Cloud Contraptions
TRAINING & CONSULTING

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

Rust Parallel Programming

Class Duration

- 14 hours of intensive training with live instruction delivered over two to four days to accommodate varied scheduling needs

Student Prerequisites

- Software engineers with Rust programming experience
- Software engineers with a very good understanding of Rust's approach to memory management and data structures

Target Audience

- Systems programmers building high-performance applications
- Backend developers optimizing compute-intensive workloads
- Engineers migrating parallel C/C++ code to Rust
- Developers building real-time data processing pipelines
- Teams working on scientific computing or simulations

Description

This course teaches Rust developers how to build high-performance parallel and concurrent applications. Students will learn to leverage Rust's powerful ecosystem including Rayon for data parallelism, Crossbeam for advanced concurrency primitives, and atomic operations for lock-free programming. The course also covers unsafe Rust and FFI for systems-level integration, plus declarative and procedural macros for metaprogramming. Through hands-on exercises, students will gain practical experience parallelizing real workloads while maintaining Rust's safety guarantees.

Learning Outcomes

- Distinguish between concurrency and parallelism and choose the right approach for different workloads
- Use Rayon to convert sequential code to parallel with minimal changes
- Implement work-stealing patterns and configure thread pools for optimal performance
- Apply Crossbeam utilities for scoped threads, concurrent queues, and advanced synchronization

- Understand atomic operations and memory ordering for lock-free data structures
- Design inter-process communication using shared memory and IPC channels
- Write safe abstractions over unsafe code and integrate with C libraries via FFI
- Create declarative macros with macro_rules! and procedural derive macros

Training Materials

All students receive comprehensive courseware covering all topics in the course. Courseware is distributed via GitHub in the form of documentation and extensive code samples. Students practice the topics covered through challenging hands-on lab exercises.

Software Requirements

Students will need a free, personal GitHub account to access the courseware. Student will need permission to install Rust and Visual Studio Code on their computers. Also, students will need permission to install Rust Crates and Visual Studio Code Extensions. If students are unable to configure a local environment, a cloud-based environment can be provided.

Training Topics

Parallel Programming

- What is Parallel Programming?
- Concurrency vs Parallelism
- When to Use Parallel Programming

Data Parallelism with Rayon

- Introduction to Rayon
- Parallel Iterators (par_iter, par_iter_mut)
- Converting Sequential to Parallel Code
- Parallel Methods (map, filter, for_each, fold)
- Parallel Sorting (par_sort, par_sort_by)
- par_bridge for Non-Parallel Iterators
- Work Overhead and When Not to Parallelize

Thread Pools and Work Stealing

- What is Work Stealing?
- Rayon’s Thread Pool Architecture
- Configuring Thread Pool Size
- The join() Primitive
- Recursive Work Splitting
- Scoped Threads with Crossbeam

Crossbeam Utilities
- Overview of Crossbeam Crates
- crossbeam-channel (MPMC Channels)
- crossbeam-deque (Work-Stealing Deques)
- crossbeam-queue (Concurrent Queues)
- crossbeam-utils (Scoped Threads, Backoff)
- ArrayQueue and SegQueue

Atomics and Memory Ordering
- Atomic Types (AtomicBool, AtomicUsize, AtomicPtr)
- Memory Ordering (Relaxed, Acquire, Release, SeqCst)
- Compare-and-Swap (CAS) Operations
- Building Lock-Free Data Structures
- When to Use Atomics vs Locks

Inter-Process Communication
- When to Use Multiple Processes
- ipc-channel (Servo’s IPC Library)
- Shared Memory with POSIX shm_open
- Semaphores for Process Synchronization
- ipmpsc (Ring Buffer IPC)
- Trade-offs: Threads vs Processes

Unsafe Rust
- What is Unsafe Rust?
- When and Why to Use Unsafe
- Unsafe Superpowers Overview

Raw Pointers and Memory
- Raw Pointers (*const T,* mut T)
- Creating Raw Pointers from References

- Dereferencing Raw Pointers
- Pointer Arithmetic
- Null Pointer Handling

Unsafe Functions and Traits
- Calling Unsafe Functions
- Writing Unsafe Functions
- Unsafe Traits (implementing and defining)
- Safe Abstractions over Unsafe Code

Foreign Function Interface (FFI)
- What is FFI?
- The extern Keyword and ABI
- Calling C Functions from Rust
- Exposing Rust Functions to C (#[no_mangle])
- Using CString and CStr for Strings
- Portable Type Aliases (c_int, c_char, etc.)
- Handling Callbacks Across FFI
- Preventing Panics at FFI Boundaries

FFI Tooling
- rust-bindgen (Generate Rust from C headers)
- cbindgen (Generate C headers from Rust)
- The libc Crate

Macros and Metaprogramming
- What is a Macro?
- Declarative vs Procedural Macros
- When to Use Macros vs Functions

Declarative Macros (macro_rules!)
- Define a Macro with macro_rules!
- Matchers and Transcribers
- Fragment Specifiers (ident, expr, ty, tt, etc.)
- Repetition Patterns ($(...),* and $(...),+)
- Handling Trailing Commas
- Multiple Match Arms
- Macro Hygiene Rules

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

Procedural Macros

- What are Procedural Macros?
- The Three Types (Derive, Attribute, Function-like)
- Setting Up a proc-macro Crate
- Working with TokenStreams

Derive Macros

- Creating Custom Derive Macros
- The syn Crate (Parsing)
- The quote Crate (Code Generation)
- The proc-macro2 Crate
- Helper Attributes with darling

Attribute and Function-like Macros

- Attribute Macros (#[my_macro])
- Function-like Procedural Macros
- Real-World Use Cases