

Lecture 9: Concurrent Programming in the Real World

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

Xin Liu

Florida State University
xl24j@fsu.edu

COP 4610 Operating Systems
<https://xinliulab.github.io/cop4610.html>
September 29, 2024

- High-Performance computing (HPC)
- Data Center
- Human-Computer Interaction (HCI)

Recap: 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)

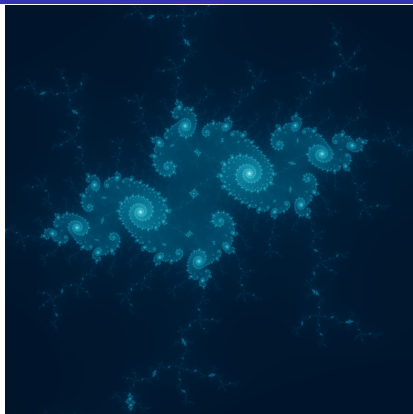
- Computation-Centric
 - System Simulation: Weather forecasting, energy, molecular biology
 - Artificial Intelligence: Neural network training
 - Mining: Pure hash computation
 - TOP 500 (<https://www.top500.org/>)
 - 1st: Frontier (8, 699,904 cores, 1206 PFLOS)

Recap: Main Challenges of HPC

- How to Break Down Computation Tasks?
 - Computation Graphs need to be easy to parallelize
 - Task decomposition happens on two levels: machine and thread
 - Parallel and Distributed Computation: Numerical Methods
- How Do Threads Communicate?
 - Communication happens not only between nodes/threads but also with any shared memory access
 - Tools:
 - **MPI** - "a specification for developers and users of message-passing libraries"
 - **OpenMP** - "multi-platform shared-memory parallel programming in C/C++ and Fortran"

Recap: 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.
- Link of Code:
[Mandelbrot Set Code](#)



- 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.

Recap: Concurrent Programming in Data Centers



Google Data Center

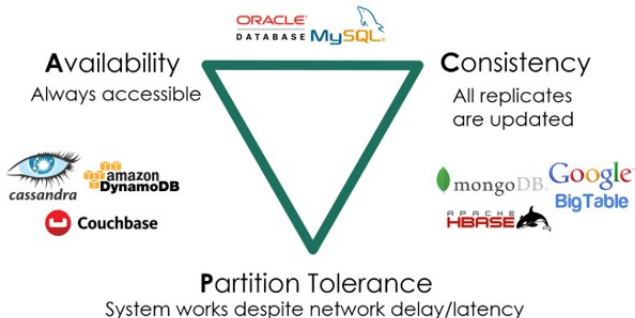
HPC

“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

Recap: 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)



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.
- Link of Code: [Thread Example Code](#)

Maximizing Parallel Request Handling: Coroutines

Advantages:

- More lightweight than threads, as they don't require system calls or context switches.
- Allow cooperative multitasking, reducing overhead compared to threads.
- Efficient for I/O-bound tasks as they can pause and resume without blocking other tasks.

Disadvantages:

- No true parallelism; all coroutines run on a single thread, limited by CPU core availability.
- Requires manual yielding and may lead to complex code management.
- Less well-supported across some programming languages compared to threads.

Why Coroutines Don't Require Context Switching

No Kernel Involvement:

- Thread context switching requires the OS to save and restore CPU registers, stack, and other states.
- Coroutines are scheduled in user space, not by the operating system kernel.
- Switching between coroutines happens entirely in user code, eliminating the need for kernel-level context switching as seen in threads.
 - The programmer explicitly controls when to yield execution (e.g., through 'await' or 'yield'), avoiding frequent kernel-level switches.
- Link of Code: [Coroutine Example Code](#)

Maximizing Parallel Request Handling: Go

Goroutines = Threads + Coroutines

Advantages:

- Goroutines are extremely lightweight and managed by Go's runtime, with low overhead.
- Automatic scheduling of goroutines by Go runtime, allowing them to run in parallel across multiple cores.
 - Near 100% performance efficiency in utilizing CPU resources
- Excellent for building high-concurrency systems with minimal developer management.

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:**
 - Go runtime automatically converts blocking system calls (e.g., file I/O) into non-blocking operations.
 - Go runtime 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.
- Link of Code: [Goroutine Example Code](#)

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!

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 => { ... });
```

- **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) });
```

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!') });
```

- 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: Event-Stream Graph
 - Technologies: Promise.