

# Lecture 20: Concurrent Programming in the Real World

(High-Performance Computing, Data Center, and Human-Computer Interaction)

Xin Liu

Florida State University  
xliu15@fsu.edu

COP 4610 Operating Systems  
<https://xinliulab.github.io/FSU-COP4610-Operating-Systems/>

# Outline

- High-Performance computing (HPC)
- Data Center
- Human-Computer Interaction (HCI)
- Concurrent Programming in the Age of AI

# High-Performance Computing (HPC)

# Concurrent Programming in HPC

- The World's Most Expensive Sofa
  - The First Supercomputer (1976)
  - Single-processor system
  - 138 million FLOPs (Floating Point Operations per Second)
    - 40 times faster than IBM 370 at the time
    - Slightly better than embedded chips today
- Processed large data sets with one instruction



First Supercomputer (CRAY-1 from Los Alamos National Laboratory in 1976)

## HPC

"A technology that harnesses the power of supercomputers or computer clusters to solve complex problems requiring massive computation." (IBM)

- Computation-Centric
  - System Simulation: Weather forecasting, energy, molecular biology
  - Artificial Intelligence: Neural network training
  - Mining: Pure hash computation
  - TOP 500 (<https://www.top500.org/>)
    - 1st: Frontier (8,699,904 cores, 1206 PFLOPS)

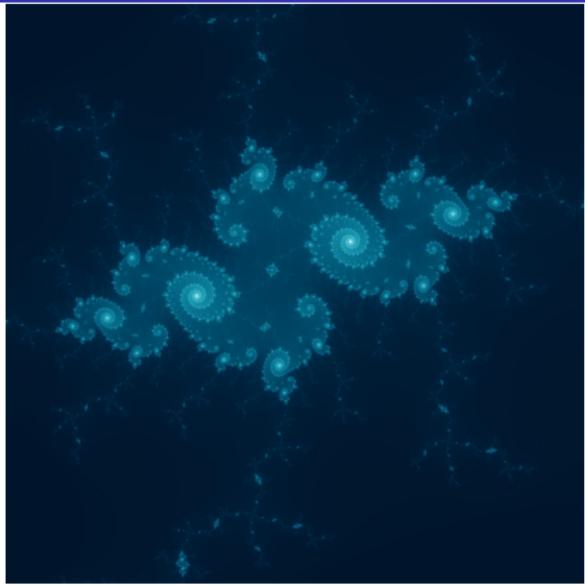
- How to Break Down Computation Tasks?
  - Computation Tasks need to be easy to parallelize
  - Task decomposition happens on two levels: machine and thread
  - Parallel and Distributed Computation: Numerical Methods
- **Independent Threads:**
  - Each thread executes its own task without needing to communicate with others
  - Useful for tasks that can be easily parallelized without shared data
  - Example: Performing different calculations on different parts of a dataset

# Challenges of Breaking Down Computation Tasks

- Large tasks cannot be accomplished without communication and shared memory between threads.
  - As calculations progress, tasks often require data sharing or synchronization, necessitating thread communication.
  - The complexity of parallelism can be more challenging than other application scenarios.
- 
- **Cooperating Threads:**
    - Threads need to communicate or synchronize to share data or resources
    - Require careful management
- 
- **Tools for Managing Communication:**
    - **MPI:** Manages message-passing between distributed threads or nodes
    - **OpenMP:** Handles shared memory parallel programming, synchronizing threads that access shared data

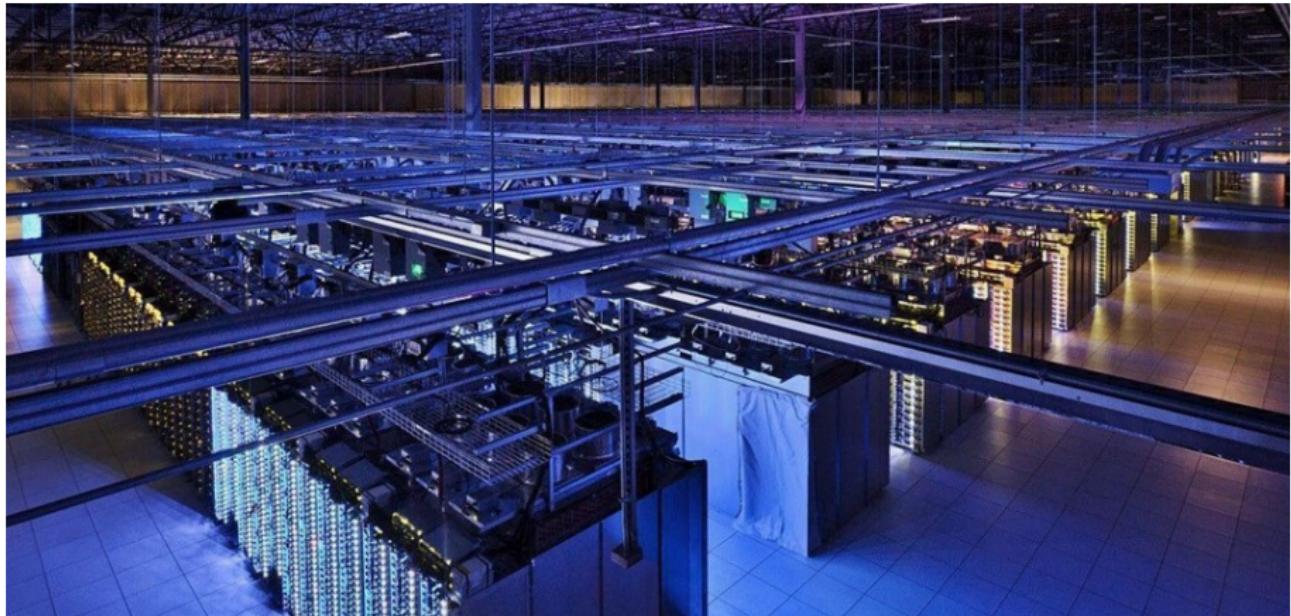
## Example: Mandelbrot Set

- $Z_{n+1}^2 = Z_n^2 + C$
- Each point in the Mandelbrot set iterates independently and is only influenced by its complex coordinate.
- Have A Try: [mandelbrot.c](#)



- While the number of cores is not the only factor, it is the most critical factor for determining thread execution efficiency.
- Core count helps estimate the system's computational capacity and parallel processing capabilities.
- Therefore, it is a key factor in HPC.

# Data Centers



Google Data Center

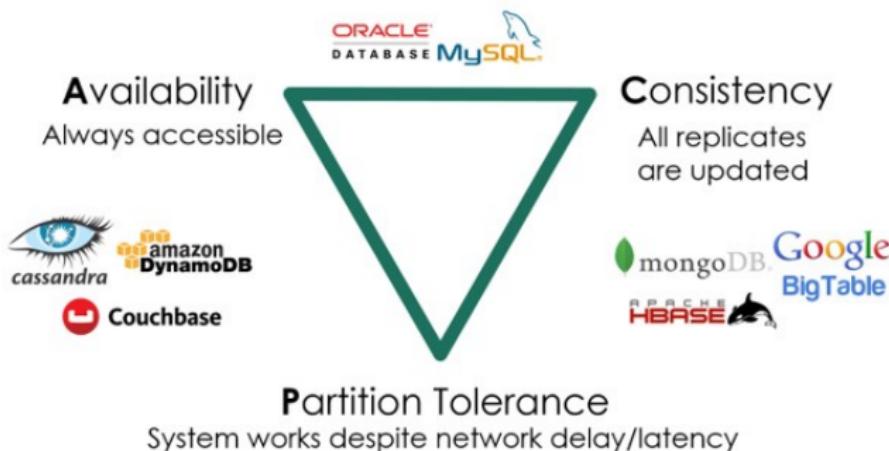
## Data Center

"A network of computing and storage resources that enable the delivery of shared applications and data." (CISCO)

- Data-Centric (Storage-Focused) Approach
  - Originated from internet search (Google), social networks (Facebook/Twitter)
  - Powers various internet applications: Gaming/Cloud Storage/ WeChat/Alipay/...
- The Importance of Algorithms/Systems for HPC and Data Centers
  - You manage 1,000,000 servers
  - A 1% improvement in an algorithm or implementation can save 10,000 servers

# Main Challenges of Data Center

- Serving massive, geographically distributed requests
- Data must remain consistent (Consistency)
- Services must always be available (Availability)
  - Must tolerate machine failures (Partition Tolerance)



# In-Class Quiz

# We focus on a single machine

## How to Maximize Parallel Request Handling with a Single Machine

- Key Metrics: QPS, Tail Latency, ...

# Maximizing Parallel Request Handling: Threads

## Advantages:

- True parallelism with multiple cores, enabling multiple execution flows.
- OS-level scheduling, allowing independent tasks to be managed efficiently by the operating system.
- Well-supported by most programming languages and operating systems.

## Disadvantages:

- Higher overhead due to system calls and context switching.
- Limited by the number of cores, potentially leading to contention and inefficiencies with too many threads.
- Memory overhead due to thread stacks and system resources.
- Have A Try: [thread.py](#)

# Maximizing Parallel Request Handling: Coroutines

## Advantages:

- More lightweight than threads, as they don't require system calls or context switches.
- Have A Try: [coroutine.py](#)

# Why are Coroutines More Lightweight?

- **User-Space Scheduling:** Coroutines are managed in user space and do not require kernel intervention, avoiding the overhead of system calls.
- **Minimal Context Switching:** Switching between coroutines requires saving only a small amount of information:
  - **Execution Position:** The point in the code where the coroutine yields or resumes (similar to the program counter in threads).
  - **Local Variables and Stack Frame:** The current state of local variables and the execution stack.
- **Comparison with Threads:** Threads require the operating system to save and restore more extensive context:
  - All CPU registers, including general-purpose and floating-point registers.
  - Program counter and stack pointer, which determine the execution position and stack location.
  - Thread-specific kernel data structures, which manage the thread's scheduling and other metadata.

# Disadvantages of Coroutines

- No true parallelism
  - A single thread can only run one coroutine at a time.
  - Although multiple coroutines can exist within the same thread, only one is active at any given moment.
  - The quick switching between coroutines creates the illusion of concurrency.
- **Blocking Operations:**
  - If a coroutine encounters a blocking operation (e.g., system calls), it blocks the entire thread.
  - This means all other coroutines in the same thread are also blocked, causing a significant performance issue.
- Requires manual yielding: Developers must manually control when a coroutine yields, which can lead to more complex code management.
- Less well-supported: Coroutines are not as universally supported across programming languages as threads.

# Maximizing Parallel Request Handling: Go

Goroutines = Threads + Coroutines

## **Advantages:**

- Extremely lightweight
- Enable true parallel execution on multiple cores
- Ideal for high-concurrency systems with minimal developer management
  - Efficient CPU utilization, achieving near 100% performance

## **Disadvantages:**

- Go runtime's scheduling decisions are opaque, making it harder to control execution flow.
- Debugging and profiling goroutines can be more difficult due to their lightweight nature and runtime control.
- Not available in all languages, limited to the Go ecosystem.

# Why Goroutines = Threads + Coroutines?

- **When a Goroutine encounters a blocking system call:**
  - Automatically converts blocking system calls (e.g., file I/O) into non-blocking operations.
  - Moves the Goroutine off the current thread and schedules another Goroutine to continue execution.
  - This ensures that no Goroutine blocks the entire system, maximizing concurrency.

Have A Try: [pc.go](#)

# Human-Computer Interaction (HCI)

## The Web 2.0 Era (1999)

- The Internet brought people closer together.
- "Users were encouraged to provide content, rather than just viewing it."
- You can even find early hints of "Web 3.0"/Metaverse in this period.

## What made Web 2.0 possible?

- Concurrent programming in browsers: Ajax (Asynchronous JavaScript + XML)
- HTML (DOM Tree) + CSS represented everything you could see.
  - JavaScript allowed dynamic changes to the DOM.
  - JavaScript also enabled connections between local machines and servers.

**With that, you had the whole world at your fingertips!**

# Features and Challenges

## Features:

- Not very complex
- Minimal computation required
  - The DOM tree is not too large (humans can't handle huge trees anyway)
  - The browser handles rendering the DOM tree for us
- Not too much I/O, just a few network requests

## Challenges:

- Too many programmers, especially for beginners
- Expecting beginners to handle multithreading with shared memory would lead to a world full of buggy applications!

# Single-Threaded + Event Loop

## Asynchronous with minimal but sufficient concurrency:

- Single thread, global event queue, sequential execution (run-to-complete)
- Time-consuming APIs (Timer, Ajax, etc.) return immediately
- When conditions are met, a new event is added to the queue

### Example: Chained Ajax Calls

```
$ajax( { url: 'https://xxx.yyy.zzz/login',
  success: function(resp) {
    $ajax( { url: 'https://xxx.yyy.zzz/cart',
      success: function(resp) {
        // do something
      },
      error: function(req, status, err) { ... }
    }
  },
  error: function(req, status, err) { ... }
);
```

# Solution: Asynchronous Event Model

## Advantages:

- Concurrency model is greatly simplified
  - Function execution is atomic (no parallel execution, reducing the chance of concurrency bugs)
- APIs can still run in parallel
  - Suitable for web applications where most time is spent on rendering and network requests
  - JavaScript code only "describes" the DOM Tree

## Disadvantages:

- Callback hell (the infamous "spaghetti code")
- As seen in the previous example, nesting 5 levels deep makes the code nearly unmaintainable

# Asynchronous Programming: Promise

## Definition:

- The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- 

## Promise: An Embedded Language for Describing Workflows

- **Chaining:**

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/
    two.js"))
  .then(script => loadScript("/article/promise-chaining/
    three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared
    there
  })
  .catch(err => { ... });

```

## Promise: An Embedded Language for Describing Workflows

- Fork-join:

```
a = new Promise((resolve, reject) => { resolve('A') });
b = new Promise((resolve, reject) => { resolve('B') });
c = new Promise((resolve, reject) => { resolve('C') });
Promise.all([a, b, c]).then(res => { console.log(res) });
```

# Advantages of Promise

- Readability: Promise chaining improves code readability by avoiding deeply nested callbacks, making asynchronous operations easier to follow.
- Error Handling: Provides a clear and structured way to handle errors through `.catch()`, reducing complexity compared to traditional callback error handling.
- Control Flow: Promises enable better control over the execution order of asynchronous tasks, ensuring that steps are completed in sequence.
- Flexibility: Easily integrates with modern JavaScript features like `async/await` for even cleaner and more readable code.

# Async-Await: Even Better

## async function:

- Always returns a Promise object
- `async_func()` - fork
- `await promise` - join

```
A = async () => await $.ajax('/hello/a');
B = async () => await $.ajax('/hello/b');
C = async () => await $.ajax('/hello/c');

hello = async () => await Promise.all([A(), B(), C()]);

hello()
  .then(window.alert)
  .catch(res => { console.log('fetch failed!') });
```

# Concurrent Programming in the Age of AI

## 8 × Tesla V100: The Computational Core of DGX-1

- **DGX-1 is a complete AI supercomputer** designed by NVIDIA.
- It integrates **8 Tesla V100 GPUs** into a single system.
- These GPUs are interconnected using **NVSwitch**, providing high-speed GPU-to-GPU communication (300GB/s).
- Compared to standalone GPUs, DGX-1 includes:
  - **2 × Intel Xeon CPUs** for coordination.
  - **512GB DDR4 RAM** for system memory.
  - **15TB NVMe SSD** for high-speed storage.
  - Optimized power and cooling system (3.2kW power consumption).
- **Performance:** 170 TFLOPS @ 3.2kW
- **Comparison:** CRAY-1: 138 MFLOPS @ 115kW

## 8 × Blackwell GPUs: The Computational Core of DGX B200

- **DGX B200 is the latest AI supercomputer** designed by NVIDIA.
- It integrates **8 Blackwell GPUs** into a single system.
- These GPUs are interconnected using **NVLink and NVSwitch**, providing ultra-high-speed communication.
- Compared to standalone GPUs, DGX B200 includes:
  - **Optimized AI acceleration** for large-scale training and inference.
  - **High-bandwidth memory (HBM)** for faster data access.
  - Advanced **power and cooling solutions** for efficient operation.
- **Performance:**
  - **72 PFLOPS (Training), 144 PFLOPS (Inference)** @ 14.3kW

# Computation Behind Large Language Models

"Attention Is All You Need"

LLM Visualization

# Single Compute-Intensive Slice (1): SIMD

## Single Instruction, Multiple Data

- Tensor Instructions (Tensor Core): Mixed Precision

$$A \times B + C$$

- A single instruction performs a  $4 \times 4$  matrix operation.

$$D = \begin{matrix} \text{FP16 or FP32} & \left( \begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) & \text{FP16} \\ & + & \end{matrix}$$
$$\left( \begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) \text{FP16} + \begin{matrix} \left( \begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right) & \text{FP16 or FP32} \end{matrix}$$

## x86 SIMD Evolution:

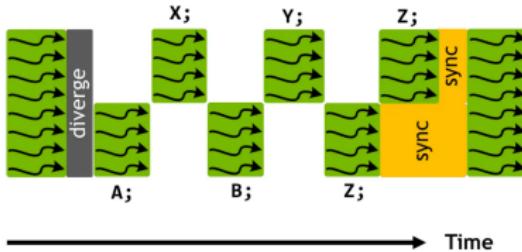
- MMX (MultiMedia eXtension, 64-bit MM) → SSE (Streaming SIMD Extensions, 128-bit) → AVX (Advanced Vector eXtensions, 256-bit) → AVX512 (512-bit)

# Single Compute-Intensive Slice (2): SIMT

## Single Instruction, Multiple Threads

- One PC (Program Counter) controls **32 execution flows simultaneously**.
  - The number of logical threads can be even larger.
- Each execution flow has its own **registers**.
  - Three registers ( $x$ ,  $y$ , and  $z$ ) are used to store the "thread ID".
- Then, **a massive number of threads!**

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;  
__syncwarp()
```



# Takeaways

- High-Performance Computing
  - Focus: Task Decomposition
  - Pattern: Producer-Consumer
  - Technologies: MPI / OpenMP
- Data Centers
  - Focus: System Calls
  - Pattern: Threads-Coroutines
  - Technologies: Goroutine
- Human-Computer Interaction
  - Focus: Usability
  - Pattern: Single Thread + Event Loop
  - Technologies: Promise
- AI-Era
  - Focus: Throughput at scale, GPU utilization
  - Pattern: Data/Model/Tensor Parallelism, Pipeline Parallelism
  - Technologies: CUDA/ROCM, NCCL/All-Reduce, PyTorch Distributed, DeepSpeed, Megatron-LM, TensorRT/XLA