Evaluating Rust for Parallelizing GREP-like Search

Harry Qian
606003725

1 Introduction

Rust is a modern, low level language with a significant focus on safe memory management and catching memory errors early. In this report, we examine the possibility of using Rust to write a parallelized version of regex-search through log files. Specifically, we will analyze the approach of the language in terms of security, ease of use, flexibility, generality, performance, and reliability.

2 Rust's Ease of Use Features

Rust provides a relatively complete and well-maintained ecosystem around its core language. For example, it provides an official package manager, cargo, out-of-the-box. This makes it easier to find and use modules (called crates) compared to C / C++, which makes development smoother. Many utility functions and helpful classes to our parallelization project can be found on cargo.io, the Rust crate registry similar to NPM.

In fact, a quick search reveals a promising starting point to our exact application. `ripgrep` is a public domain project (with the UNLICNESE license) that implements multi-threaded matching with regex. The github page reports a 2x performance improvement over `grep -r` even in worst case scenarios with high match counts and one single large file.

This project is active and well-maintained, and serves as a valuable starting point to use or modify given our company's needs. In this aspect, Rust allows our team to work at both low and high levels of abstraction and is advantageous.

Another advantage of rust is a more extensive standard library than C / C++. Fore example, Semaphores and atomic integers are implemented by the Rust language, which is handy for writing parallelized programs.

A tradeoff in Ease-of-Use is brought about by Rust's focus on memory safety. Many features designed to catch bugs at compile time, such as handling memory ownership, transferring ownership between threads, and specifying lifetimes of references and structs all lead to possibly verbose syntax, especially if our development team is not familiar with Rust before. Rust's design philosophy heavily departs away from C/C++'s attitude where the developer ensures the program's behavior is well-defined and intended, and focuses on the compiler enforcing a strict set of rules that ensure memory safety.

3 Rust's Safety Features

For bugs related to concurrency (race conditions, deadlock), Rust provides similar primitives to Java (like mutexes and Atomic integers).

Thanks to the relatively simple nature of our utility (we don't write to files), race conditions should be relatively rare. Rust's buffered I/O will help maintain a queue of write requests (printing matching lines) and read requests (read further into file to search).

References and local variables created through `let` are immutable by default, which helps eliminate unwanted tampering. Most of our strings should be immutable since we are simply searching on them.

Pointer safety is easier to maintain, as Rust allows us to generate immutable "slices" that accesses parts of a longer string to search on. This is achieved by the ownership and borrowing rules Rust enforces, which states that only one valid access point to the heap can exist at each moment, and we can create either one mutable reference to this access pointer or multiple immutable references.

Rust implements a null pointer via states similar to ml's None and Some. These are handled by pattern matching, and throws an error when attempting to dereference None.

The compiler will check if the string is modified such that a "slice" will access invalid memory and throws a compile error to warn us early, which helps avoid bugs about dangling pointers and double freeing.

Rust also supports "unsafe" operations wrapped in specially designated blocks or functions. This can be used to implement C-like behavior should it prove to be necessary or desirable.

4 Rust's Performance and Reliability

Generally, Rust is a performant language. On most benchmarks, it performs remarkably fast, although on occasion slightly slower than C / C++. See references for these speed comparisons.

Rust checks types statically and manages unused storage by inserting `drop()` calls, which are similar to `free()` calls into the program whenever the region of memory's owner variable (think pointer) goes out of scope. It does not collect garbage at runtime. Therefore, it is more performant than other languages that need to check for garbage collection at runtime.

Because memory is freed according formalized rules that the compiler adopts, the program behaves reliably. We know when a region of memory becomes not used. Expensive operations like deep copying is performed via explicit calls to `clone()`, which informs the programmer that something special is taking place.

For our application, file reading is likely to be a major bottleneck for latency. We will want to read in log files from the system, and break them down into multiple blocks to hand off to each thread for searching on. Therefore we want to read in large blocks of data at a time (unless the file is small). We are looking at using BufReader for buffered reads.

[USER]:s on the Rust Forum achieved around 400MiB / second with their machines. Rate on our machines will depend on hardware architecture and storage solutions. Using Rust is a good choice for performance here, but not by much compared to other languages.

Thread creation in rust is relatively lightweight, as calling `std::thread::spawn` just creates a system native thread. This achieves similar performance to C / C++, and results in much more performant applications than languages like Java or Python. This also gives us the opportunity to spawn multiple threads to analyze the same line if the line is very long.

5 Generality and Flexibility of Rust Programs

Rust allows the user to specify types like `f32, u64`, directly specifying the number of bits they take up on the architecture. This can be very nice when we attempt to deploy code on to different architectures.

Rust is a compiled language that supports many plaforms, similar to C and C++.

One can create modules and generate modular code in similar fashion to C / C++, making the addition of new functionality easier. Rust additionally supports some functional programming-inspired features like closures, i.e. storing unnamed functions in variables and passing them around; and pattern-matching flow control; in Rust spirit, a pattern-matching flow control block must account for all possible inputs.

Rust is a more flexible language than C / C++ in the sense that it allows our team to operate at different levels of abstraction and use a multitude of programming paradigms. There is support for object-oriented programming and nicely wrapped classes prewritten in STL like BufReader and Mutex, but we can also implement our own versions of these objects from scratch if we choose to. For parallel computing, there are low and high levels of primitives as well, allowing our team to choose a specific strategy that targets our application.

6 Summary

In summary, Rust offers several advantages in our parallelization effort on the search utility. The language is more flexible and memory-safe than C / C++, offering more complete solutions that come with the language. Additionally, because of its static type checking and lightweight mechanisms (at runtime) for reclaiming unused memory (compiler code insertion) and creating threads (light overhead), it is on par in performance with C / C++ and significantly faster than Python or Java.

On the other hand, Rust's unique focus on memory safety results in restrictive syntaxes that, while helping teams catch bugs at compile time, nevertheless creates cryptic code and has a steep learning curve.

Since we are creating an internal tool for searching log files, having memory management bugs isn't too bad unless it impacts performance significantly. If may be optimal to stick with C/C++ and save on development time. There are considerations both for and against using Rust in our multithreaded file search.

7 Sources Cited

Benchmarks Game, The Computer Language 23.03 Benchmarks Game (2023).

Github, ripgrep. Accessed March 6 2024.

Programming Language and Compiler Benchmarks. All mandelbrot problem benchmarks (2024).

The Rust Programming Language. The Rust Programming Language (2024).

The Rust Programming Language. The Rustc Book (2024).

Nicholas Nethercote. The Rust Performance Book (2024).

Rust Programming Language users Forum. How to do Fast File IO? (2016).

ChatGPT 3.5. ChatGPT (2024).

8 Appendix: ChatGPT prompts

[USER]:

The programming language Rust has the following feature: dynamically allocated memory is automatically returned once the variable that owns it goes out of scope. How does rust achieve this? does the compiler insert code that frees memory, or does this get determined at runtime?

[CHAT GPT]:

Rust achieves automatic memory management through a mechanism called ownership and borrowing, combined with its ownership system and lifetime checks at compile-time. It does not rely on garbage collection or runtime checks to reclaim memory.

When you create a variable in Rust, it becomes the owner of the memory it refers to. When that variable goes out of scope, Rust automatically calls a special function called drop on that variable. This drop function deallocates the memory associated with the variable. The key point here is that this deallocation happens at compile time and is determined by the ownership and borrowing rules enforced by the compiler.

The ownership system in Rust ensures that each piece of memory has exactly one owner at any given time. When a variable is reassigned or goes out of scope, Rust automatically checks and enforces these ownership rules. If there are no more references to a particular piece of memory, Rust knows it's safe to deallocate that memory.

So, in summary, Rust achieves automatic memory management through compile-time checks enforced by the ownership system, rather than relying on runtime mechanisms like garbage collection. This results in predictable memory usage and performance characteristics without the overhead of runtime checks.

[USER]:

so drop() is essentially inserted into the source code by the compiler?

[CHAT GPT]:

Yes, that's correct. The drop() function is not explicitly written by the programmer in Rust code. Instead, it is automatically generated by the compiler for types that implement the Drop trait. When a variable goes out of scope, the compiler inserts a call to the drop() function for that variable. This function is responsible for performing any necessary cleanup, such as deallocating memory or releasing other resources associated with the variable.

The Drop trait allows customizing the behavior of the drop() function for user-defined types. By implementing this trait, a programmer can define specific actions that should be taken when an instance of their type is dropped, providing flexibility and control over resource management in Rust programs.

[USER]:

my cpu has multiple cores. Generally speaking, can each core access flash storage concurrently, or is there a limited number of channels for CPUs to talk to flash?

[CHAT GPT]:

Generally speaking, multiple cores in a CPU can access flash storage concurrently, but the concurrency might be limited by the underlying storage architecture and the specific flash storage device.

Flash storage devices, such as SSDs (Solid State Drives), typically have multiple NAND flash memory chips that can be accessed in parallel. These chips are organized into blocks and pages, and the storage controller in the SSD manages access to these blocks and pages.

While modern SSDs can handle concurrent access from multiple CPU cores, there are still limitations to consider:

Interface Bandwidth: The interface between the CPU and the flash storage device (e.g., SATA, PCIe) may have limited bandwidth, which could become a bottleneck if multiple cores are trying to access the storage simultaneously.

Controller Efficiency: The SSD's controller plays a crucial role in managing concurrent access from multiple cores. Some SSD controllers may be more efficient than others in handling concurrent access patterns, resulting in better performance.

I/O Queuing: SSDs often have internal queues to manage incoming I/O requests. These queues can help optimize performance by reordering and scheduling requests from multiple cores.

Parallelism within the Flash Device: Flash memory chips themselves can support internal parallelism, allowing multiple read or write operations to be executed concurrently within the chip.

Overall, while multiple CPU cores can theoretically access flash storage concurrently, the actual level of concurrency and performance may vary depending on factors such as the SSD's architecture, interface bandwidth, controller efficiency, and the specific workload characteristics. Additionally, software optimizations, such as proper I/O scheduling and parallelism management, can further improve the efficiency of concurrent access to flash storage.

[USER]:

is thread creation lightweight in rust?

[CHAT GPT]:

Yes, thread creation is relatively lightweight in Rust compared to some other programming languages. Rust's standard library provides a std::thread::spawn function for creating new threads, and this function is designed to be efficient and lightweight.

When you call std::thread::spawn, Rust's runtime system (backed by the operating system) creates a new native thread and associates it with a lightweight Rust thread object. This operation typically involves minimal overhead compared to creating threads in languages like Java or Python, which may incur more significant overhead due to the runtime environment or garbage collection mechanisms.

Rust's lightweight threading model allows you to create and manage thousands of threads with minimal performance impact. However, it's essential to consider the potential overhead of context switching and synchronization when designing multithreaded applications. Rust provides synchronization primitives like Mutex, Arc, and Atomic types to facilitate safe concurrent access to shared data.

Overall, while thread creation in Rust is lightweight, it's essential to use threading judiciously and consider factors such as scalability, synchronization overhead, and potential contention when designing concurrent applications.

[USER]:

are semaphores implemented in the C++ std?

[CHAT GPT]:

As of my last update in January 2022, the C++ standard library (C++ STL) does not include a native semaphore implementation. However, C++20 introduced the semaphore header, which provides standard facilities for semaphore-like synchronization primitives.

These semaphore classes provide a modern and standardized way to perform synchronization in C++ programs. However, it's important to note that support for <semaphore> may vary across different compilers and library implementations, especially for older compiler versions. Additionally, as of my last update, not all compilers fully support the C++20 standard, so it's essential to check the compiler documentation for compatibility and feature availability.

[USER]:

what happens if the file my rust program is reading gets modified by some other process?

[CHAT GPT]::

If a file being read by a Rust program gets modified by another process while the Rust program is actively reading it, the behavior can vary depending on several factors:

Operating System Behavior: Different operating systems may handle file modifications differently. Some operating systems may allow a file to be read even if it's being modified concurrently, while others may prevent reading or provide inconsistent results.

Filesystem Behavior: The behavior can also depend on the filesystem used. Some filesystems support concurrent reads and writes to files, while others may have stricter consistency guarantees that prevent concurrent access.

Rust's File I/O APIs: Rust's file I/O APIs generally provide low-level access to files, and they rely on the underlying operating system for file management. If the operating system allows concurrent access to the file, the Rust program may continue reading from the file even if it's being modified concurrently.

Buffering and Caching: If the Rust program is using buffered or cached I/O, it's possible that the program may read stale or outdated data from the file's buffer or cache, especially if the file is being modified concurrently.

In general, reading a file that is being modified concurrently by another process can lead to unpredictable behavior and potential data corruption or inconsistencies. To avoid such issues, it's recommended to use appropriate synchronization mechanisms (e.g., file locks or mutexes) to coordinate access to the file between different processes or threads.