Faculty of Mathematics
and Computer Science

Parallel Computing in Rust

Author name

Department of Computer Science, Babeş-Bolyai University
1, M. Kogalniceanu Street, 400084, Cluj-Napoca, Romania

Abstract

This paper presents an overview of Rust, a modern systems programming language designed with performance and safety in mind. It explores Rust's approach to parallel and concurrent programming through its ownership model, thread safety guarantees, and concurrency primitives. Various models of parallelism in Rust—such as threads, async/await, and data parallelism using libraries like Rayon—are explained with code examples. A comparison with other languages like C++, Java, Go, and Python highlights Rust's advantages in memory safety and performance. The paper also discusses practical applications of Rust in real-world systems and outlines both its strengths and limitations in the context of parallel computing.

Keywords:

1. Introduction

Parallel computing has become an essential paradigm in modern software development, driven by the increasing demand for performance and the prevalence of multi-core processors. It enables programs to execute multiple operations or tasks simultaneously, making better use of hardware resources and reducing execution time. To fully leverage parallelism, programming languages and models must provide efficient, safe, and user-friendly tools for concurrent and parallel execution.

Rust is a systems programming language that has gained significant popularity due to its unique approach to safety, performance, and concurrency. Developed by Mozilla and supported by an active open-source community, Rust provides fine-grained control over memory management while preventing common programming errors such as null pointer dereferencing and data races through its ownership model and strong type system.

This paper explores how parallelism is achieved in Rust, presenting both low-level and high-level concurrency constructs available in the language and its ecosystem. It includes practical examples demonstrating different parallel programming patterns and compares Rust's approach to those of other commonly used languages in the domain, such as C++, Java, and Go. The paper also evaluates the strengths and limitations of Rust as a parallel computing model, considering both performance characteristics and developer experience.

The goal is to provide a comprehensive understanding of Rust's capabilities in parallel computation and assess its suitability for building robust, efficient, and scalable software systems.

2. General Presentation of Rust

Rust is a modern systems programming language designed for performance, safety, and concurrency. First released in 2010 and reaching version 1.0 in 2015, Rust was originally developed by Mozilla to address the limitations of existing low-level programming languages such as C and C++. Rust offers fine-grained control over system resources while introducing memory safety guarantees without the need for a garbage collector. [3]

At the core of Rust's design is its **ownership model**, a compile-time system that enforces strict rules about how memory is accessed and shared between parts of a program. This model eliminates entire classes of bugs such as use-after-free, double free, and data races, which are common in other systems languages. By making these safety guarantees a part of the language itself, Rust allows developers to write highly performant and reliable code.

Rust also emphasizes zero-cost abstractions, meaning that higher-level constructs do not incur runtime penalties compared to equivalent low-level code. This feature makes Rust ideal for performance-critical applications such as game engines, operating systems, and embedded systems. [3]

In addition to safety and performance, Rust provides strong support for concurrency. The standard library includes primitives for thread creation and synchronization, and the language ecosystem offers popular libraries such as `rayon` for data parallelism and `tokio` for asynchronous programming.

Key Features of Rust

- **Ownership and Borrowing:** A unique model that ensures memory safety and thread safety at compile time.
- **No Garbage Collection:** Manual but safe memory management enabled by the compiler's strict checks.
- **Zero-cost Abstractions:** High-level constructs that translate directly into efficient machine code.
- **Pattern Matching and Algebraic Data Types:** Enables expressive and concise code.
- **Package Manager and Build System:** `Cargo` handles dependency resolution, compilation, and testing.

Rust's Ecosystem

Rust benefits from a vibrant ecosystem and tooling support. The central package registry, `crates.io`, hosts thousands of libraries that span from web development (`actix-web`, `rocket`) to scientific computing (`ndarray`, `nalgebra`) and asynchronous programming (`tokio`, `async-std`).

Use Cases

Rust is used in a variety of domains:

- **Systems Programming:** Operating systems (e.g., Redox OS), file systems.
- **Web Development:** High-performance backends and APIs.
- **Embedded Systems:** Applications in automotive, robotics, and IoT.
- **Concurrency and Parallelism:** Scalable and safe multi-threaded applications.

Example: Hello World in Rust

```
fn main() {
    println!("Hello, world!");
}
```

Rust's syntax is influenced by C, but it provides modern safety features and ergonomic abstractions that make it suitable for a broad range of applications.

3. Rust and Parallelism

Rust provides powerful tools for writing safe and efficient parallel and concurrent programs. Its strict compile-time checks ensure memory and thread safety, which are crucial in multi-threaded environments. Rust supports several models for parallel computation, ranging from low-level thread spawning to high-level data and task parallelism through external libraries. [2]

3.1. 1. Native Threading with `std::thread`

Rust's standard library includes basic support for spawning threads using the `std::thread` module. This allows developers to run tasks in parallel by creating multiple threads, each executing independently.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!("Running in a separate thread!");
    });

    handle.join().unwrap();
}
```

Each call to `thread::spawn` creates a new OS thread, and the main thread can wait for its completion using `join()`. While this model is powerful, it requires manual management of data sharing and synchronization, which can be error-prone if not handled correctly.

3.2. 2. Data Parallelism with `rayon`

The `rayon` crate provides a high-level abstraction for data parallelism. It allows easy parallel iteration over collections, enabling multi-core utilization without explicit thread management. [5]

```
use rayon::prelude::*;

fn main() {
    let data: Vec<i32> = (1..10_000).collect();
    let squared: Vec<i32> = data.par_iter().map(|x| x * x).collect();
    println!("First 5 squared numbers: {:?}", &squared[..5]);
}
```

Rayon automatically distributes work across threads, handling scheduling and load balancing internally. This makes it ideal for CPU-bound tasks that involve processing large datasets.

3.3. 3. Asynchronous Programming with `tokio`

For I/O-bound applications, asynchronous programming can be more efficient than multi-threading. Rust's async ecosystem is built around the `async/await` syntax and runtimes like `tokio`. [6]

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    let task = tokio::spawn(async {
        sleep(Duration::from_secs(2)).await;
```

```
        println!("Async task completed!");
    });

    task.await.unwrap();
}
```

In this example, the task sleeps asynchronously without blocking a thread. Tokio uses a lightweight task scheduler to efficiently manage thousands of tasks within a few threads, making it suitable for high-performance servers and networked applications.

3.4. 4. Message Passing with Channels

Rust also supports concurrency using message passing via channels. This model decouples threads by avoiding shared memory and using communication to synchronize data.

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send("Message from thread").unwrap();
    });

    let received = rx.recv().unwrap();
    println!("{}", received);
}
```

This pattern encourages isolation between threads, reducing the risk of race conditions and deadlocks.

3.5. Summary of Parallel Models in Rust

Model	**Library**	**Use Case**
Thread-based	`std::thread`	Fine-grained control over threads
Data parallelism	`rayon`	Parallel processing of collections
Async tasks	`tokio`, `async-std`	Scalable I/O-bound applications
Message passing	`mpsc`, `crossbeam`	Safe inter-thread communication

Table 1. Rust parallelism models and their typical applications

Rust's approach to parallelism balances control and safety. By leveraging its strong type system and ownership model, developers can write concurrent code with confidence, minimizing the risks traditionally associated with parallel programming.

4. Comparison with Other Models

To better understand Rust's position in the landscape of parallel computing, it is useful to compare it with other widely used programming languages and models that support concurrency and parallelism. Each language offers different abstractions, safety guarantees, and runtime behaviors that influence how parallel computation is implemented and executed.

4.1. Rust vs. C++

C++ offers low-level control over hardware and memory and is widely used in high-performance computing. However, manual memory management and the absence of built-in thread safety can lead to errors such as data races and memory leaks. Rust addresses these issues by enforcing strict ownership and borrowing rules at compile time, eliminating most concurrency-related bugs before runtime. [4]

4.2. Rust vs. Java

Java provides high-level concurrency APIs such as `java.util.concurrent` and features like the Fork/Join framework. While Java's garbage collector simplifies memory management, it introduces nondeterminism and potential pauses in performance-critical applications. Rust, with its deterministic memory management and zero-cost abstractions, can often achieve better real-time performance and predictability, though at the cost of a steeper learning curve.

4.3. Rust vs. Go

Go simplifies concurrency using lightweight threads called goroutines and communication through channels. While this model is easy to use and effective for many tasks, Go's runtime does not enforce memory safety in the same way as Rust. Rust offers more control and stricter safety guarantees but requires developers to manage complexity more explicitly. [1]

Summary Table

Feature	**Rust**	**C++**	**Java**	**Go**
Memory Safety	Enforced at compile time	Manual	GC-managed	GC-managed
Concurrency Model	Threads, async, channels	Threads, locks	Threads, Fork/Join	Goroutines, channels
Race Condition Prevention	Yes (compile-time)	No	No	No
Garbage Collection	No	No	Yes	Yes
Ease of Use	Moderate to Hard	Hard	Moderate	Easy
Performance	High	High	Moderate	Moderate to High
Abstraction Level	Low to High	Low	High	High

Table 2. Comparison of Rust with other parallel computing models

Rust stands out for combining memory safety and high performance without relying on a garbage collector. It achieves parallelism with fine control and safety guarantees, making it suitable for critical systems where correctness and efficiency are paramount. Other languages may offer simpler or more mature concurrency models, but they often trade safety or performance for ease of development.

5. Advantages and Disadvantages

Rust brings a unique set of strengths to parallel programming, largely due to its design choices around safety and performance. However, these same choices introduce some trade-offs that developers must consider.

Advantages

- **Memory and Thread Safety:** The ownership system and type checker prevent data races and other concurrency bugs at compile time.
- **High Performance:** Rust compiles to efficient machine code with zero-cost abstractions, often matching or exceeding C++ performance.

- **No Garbage Collector:** Predictable performance and low-latency behavior are ideal for real-time and embedded systems.
- **Modern Tooling:** Tools like `Cargo`, `clippy`, and `rustfmt` enhance development productivity.
- **Rich Ecosystem:** Libraries like `rayon`, `tokio`, and `crossbeam` support various concurrency and parallelism patterns.

Disadvantages

- **Steep Learning Curve:** The ownership and lifetime system can be challenging, especially for newcomers.
- **Complexity in Async and Multithreaded Code:** Writing efficient async code often involves understanding complex lifetimes and pinning.
- **Young Ecosystem (in some domains):** Although growing rapidly, Rust's ecosystem is still maturing compared to C++ or Java in some areas like scientific computing.
- **Compilation Time:** Compile times can be longer, especially for large projects or with heavy generics and macros.

In summary, Rust offers a powerful and safe environment for parallel programming but requires an initial investment in learning and adapting to its paradigms.

6. Use Cases and Real-World Applications

Rust's performance, safety, and concurrency features make it well-suited for a variety of real-world applications, particularly in systems where correctness and efficiency are critical.

- **Web Servers and Backends:** Frameworks like `Actix` and `Axum` use asynchronous Rust to build fast and scalable web services.
- **Operating Systems and Embedded Systems:** Projects like `Redox OS` and embedded firmware benefit from Rust's memory safety and low-level control.
- **Blockchain and Cryptography:** Rust is used by platforms like `Parity Ethereum` and `Solana` for secure, parallelizable transaction processing.
- **Game Engines and Graphics:** Game engines like `Bevy` utilize Rust's multithreading and performance features.
- **Command-line Tools:** Tools like `ripgrep` and `fd` showcase Rust's speed and safe parallel execution.

These examples highlight Rust's applicability across domains requiring high concurrency, performance, and reliability.

7. Conclusion

Rust presents a compelling model for parallel and concurrent programming, combining modern abstractions with low-level performance. Its strict compile-time checks for ownership and borrowing ensure thread and memory safety, significantly reducing bugs common in multi-threaded code.

By supporting multiple approaches to parallelism—threads, asynchronous tasks, data parallelism, and message passing—Rust offers developers flexibility without compromising safety. While it comes with a steeper learning curve compared to some languages, the benefits in performance and reliability are substantial.

As the ecosystem continues to mature, Rust is becoming an increasingly attractive choice for developing high-performance, safe, and scalable systems across domains such as web development, embedded systems, and blockchain technologies.

References

[1] David Anderson. Go: A language for concurrent programming. *Communications of the ACM*, 59(2):58–65, 2016.

[2] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.
[3] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.
[4] Scott Meyers. Effective c++ and memory management. *Software Development Times*, 2005.
[5] Nichols, Niko. Rayon: Data parallelism in rust. https://github.com/rayon-rs/rayon, 2020. Accessed: 2025-05-31.
[6] Tokio Project. Tokio: An asynchronous runtime for rust. https://tokio.rs, 2025. Accessed: 2025-05-31.