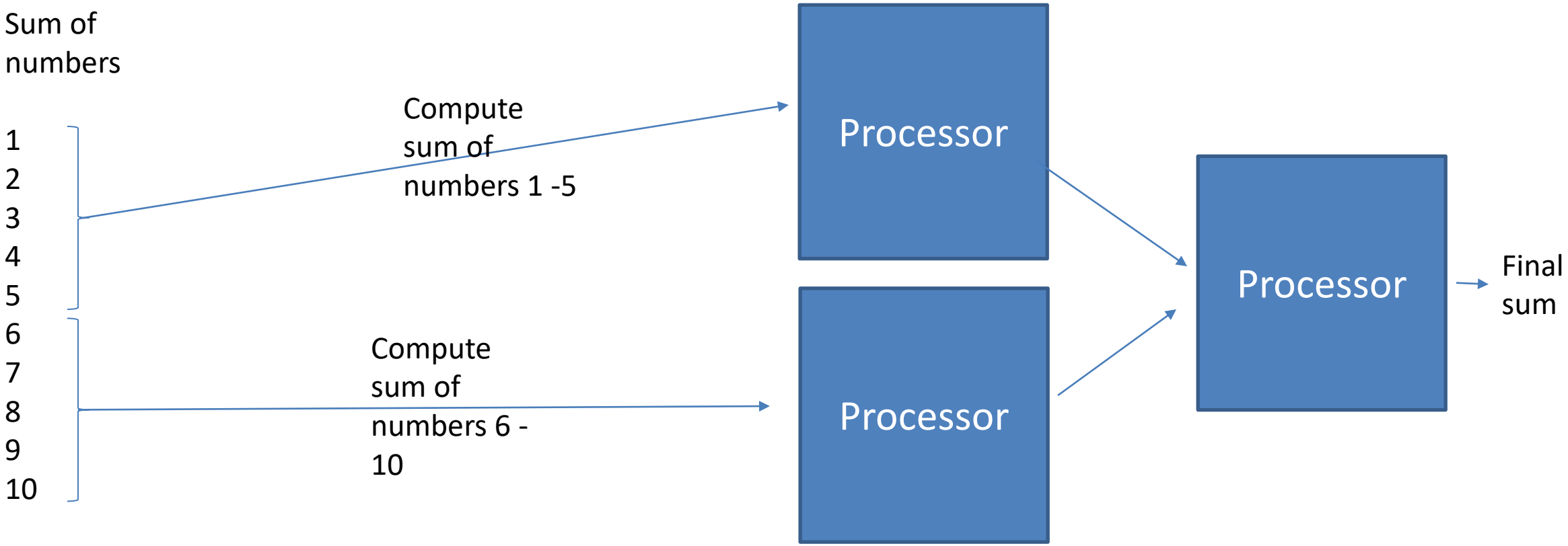
Sum of numbers

1
2
3
4
5
6
7
8
9
10

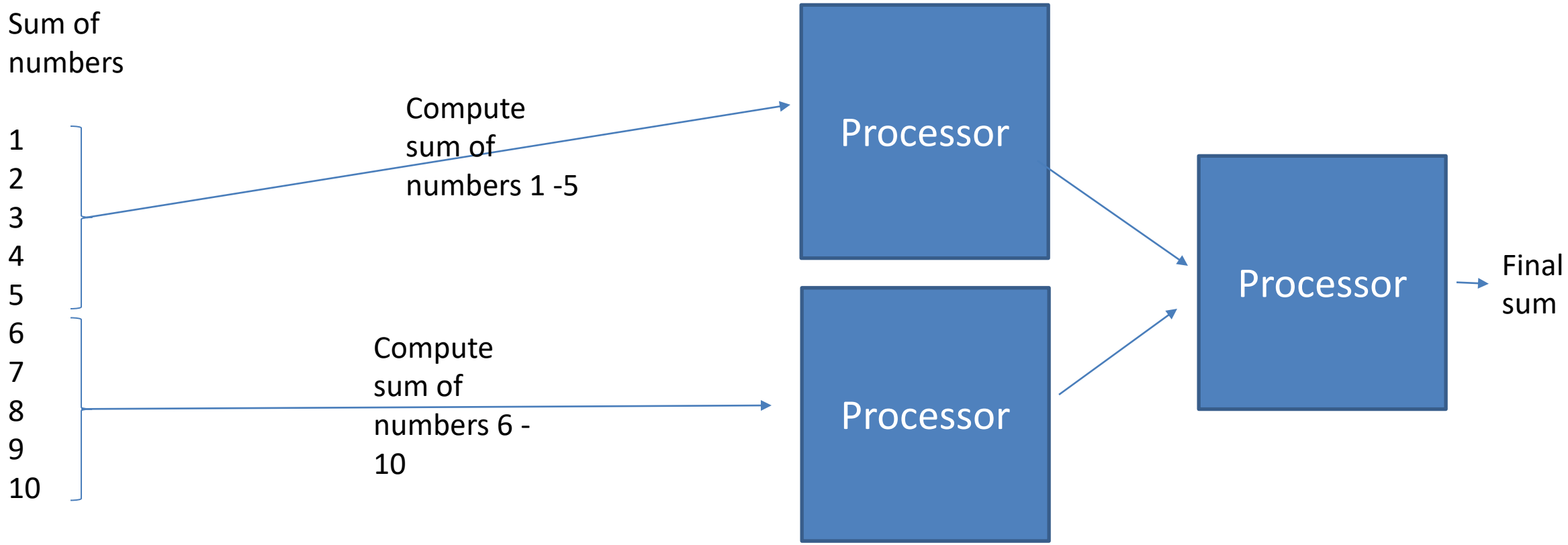
```
int sum = 0
for (i = 0, i <= 10, i++)
{
        sum += a[i]
}
```



# Example


Sum of numbers
1
2
3
4
5
6
7
8
9
10
Compute sum of numbers 1 -5
Compute sum of numbers 6 - 10
Processor
Processor
Processor
Final sum

# Example




```
int sum = 0
for (i = 0, i <= 10, i++)
{
        sum += a[i]
}
```

**Convert to parallel code, may using conditional statements**

# Realizing Concurrency and Parallelism

- Concurrency:
  - OS can perform scheduling
  - Hardware Multi-threading
- Parallel execution
  - Task level parallelism
    - Task scheduling (assign task to individual processes)
  - Parallel process programming
    - Needs to be implemented by programmer
    - Challenge to create software and hardware that will make it easy to write correct parallel processing programs

# Difficulty of Creating Parallel Programs

- Difficult to write program that can be executed on multiple processors
  - Complexity increases as the number of processors increase
- Parallel programs should produce better performance or better energy efficiency
- Difficulty in writing parallel programs
  - Breaking a task into independent segments
  - Communicating between segments
  - Balance load approximately across processors

# Speed Up challenge

- Will using n processors produce a speed up of n?

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

$$t_1 = \frac{parallel\ program}{100} + nonparallel\ program$$

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

$$t_1 = \frac{(x)}{100} + (1 - x)$$

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

$$t_1 = \frac{(x)}{100} + (1 - x)$$

$$speed\ up(90) = \frac{t}{t_1} = \frac{1}{\frac{(x)}{100} + (1 - x)}$$

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

$$90 = \frac{1}{\frac{(x)}{100} + (1 - x)}$$

$$x = 0.999$$

# Speed Up challenge

- Will using n processors produce a speed up of n?
- Program A executes in time $t$ on 1 processor
- We use $100$ processors and want a speed up of $90$ times.
- Assume $x\%$ of the program can be parallelised
- Execution time of parallel program

$$t_1 = \frac{(1-x)}{100} + x$$

$$x = 0.001$$

Speed up of 90 times is possible if 99.9% of code can be parallized.

- Speed up depends on problem size
  - More speed up for larger problems
- Assuming that even load balancing is possible across all the processors

# Classification of Parallel Architectures

Flynn's Taxonomy

- A classification of computer architectures
- Based upon the number of concurrent instruction (or control) streams and data streams available in the architecture.

# Classification of Parallel Architectures

Flynn's Taxonomy

- A classification of computer architectures
- Based upon the number of concurrent instruction (or control) streams and data streams available in the architecture.

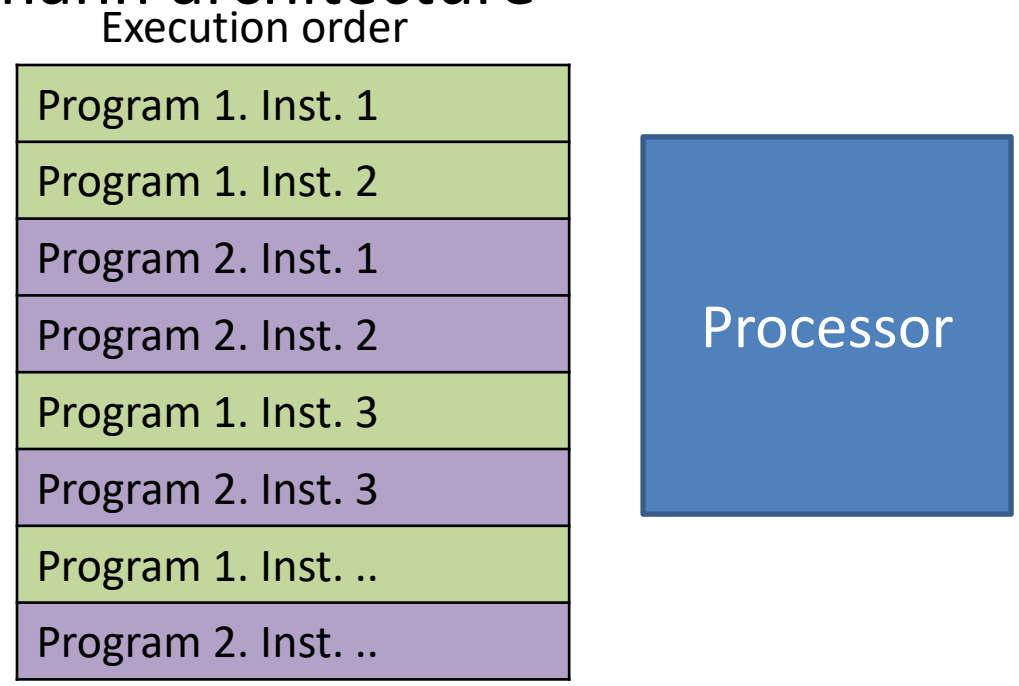
(instructions)

(processors)

# Classification of Parallel Architectures

Flynn's Taxonomy

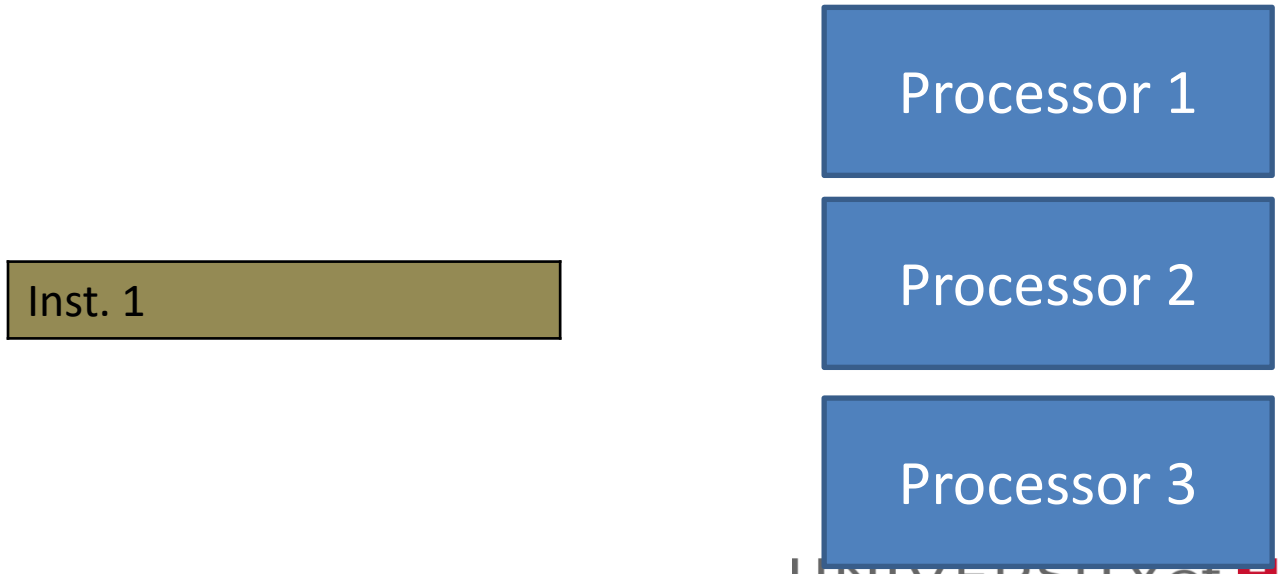
- SISD: Single instruction single data
  - Classical von Neumann architecture



# Classification of Parallel Architectures

Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data




1. All the processors are working to execute instruction 1.
2. Need to work synchronously.

# Vector Architecture

- Support vector operations
- Components
  - Vector registers: Hold 64 elements (vector)
  - Load and Store Vectors
  - Perform actions on Vectors

# Vector Architecture

- Support vector operations
- Example

$$V1 = [a_0, a_1, .., a_{63}]$$
$$V2 = [b_0, b_1, .., b_{63}]$$

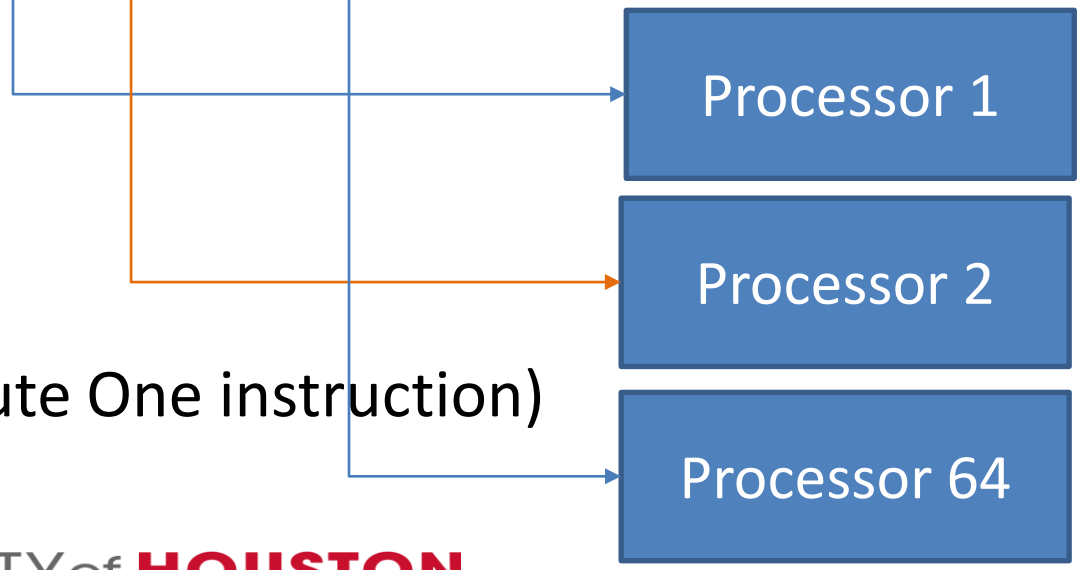
Vector LEGv8 code:

*FADDDV V3, V1, V2*

# Vector Architecture

- Support vector operations
- Example

$$V1 = [a_0, a_1, .., a_{63}]$$
$$V2 = [b_0, b_1, .., b_{63}]$$




Vector LEGv8 code:

*FADDDV V3, V1, V2*

(Multiple processors working to execute One instruction)

# Classification of Parallel Architectures

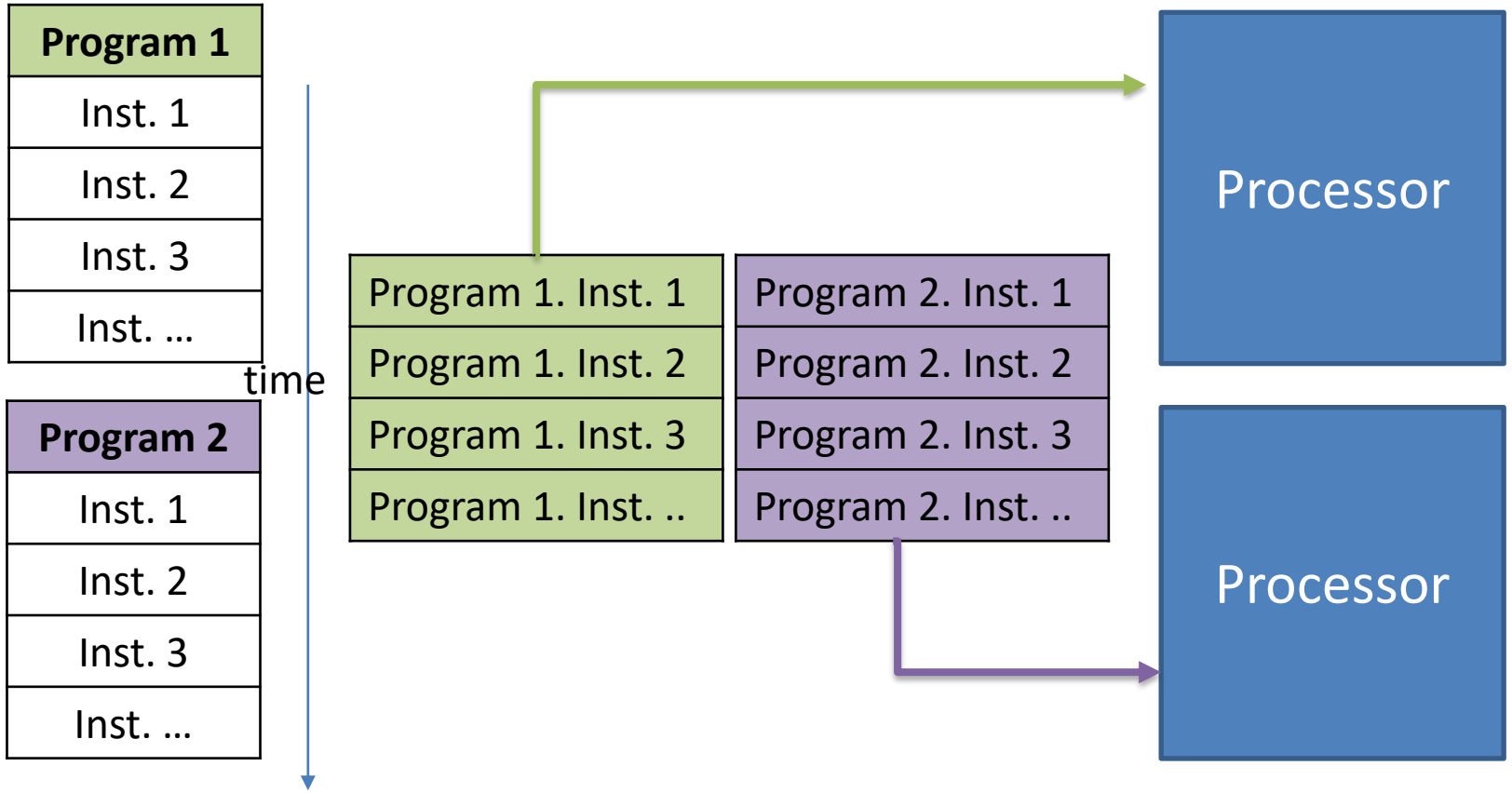
Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness

# Classification of Parallel Architectures

Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine




1. Multiple processors or mulit-core processors are available.
2. Multiple processors executing various instructions asynchronously.

# Parallelism in Single Processor Architectures

# Parallelism in Single Processor Architectures

- Pipelining
  - Overlap the execution of multiple instructions
  - N stages ➔ N instructions executed

# Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle

# Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1

# Multiple Issue

- Static multiple issue
  - Compiler groups **independent instructions** to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

# Limitations of ILP

- Problem: within a single instruction stream we do not find enough independent instructions to execute simultaneously due to
  - data dependencies
  - difficulties to detect memory dependencies among instruction
- Consequence: significant number of functional units are idling at any given time
- Question: Can we maybe execute instructions from another instructions stream concurrently
  - Another thread?
  - Another process?

# Hardware Multithreading

- Performing multiple threads of execution in parallel

# Hardware Multithreading

- Performing multiple threads of execution in parallel
- Problems for executing instructions from multiple threads at the same time
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter

# Hardware Multithreading

- Performing multiple threads of execution in parallel
- Problems for executing instructions from multiple threads at the same time
  - The instructions in each thread might use the same register names
  - Each thread has its own program counter
- When to switch between different threads?

# Fine-grain multithreading

- Switch threads after each cycle
- Interleave instruction execution
- If one thread stalls, others are executed

Execution order

| |
|---|
| Program 1. Inst. 1 |
| Program 2. Inst. 1 |
| Program 1. Inst. 2 |
| Program 2. Inst. 2 |
| Program 1. Inst. 3 |
| Program 2. Inst. 3 |
| Program 1. Inst. .. |
| Program 2. Inst. .. |

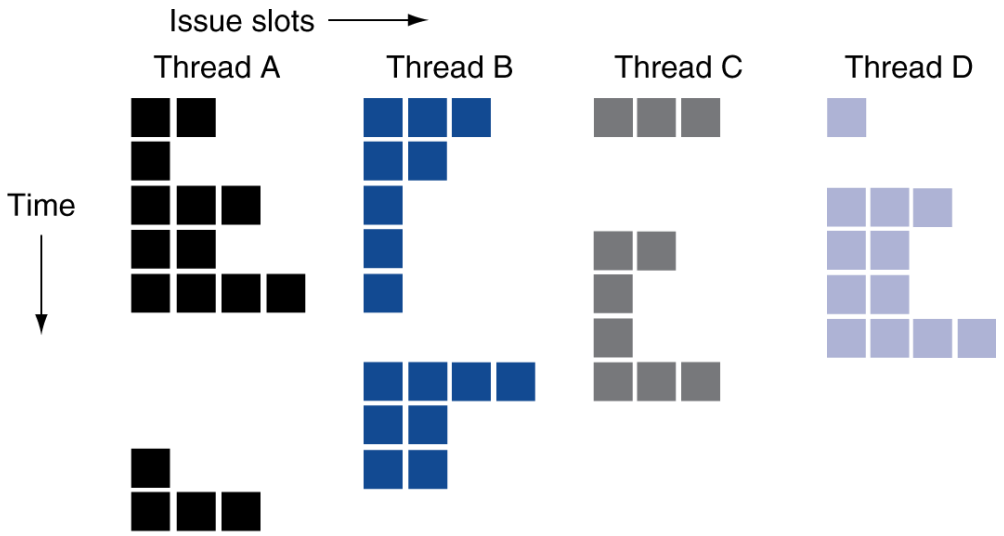
# Coarse-grain multithreading

- Only switch on long stall (e.g., L2-cache miss)
- Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

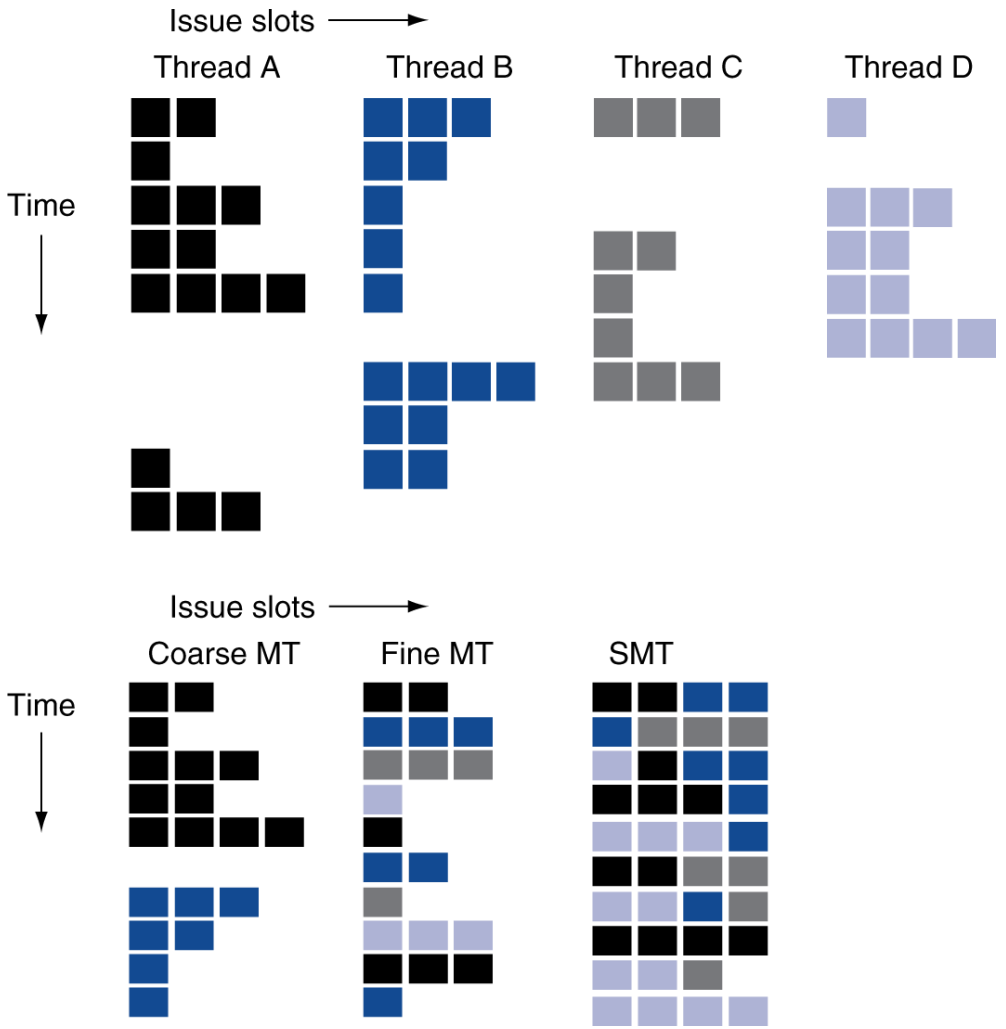
# Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

# Multithreading Example



# Multithreading Example

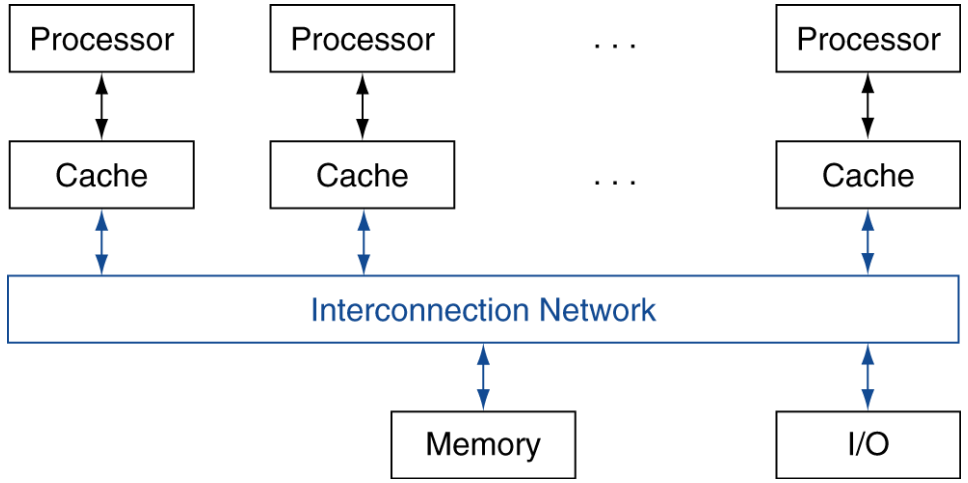


# Parallelism in Multi Processor/Core Architectures

- Difficult to write parallel programs
- Simplify
  - Single physical address space, shared by all processors
  - Programs do not have worry about where the data is stored
  - All variable available to all processors
- Shared Memory Multi-Processors

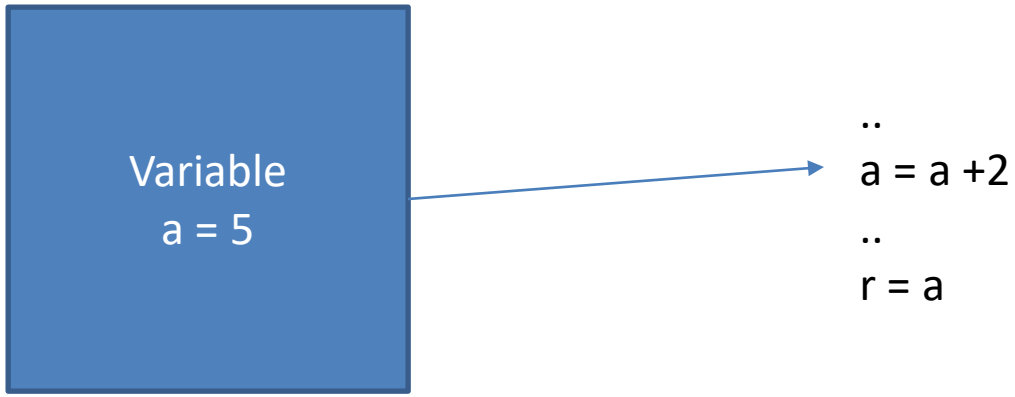
# Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors



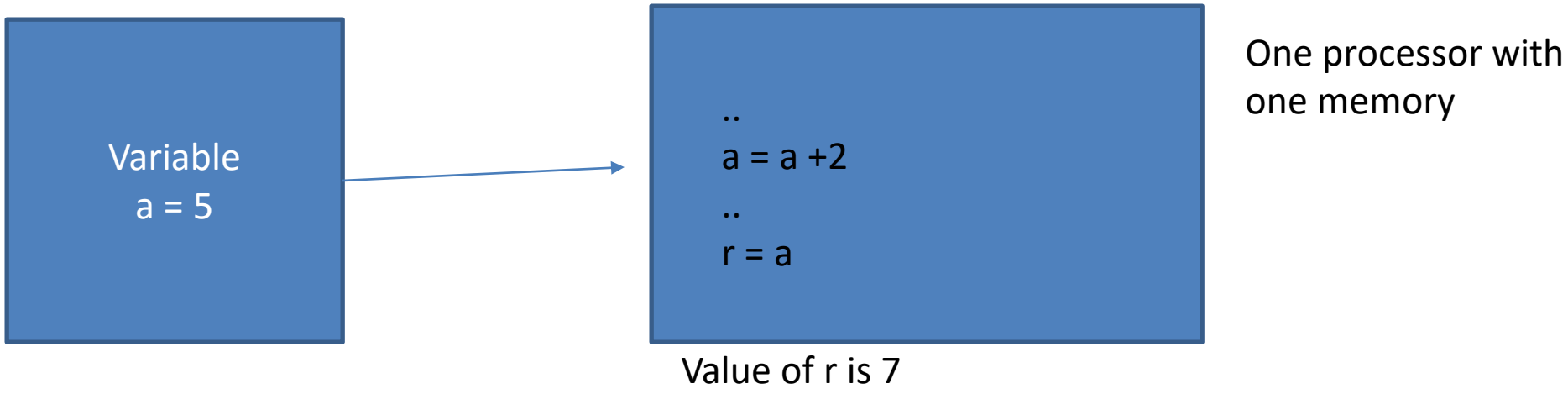
# Synchronization

- Two processors sharing an area of memory



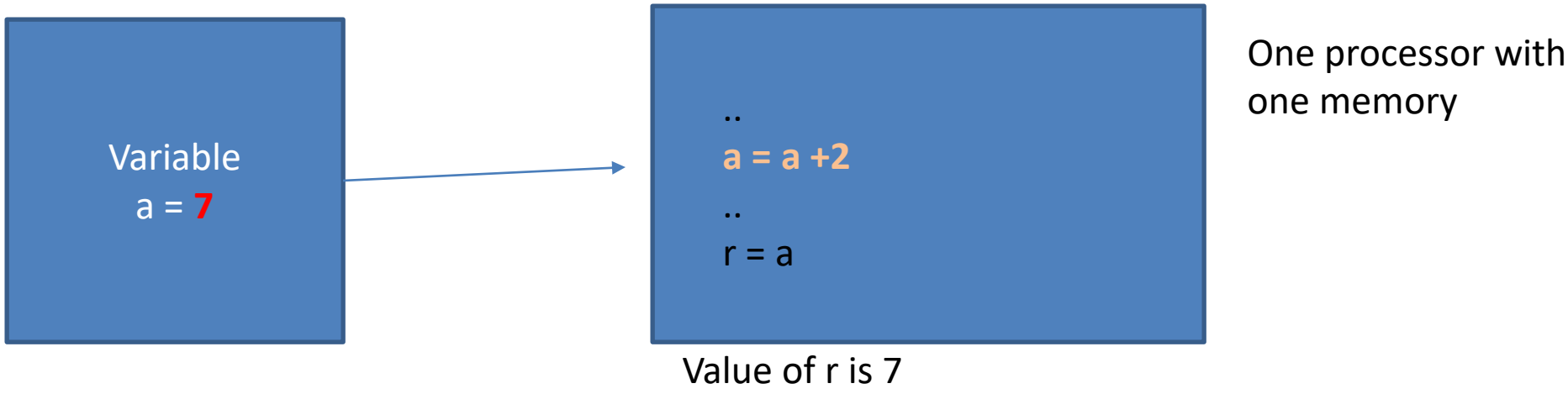
# Synchronization

- Two processors sharing an area of memory



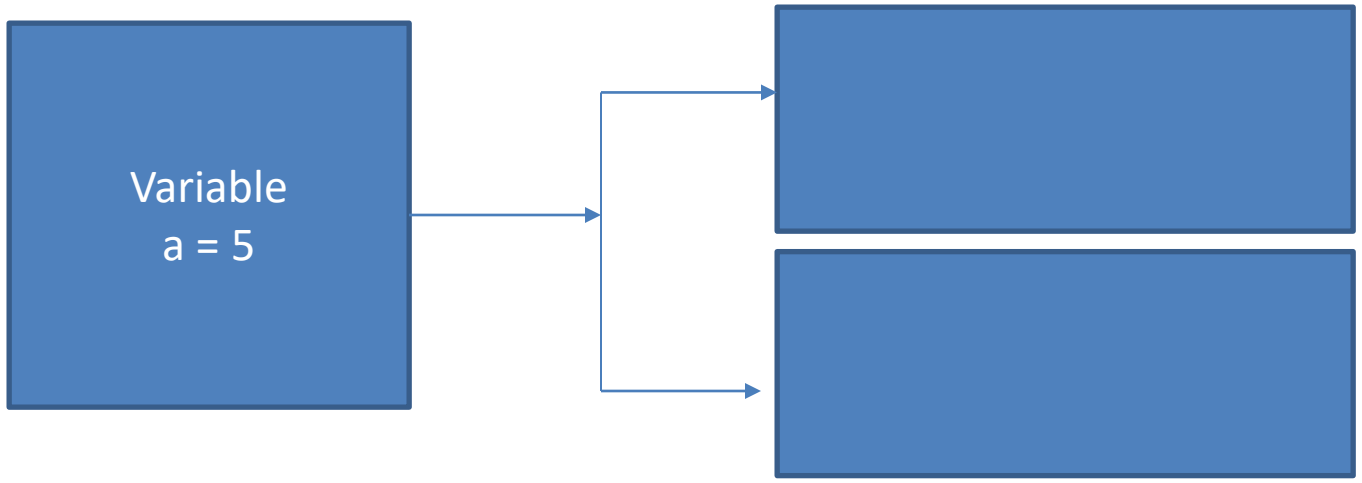
# Synchronization

- Two processors sharing an area of memory



# Synchronization

- Two processors sharing an area of memory

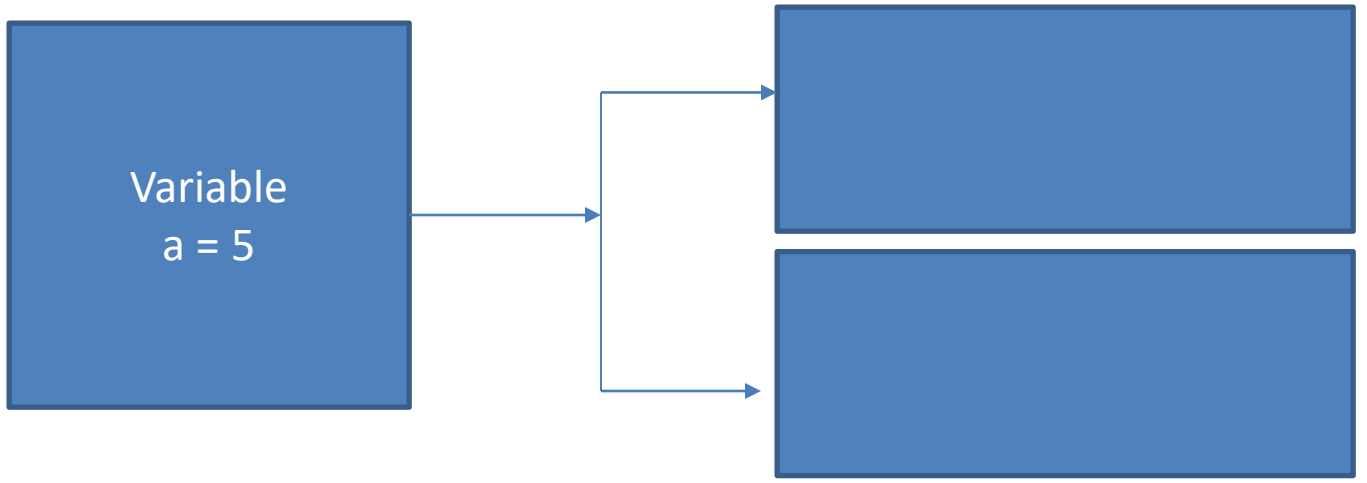



Two processor with shared memory

# Synchronization

- Two processors sharing an area of memory

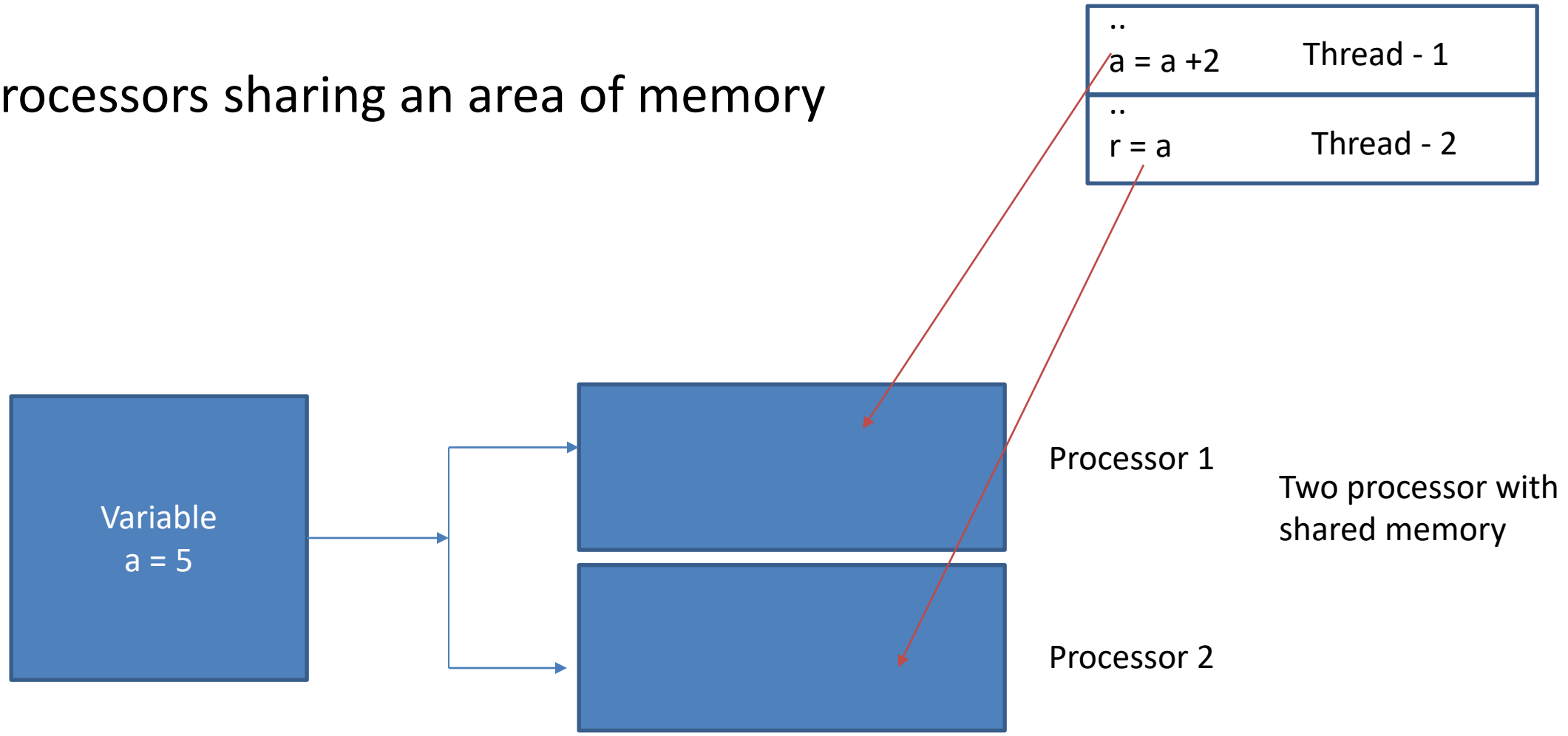
| | |
|---|---|
| ..<br>a = a +2 | Thread - 1 |
| ..<br>r = a | Thread - 2 |




Two processor with shared memory

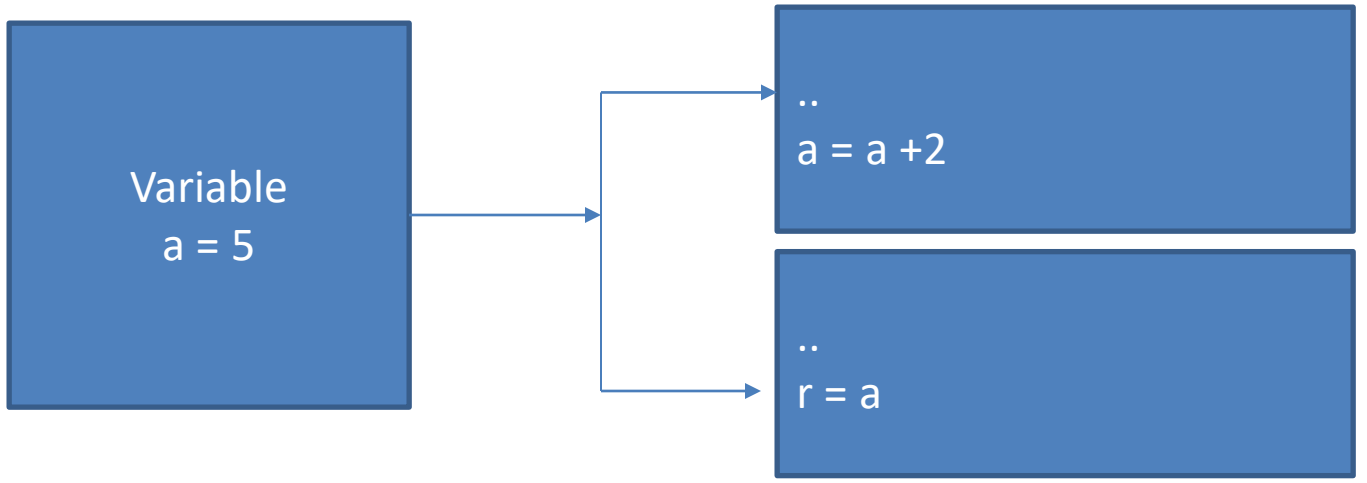
# Synchronization

- Two processors sharing an area of memory



# Synchronization

- Two processors sharing an area of memory




Processor 1

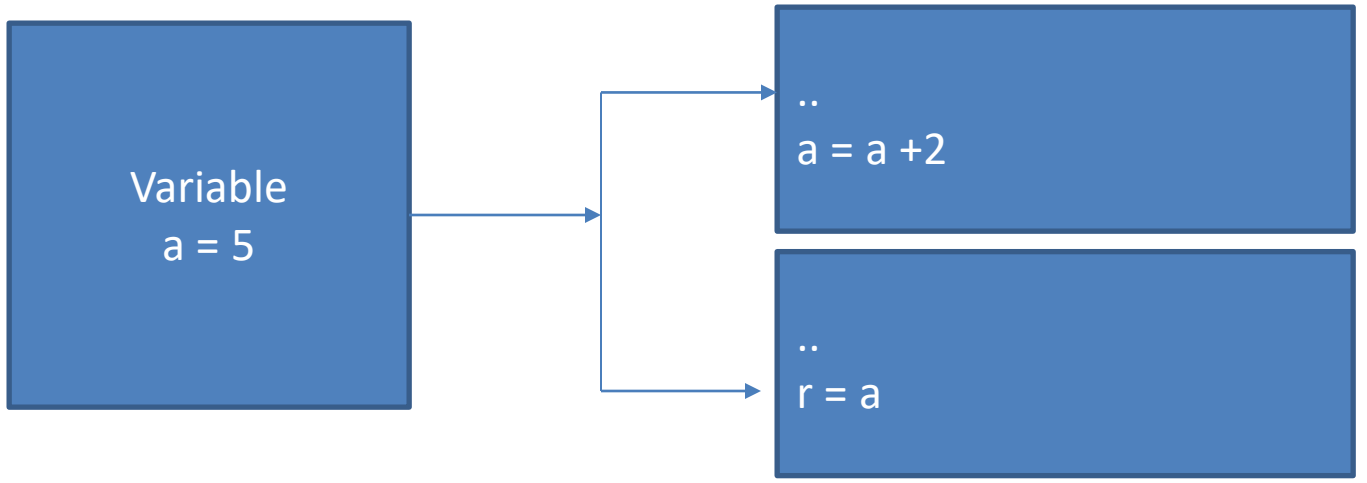
Processor 2

Two processor with shared memory

If a is read while the previous instruction is executed

# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses

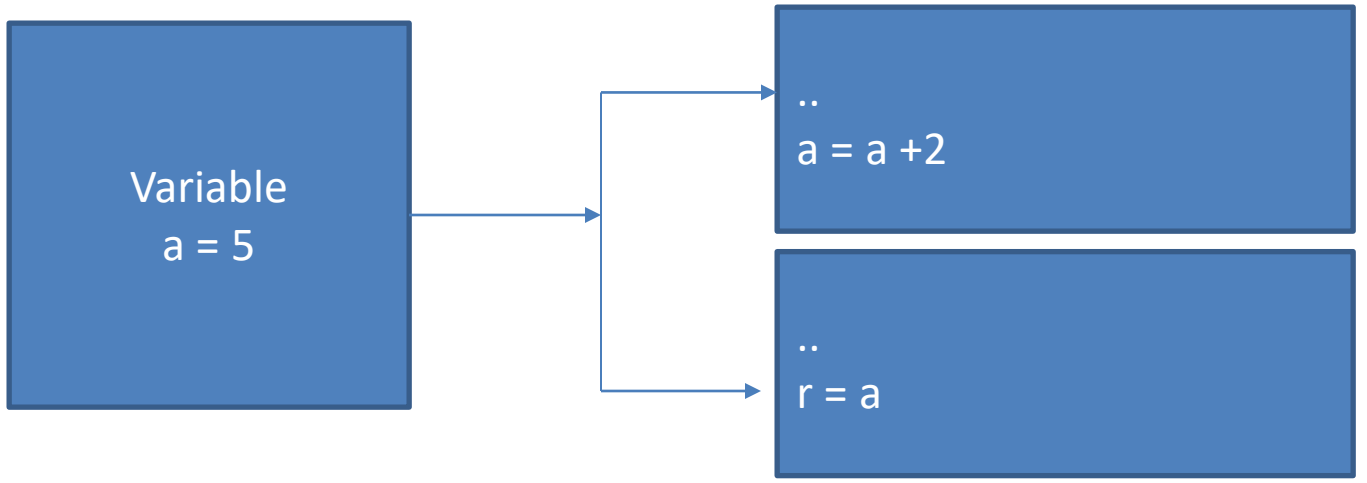



Two processor with shared memory

Value of r = 5 (incorrect value)

# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses

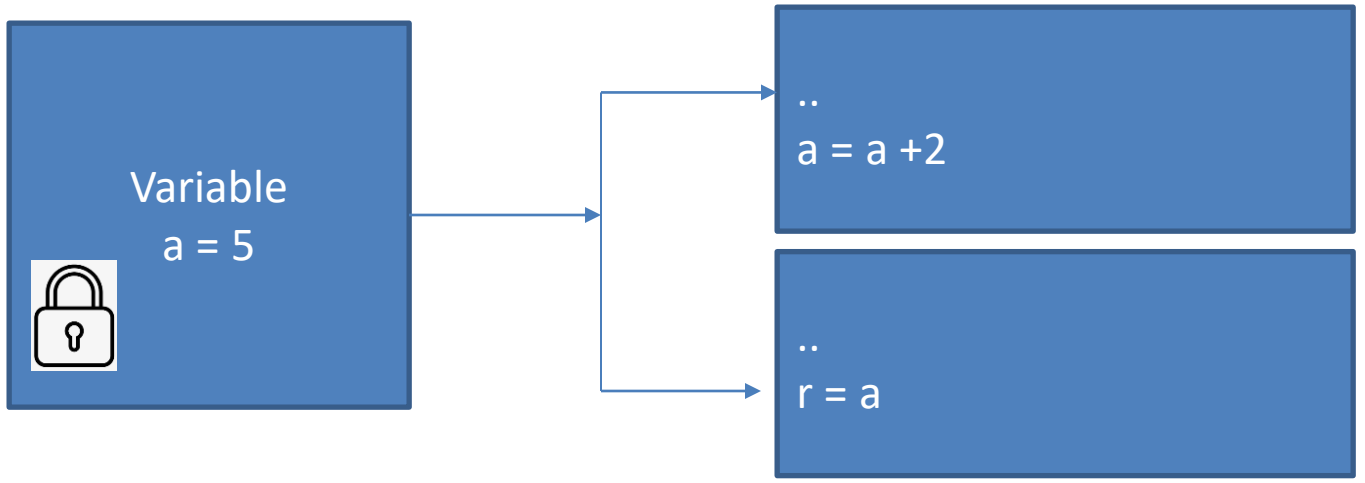



Two processor with shared memory

When processor 1 accesses variable a put a lock on it

# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses

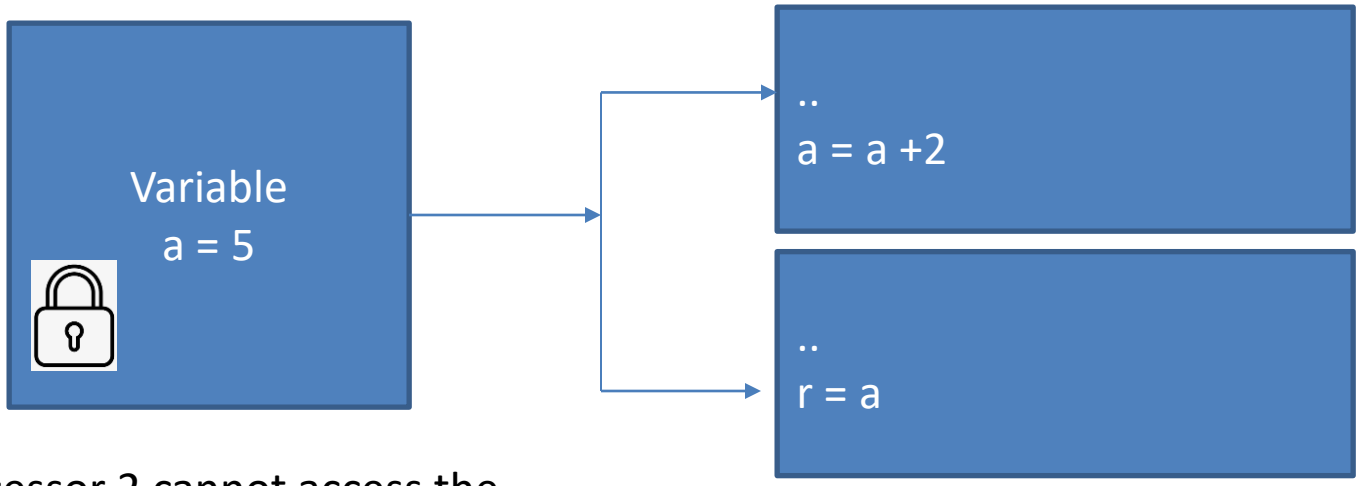



Two processor with shared memory

When processor 1 accesses variable a put a lock on it

# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on order of accesses




Two processor with shared memory

Processor 2 cannot access the variable as long as it is locked

# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

# Multi-Core processors

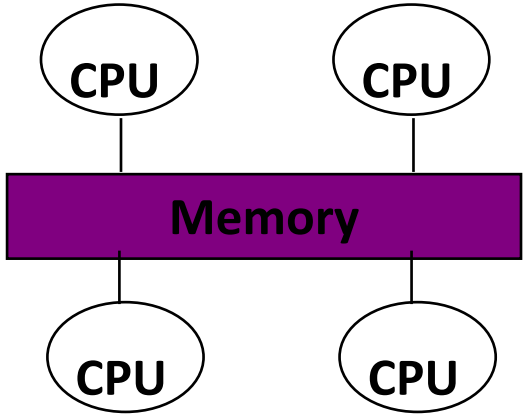
- Compute cores on a **multi-core** processor share the same main memory -> Shared memory architecture
- Difference to previous multi-processor systems:
  - compute cores are on the same chip
  - Multi-core processors typically **connected over a cache**, while previous SMP systems were typically connected over the main memory

# Shared memory systems (I)

- All processes have access to the same address space
  - E.g. PC with more than one processor
- Data exchange between processes by writing/reading shared variables
  - Shared memory systems are (relatively) easy to program
  - Current standard in scientific programming: OpenMP
- Two versions of shared memory systems available today
  - Centralized Shared Memory Architectures
  - Distributed Shared Memory architectures

# Centralized Shared Memory Architecture

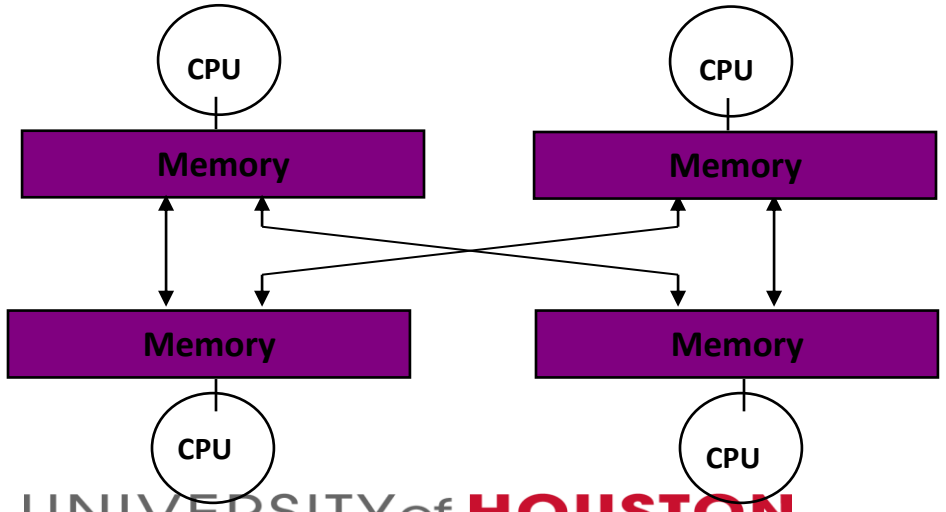
- Also referred to as Symmetric Multi-Processors (SMP)
- All processors share the same physical main memory




- Memory bandwidth per processor is limiting factor for this type of architecture
- Typical size: 2-32 processors

# Distributed Shared Memory Architectures

- Also referred to as Non-Uniform Memory Architectures (NUMA)
  - Some memory is closer to a certain processor than other memory
  - The entire memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly

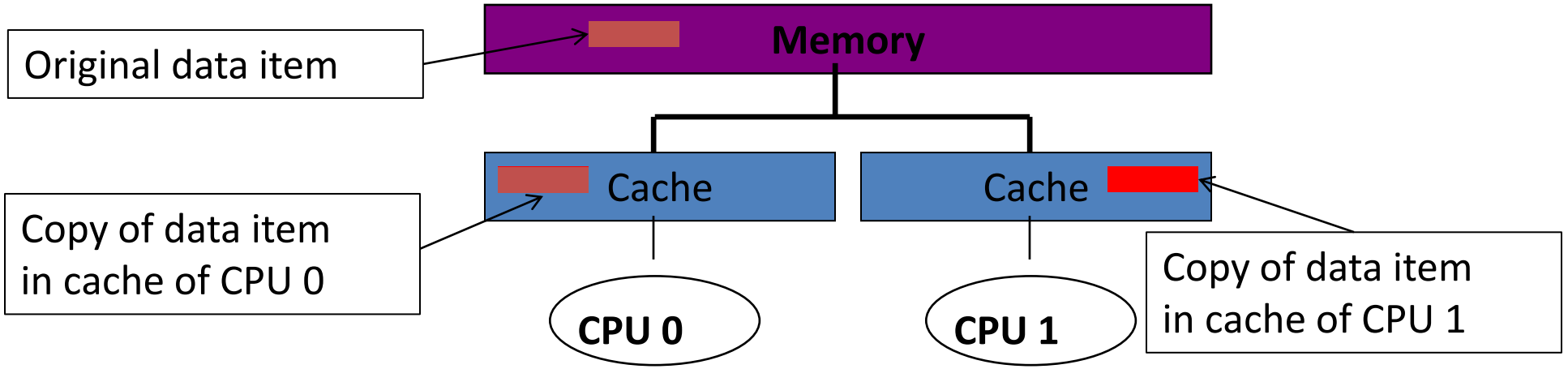


# NUMA architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - First touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used

# Cache Coherence

- Real-world shared memory systems have caches between memory and CPU
- Copies of a single data item can exist in multiple caches
- Modification of a shared data item by one CPU leads to outdated copies in the cache of another CPU



# Cache coherence (II)

- Typical solution:
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, ‘notification’ of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- Cache coherence only an issue in case multiple tasks access the same item
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another