Tikrit University
Electrical Engineering Department

EE-317
Computer Engineering
2024-2025

Parallel Processors

Jalal Nazar Abdulbaqi, Ph.D.

jalal.abdulbaqi@tu.edu.iq

Outline

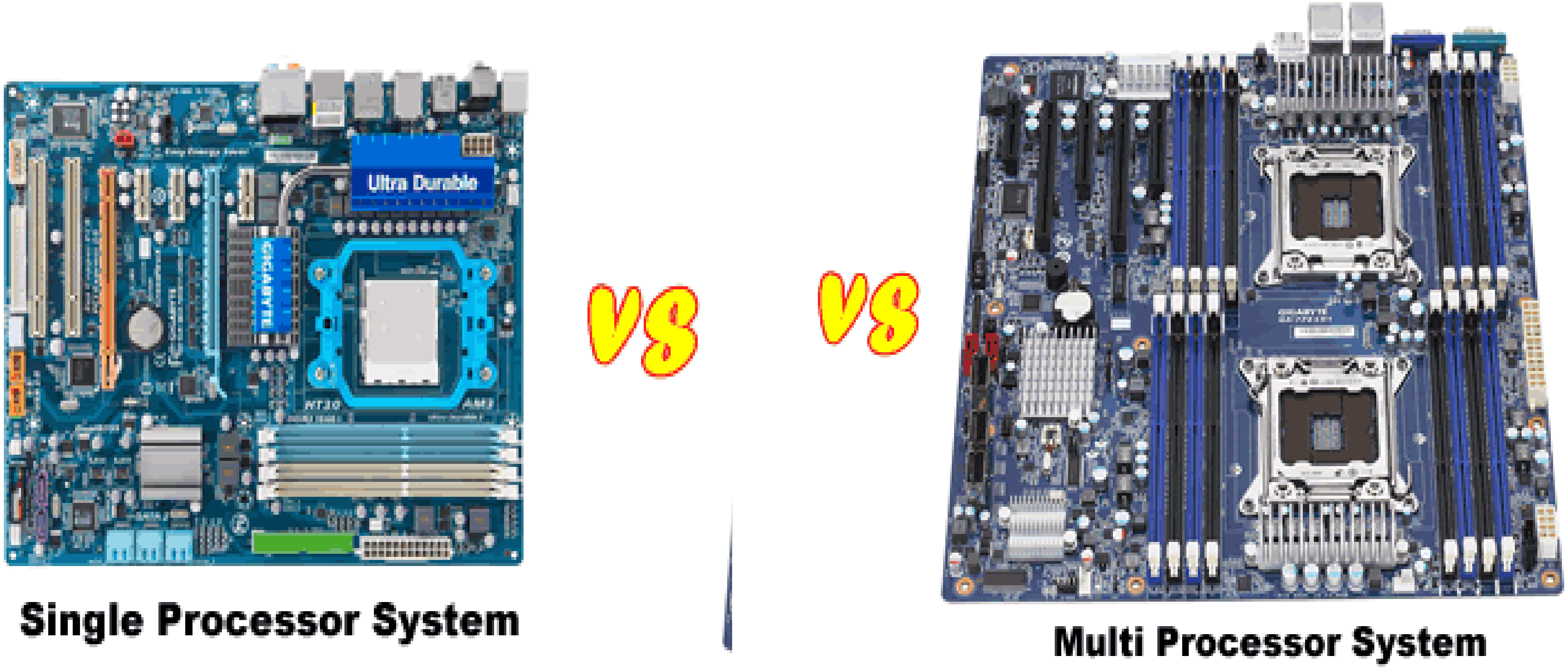
- Introduction
- The Difficulty of Creating **Parallel Processing Programs**
- SISD, MIMD, SIMD, SPMD, and Vector
- Hardware **Multithreading**
- **Multicore** and Other Shared Memory Multiprocessors

Introduction

- Goal: connecting multiple computers to get **higher performance**
  - Multiprocessors
  - Scalability, availability, **power efficiency**
- **Cluster**
  - Set of computers linked via LAN as a single large multiprocessor.
- **Multicore** microprocessors
  - Chips with multiple processors (cores)
- **Shared Memory Processor(SMP)**
  - Parallel processor with single physical address space
- **Parallel processing program**
  - Single program run on multiple processors
- **Task-level (process-level) parallelism**
  - High throughput for independent jobs

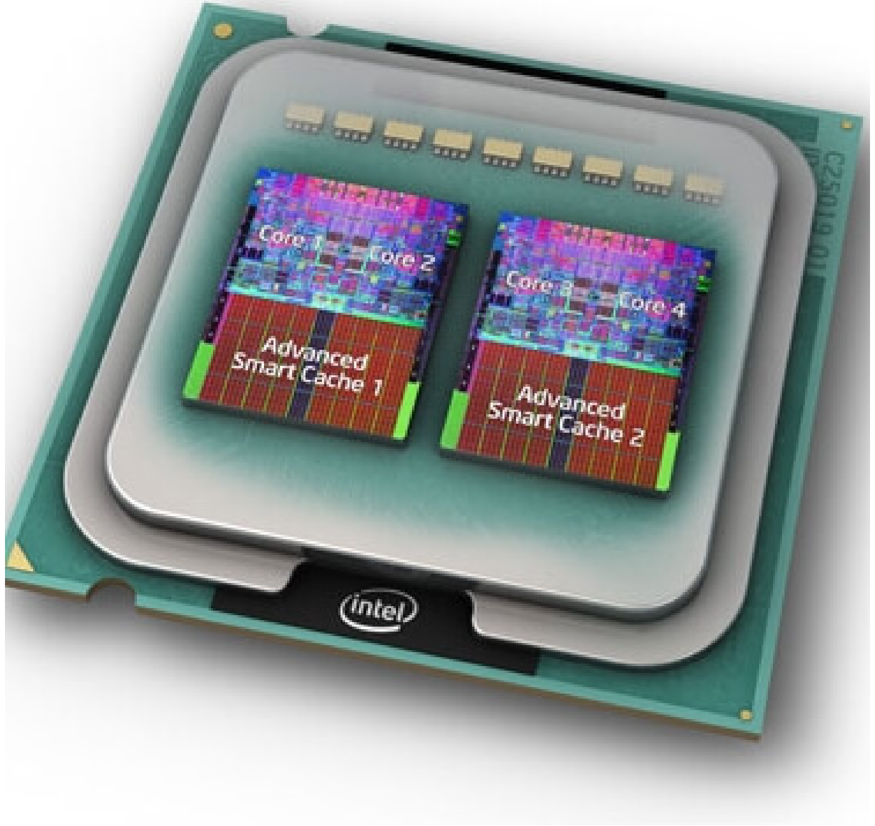
Multiprocssor

- A computer system with at least two processors.



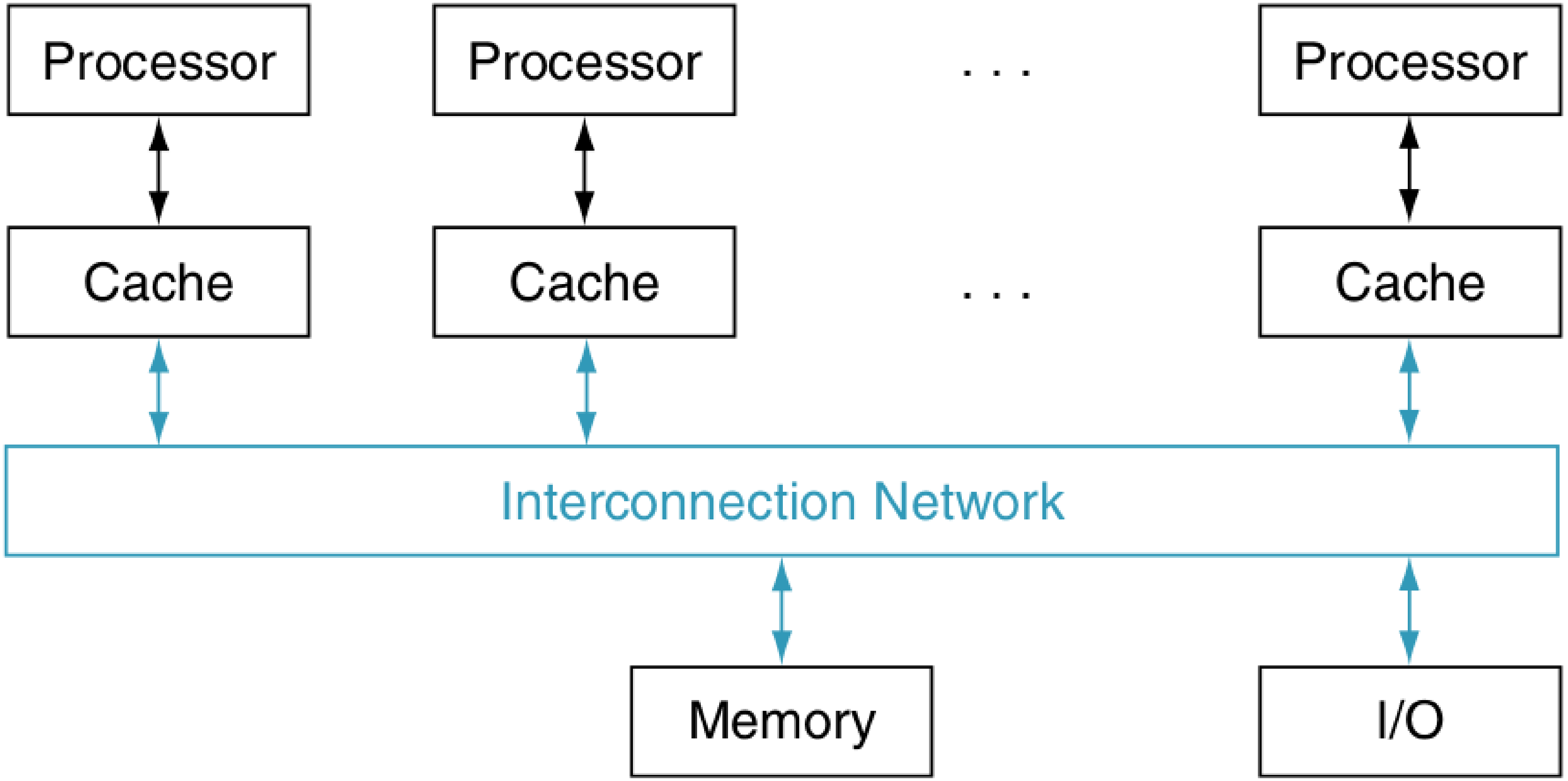
Multicore Microprocessor

- A microprocessor containing multiple processors ("cores") in a single integrated circuit (Chip).



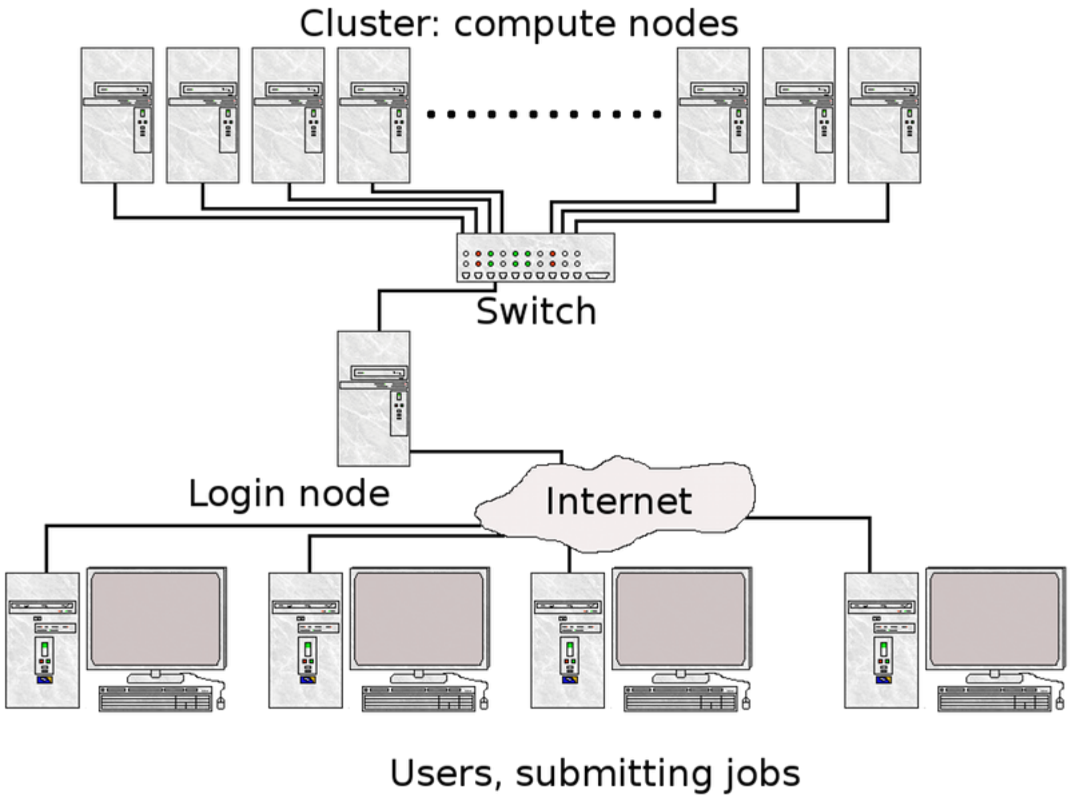
Shared Memory Processors (SMPs)

- A parallel processor with a single physical address space.



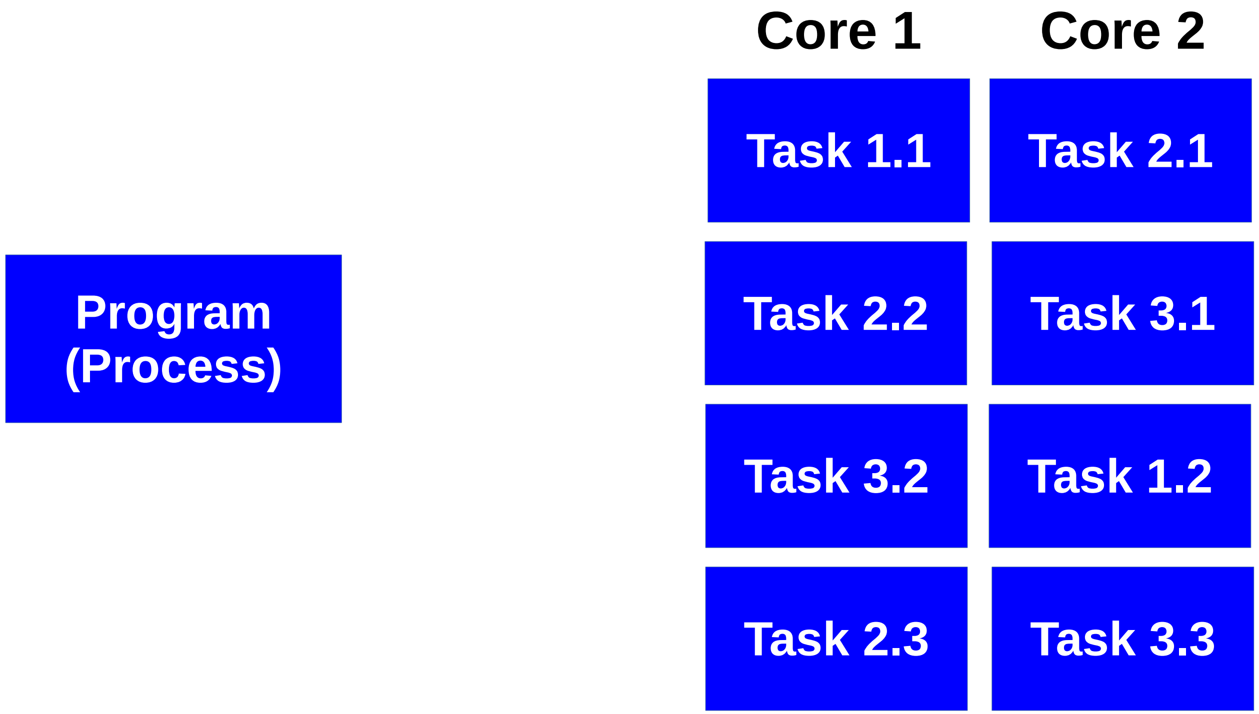
Cluster

- A set of computers connected over a local area network that function as a single large multiprocessor.



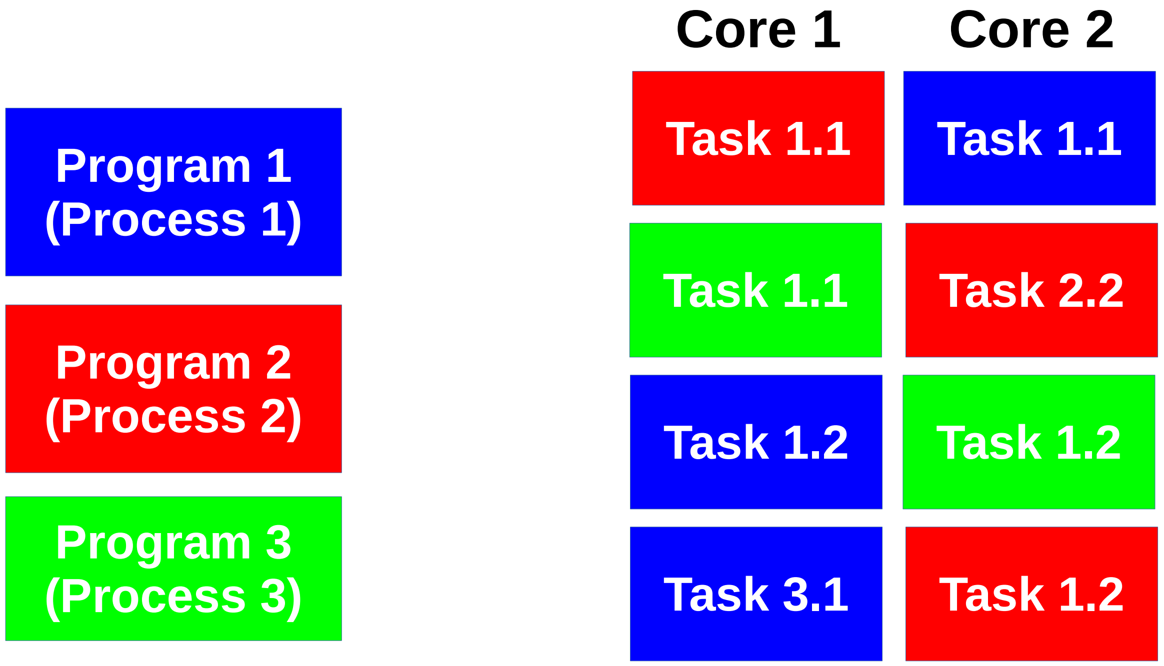
Parallel Processing Program

A single program that runs on multiple processors simultaneously.



Task-level (Process-level) Parallelism

- Utilizing multiple processors by running independent programs simultaneously.



Hardware and Software

		Software: Sequential	Software: Concurrent
Hardware	Serial	**Matrix Multiply** written in **Matlab** running on an **Intel Pentium 4**	Windows Vista OS running on **Intel Pentium 4**
	Parallel	**Matrix Multiply** written in **Matlab** running on an **Intel Core i7**	Windows Vista OS running on **Intel Core i7**

- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

Parallel Programming

- **Parallel software** is the problem
- Need to get significant **performance** or **energy efficiency** improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning (split tasks equally)
  - Coordination (Balancing the load evenly + synchronizing)
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
- $T_{new} = T_{parallelizable}/100 + T_{sequential}$

$$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$

- Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 40 processors
  - Assume only the matrix sum is parallelizable
  - Assumes load can be balanced across processors
- **Single processor**: Time = $(10 + 100) \times t_{add} = 110\ t_{add}$
- **10 processors**
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (5.5 / 10 = **55%** of potential)
- **40 processors**
  - Time = $10 \times t_{add} + 100/40 \times t_{add} = 12.5 \times t_{add}$
  - Speedup = 110/12.5 = 8.8 (8.8 / 40 = **22%** of potential) - ***Strong scaling***

Scaling Example (cont)

- What if matrix size is 20×20?
- Single processor: Time = $(10 + 400) \times t_{add} = 410 \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 400/10 \times t_{add} = 50 \times t_{add}$
  - Speedup = 410/50 = 8.2 (8.2/10 = **82%** of potential)
- 40 processors
  - Time = $10 \times t_{add} + 400/40 \times t_{add} = 20 \times t_{add}$
  - Speedup = 410/20 = 20.5 (20.5 / 40 = **51%** of potential) - ***Weak scaling***

Strong vs Weak Scaling

- **Strong scaling**: measuring speed-up while keeping the problem size fixed
- **Weak scaling**: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1024/100 \times t_{add} = 20.24 \times t_{add}$
  - Constant performance in this example

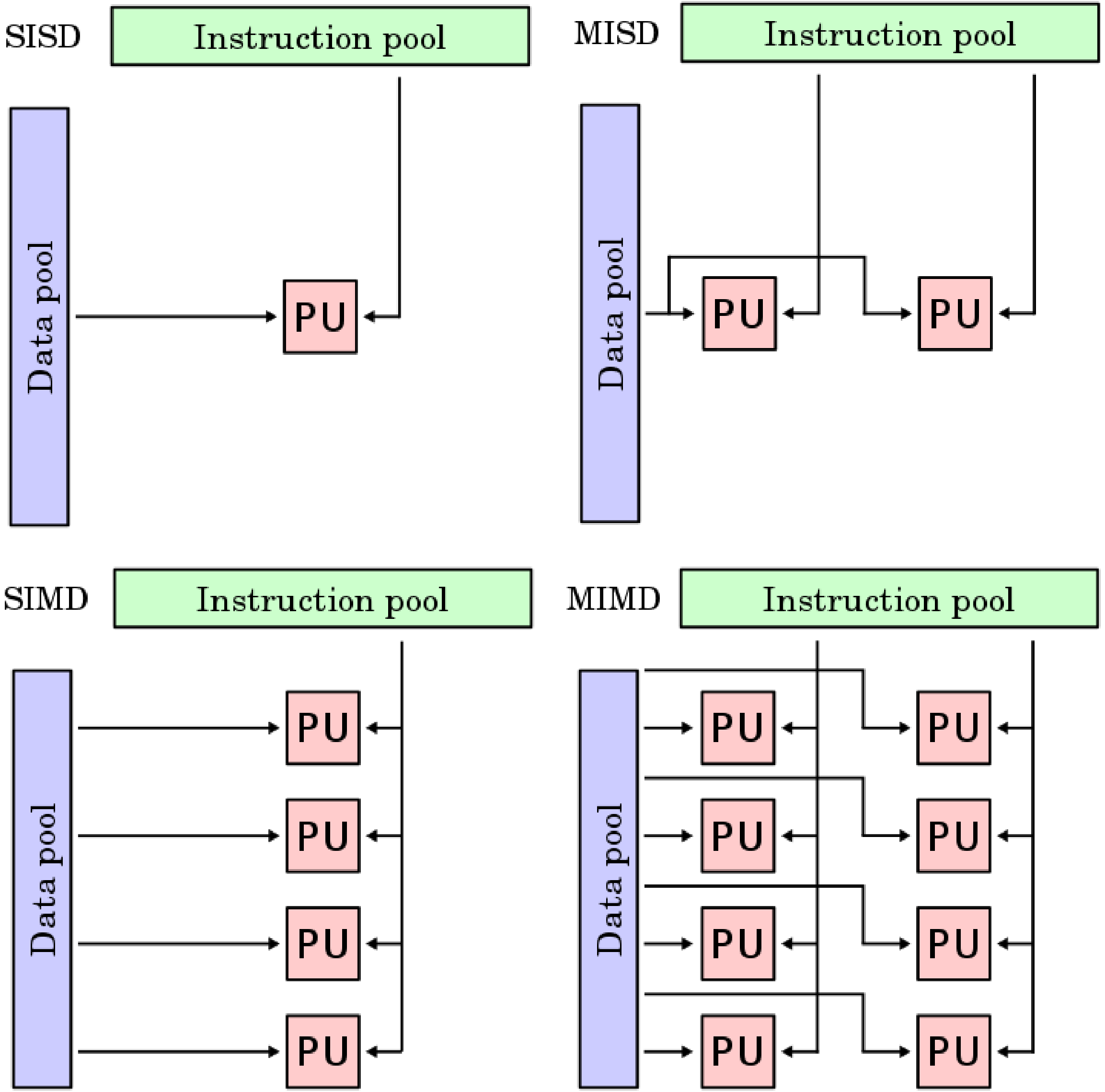
Flynn's taxonomy

- **Single Instruction Single Data (SISD)** ==> uniprocessor
- **Single Instruction Multiple Data (SIMD)**:

  The same instruction is applied to many data streams (e. g. MMX in x86 and Vector Processor).
- **Multiple Instruction Single Data (MISD)**:

  perform a series of computations on a single data stream in a pipelined.
- **Multiple Instruction Multiple Data (MIMD)** ==> multiprocessor

Flynn's taxonomy (Instruction & Data Streams)



Instruction and Data Streams

Flynn's taxonomy

		Data Streams: Single	Data Streams: Multiple
Instruction Streams	Single	**SISD** Intel Pentium 4	**SIMD** SSE Instructions of x86
Instruction Streams	Multiple	**MISD** No examples today	**MIMD** Intel Xeon e5345

MIMD

- **MIMD** implemented by two methods:
  - **Single Program Multiple Data (SPMD)**
    - A parallel program on a MIMD computer
    - Conditional code for different processors
    - Most common parallel programming style
  - **Multiple Program Multiple Data (MPMD)**
    - Multiple processors simultaneously operating at least two independent programs

SIMD

- Operate elementwise on vectors of data
  - e.g. **MMX** and **SSE** instructions in x86 & **Vector** processors
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

SIMD in x86: Multimedia Extensions

- Multimedia Extension (MMX)
  - subword parallelism for narrow integer data
- Streaming SIMD Extensions (SSE)
  - more instructions were added with single precision floating-point
- Advanced Vector Extensions (AVX)
  - supports the simultaneous execution of eight 64-bit floating-point numbers

Vector Processors

- Highly **pipelined function (execution) units**
- Stream data from/to **vector registers** to **units**
  - Data collected from **memory** into **registers**
  - Results stored from **registers** to **memory**
- Example: Vector extension to RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
  - Vector instructions
    - `fld.v`, `fsd.v`: load/store vector
    - `fadd.d.v`: add vectors of double
    - `fadd.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional RISC-V code:

```
      fld    f0,a(x3)      // load scalar a
      addi   x5,x19,512    // end of array X
loop: fld    f1,0(x19)     // load x[i]
      fmul.d f1,f1,f0      // a * x[i]
      fld    f2,0(x20)     // load y[i]
      fadd.d f2,f2,f1      // a * x[i] + y[i]
      fsd    f2,0(x20)     // store y[i]
      addi   x19,x19,8     // increment index to x
      addi   x20,x20,8     // increment index to y
      bltu   x19,x5,loop   // repeat if not done
```

- Vector RISC-V code:

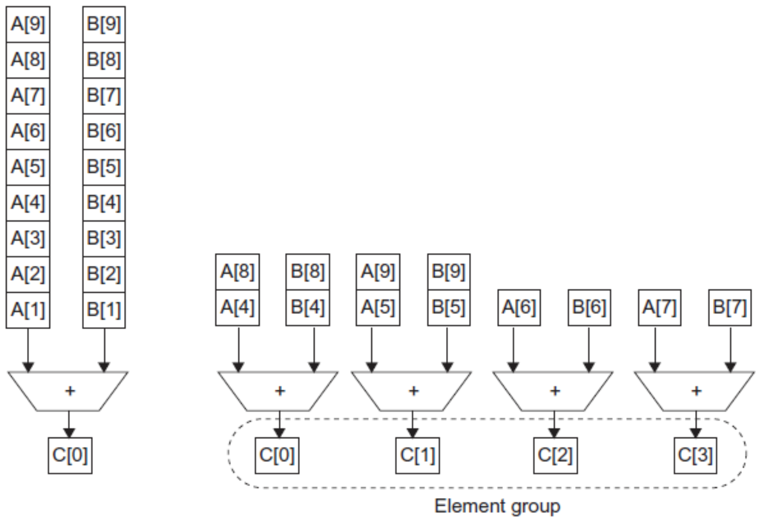
```
      fld       f0,a(x3)     // load scalar a
      fld.v     v0,0(x19)    // load vector x
      fmul.d.vs v0,v0,f0     // vector-scalar multiply
      fld.v     v1,0(x20)    // load vector y
      fadd.d.v  v1,v1,v0     // vector-vector add
      fsd.v     v1,0(x20)    // store vector y
```

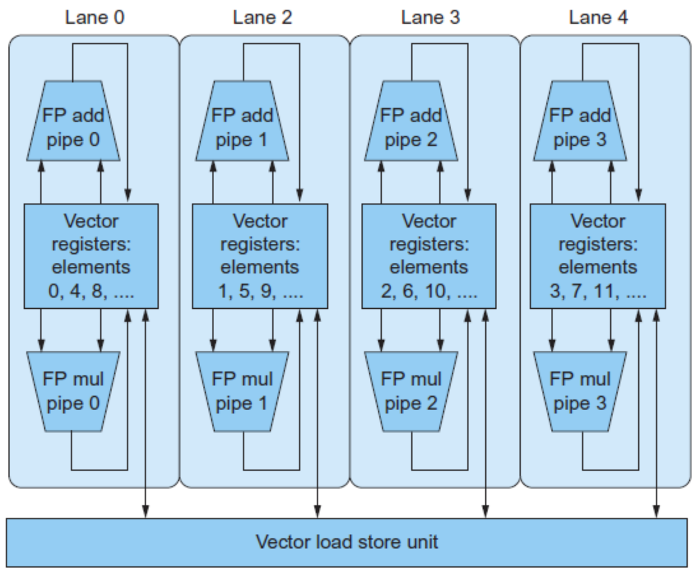
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

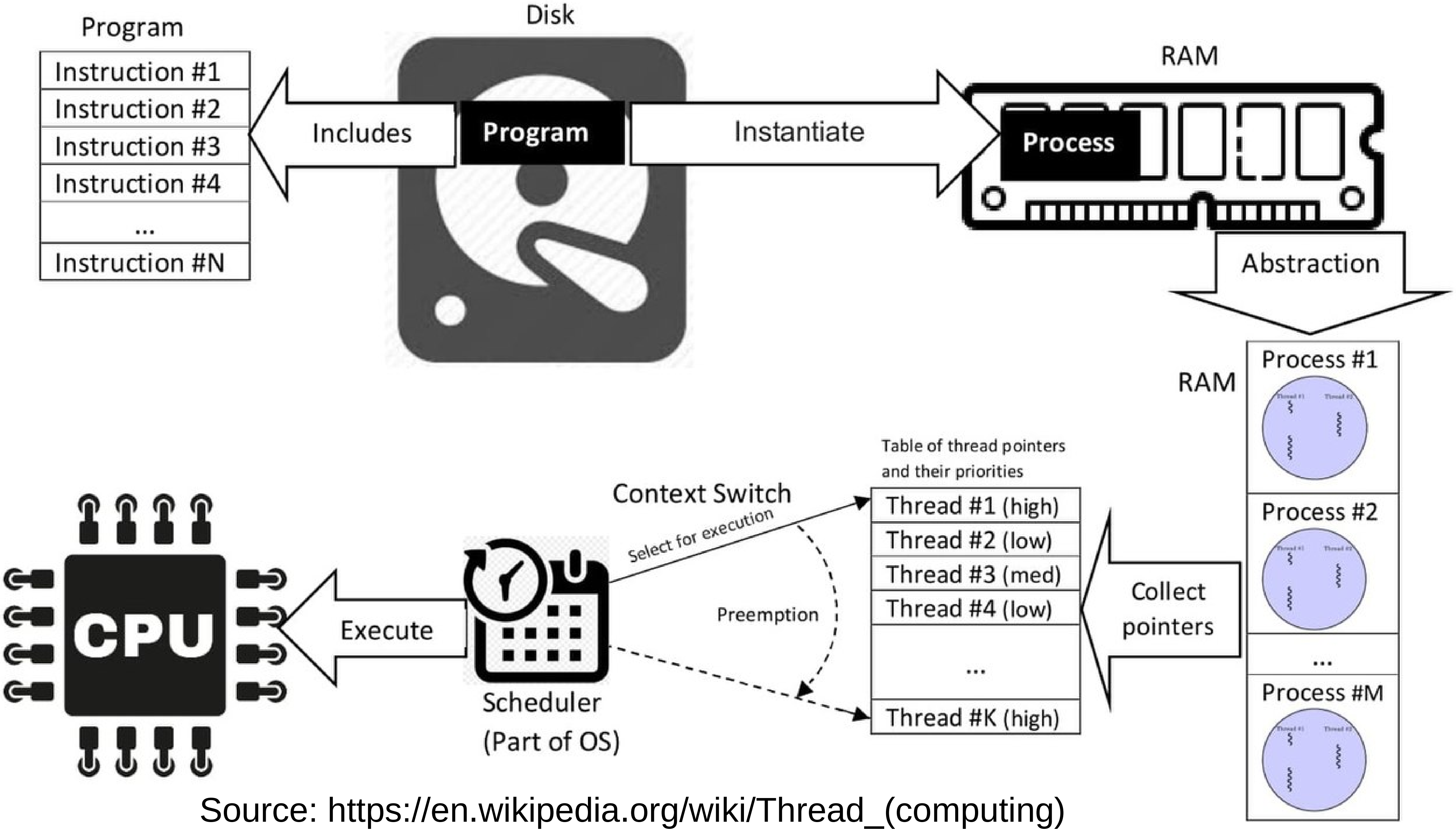
Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:





Multithreading



Source: https://en.wikipedia.org/wiki/Thread_(computing)

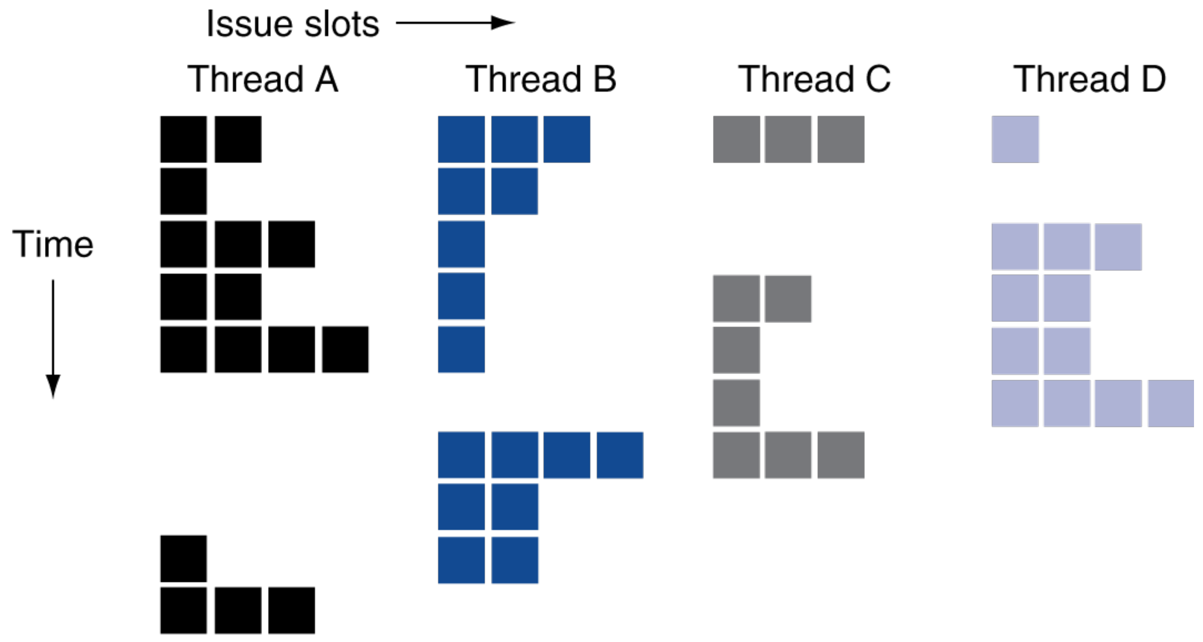
Multithreading

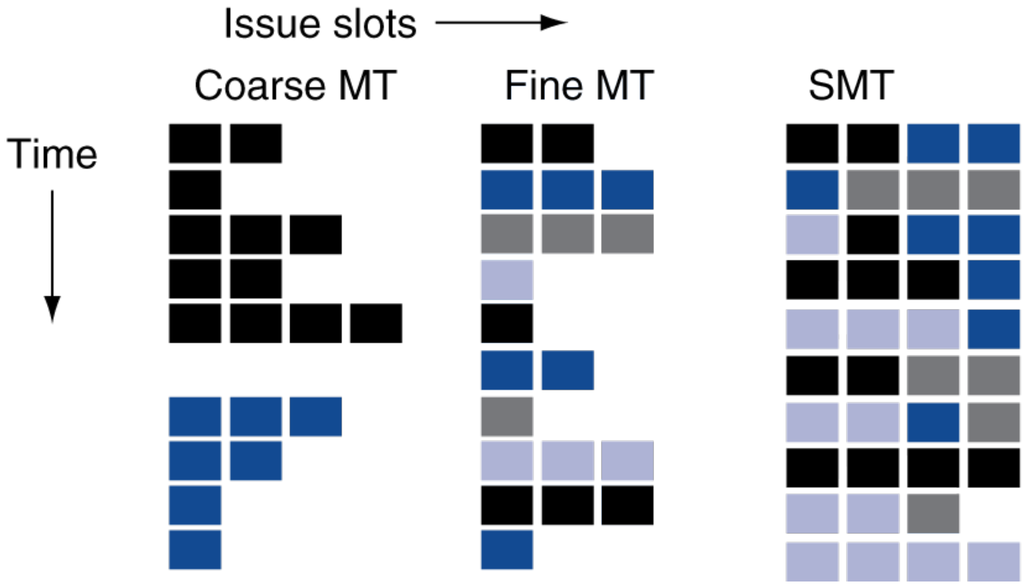
- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast **switching between threads**
- **Fine-grain multithreading**
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- **Coarse-grain multithreading**
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (e.g., data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled pipelined 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

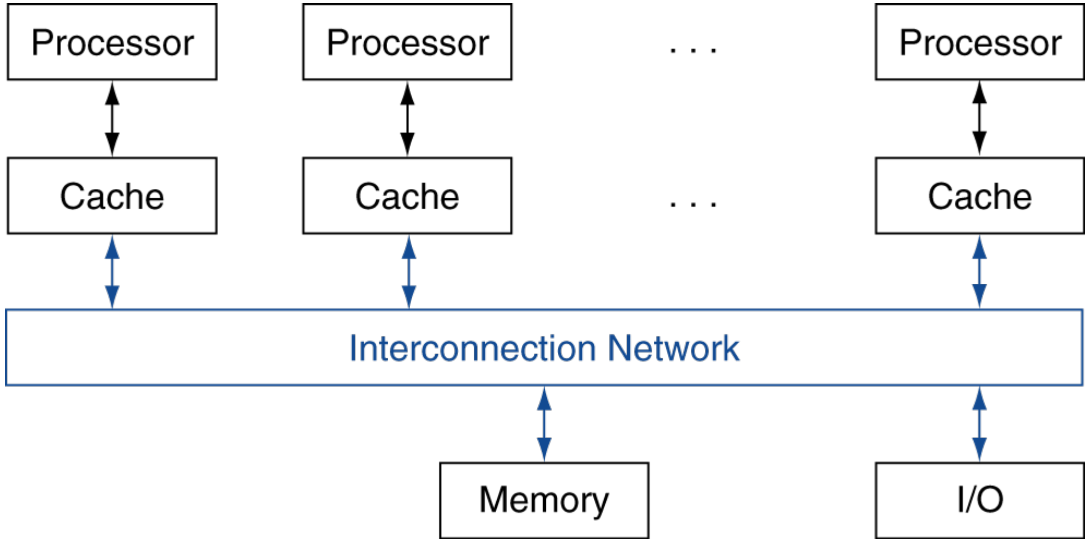
Multithreading Example





Shared Memory

- **Shared Memory Multiprocessor (SMP)**
  - Hardware provides **single physical address space** for all processors
  - **Synchronize** shared variables using **locks**
  - Memory access time
    - **UMA (uniform)** vs. **NUMA (nonuniform)**



Shared Memory

- **Synchronization**

  The process of coordinating the behavior of two or more processes, which may be running on different processors.

- **Lock**

  A synchronization device that allows access to data to only one processor at a time.

- Single address space multiprocessors styles:
  - **Uniform Memory Access (UMA)**
    - Same access time for all processors
  - **NonUniform Memory Access (NUMA)**
    - Different access time for different processors