Software Systems - Assignment 01 Group 19 - Concurrency

Zhengtao Huang (5833469, zhengtaohuang) Barry Tee Wei Cong (5662834, btee)

1 Introduction

The purpose of this task is to create a concurrent variant of the grep command. Grep is a utility that permits the search for lines matching a specific pattern in plain text datasets. The implementation of the program must adhere to a predetermined structure and is not permitted to utilize any external libraries apart from those provided. The two topics chosen for discussion are Channels and Rayon. We have implemented two versions of multi-threaded grep, one with Rayon and one without. Our benchmark results have shown that both versions of the multi-threaded grep are faster than GNU grep, which is further discussed in Section 5.

In our code, there is a constant "GREP_CHOICE" in main.rs, which is used to choose between the two versions of implementations.

2 Experimental Setup

We utilised one of our laptop to perform Grep Experiments. The HP ProBook 440 G7 laptop has an 10th Gen Intel(R) Core i7-10510U CPU. The specifications i.e. cores, threads and Random-Access Memory (RAM) of both laptops have been listed in Table 1.

HP ProBook 440 G7	
Physical Cores	4
Logical Cores	8
Total RAM	16 GB
Operating System	Fedora Linux 39

Table 1: Laptop Specifications

3 Channels

3.1 Working of Channels

The module 'std::sync::mpsc' in Rust uses a "multiple-producer, single-consumer queue" data structure. To prevent data race by the multiple senders, the channel uses a lock to synchronize access i.e. only one thread can modify the channel at a time. The bounded queue implies that the channel size is limited and that the sender can block until channel space is available. Messages are sent from sender to receiver without copying the data via RUST's ownership and move semantics to avoid unnecessary memory allocations.

3.2 Various Variants of Channels

Rust provides two main channel variants: bounded and unbounded. As mentioned, bounded refers to a specified capacity of messages in the channel also referred to as a buffer size. The function used is 'std::sync::mpsc::sync_channel()'.

In the case of unbounded channels, the channel is allowed to grow dynamically using the function 'std::sync::mpsc::channel()' without a specific buffer size. The key difference can be observed when the sender is faster than the receiver. The unbounded channel can grow indeterminately, that causes expanded memory growth.

The advantages of using bounded channels are prevention of resource exhaustion, i.e., memory or processing time, and feedback pressure applied to sender via blocking. The disadvantages are potential deadlock and inadvertent drop or delay in messages.

The advantages of using unbounded channels are the ability to handle unpredictable workload capacities and no block restrictions on messages sent. The disadvantages are the potential resource exhaustion and the lack of feedback pressure safety mechanisms. One consideration to use unbounded channels are in event-based architectures, and predictability of messages being sent is closed to none. An example can be a simple Graphic User Interface (GUI) application using the 'druid' crate. When the user clicks on a front-end screen, the user's clicks are deemed asynchronous and hence in the back-end, messages are sent to an unbounded channel. It is imaginable if the user gets frustrated when his clicks are blocked and causes an unresponsive response.

In the interest of knowledge, the Tokio library can be considered an upgrade for async channels, as they are built to integrate with the Tokio runtime while supporting feedback pressure mechanisms. This means that when the sender is faster than the receiver, methods such as send and try_send provide critical information on the success of the send task, thus providing a control feature to tune the buffer size. However, Tokio declares that it suits interleaving tasks while waiting for I/O operations and is a bad fit for an application that runs purely on computations in parallel. Therefore, Tokio recommends the use of Rayon. Another disadvantage of Tokio is that it provides no additional benefit for projects with many files to be read.

3.3 Advantage over Mutex

In essence, the key takeaway from studying Channels is to enable concurrency i.e., safe data sharing between multiple senders and single receiver whereas a Mutex is a fundamental synchronization processing element that limits one thread to access shared data and prevents concurrent access***. Mutex are more prone to deadlocks, and careful management is hard to program, not forgetting the need for performance optimization. Channel, on the other hand, circumvents deadlocks via implicit thread synchronisation via usage of the library.

*** Mutex advantages over Channels are not mentioned here as it is not a requirement.

4 Rayon vs OpenMP

4.1 How Rayon Schedule Tasks

Rayon is a parallelism library found in Rust that is based on a work-stealing algorithm to schedule tasks, i.e., idle threads steal tasks from other busy threads, ensuring constant usage of available resources. When a large task is submitted to the Rayon scheduler, Rayon divides them into smaller sub-tasks forming a binary tree structure and send sub-tasks to available threads for parallel execution. If a thread finishes its task, it can steal tasks from the queue, which were meant for other busy threads. The main goal is to reduce idle time. The process can be divided into smaller tasks, and threads adapt to the dynamic number of small tasks, creating an efficient workload operation.

4.2 How Rayon is Different to OpenMP

Rayon ensures tasks are dynamically distributed among worker threads, which is more efficient in balancing the load compared to OpenMP using static distribution where tasks is split into fixed-sized chunks. OpenMP operation results in sub-optimal load balancing. Another key difference lies in Rayon is written for Rust programming language by using parallel iterators, but OpenMP is designed for C which was the programming language used in Advanced Computing Systems (ACS) Course.

4.3 Experience of using Rayon vs OpenMP

Having used OpenMP in C, it is much easier to parallelise sequential iterations by simply importing and changing iter to par_iter without much modification of the code. This usage of iter() and par_iter() in Rust is intuitive and quick to convert iterations to parallel iterations within preexisting code. However, for OpenMP in C, #pragma omp parallel for is a standalone directive that tells the OpenMP compiler to distribute the iterations of the loop across the available threads in the thread-pool and does not include the additional work to refactor the code to enable parallelism.

Furthermore, Rust prevents unsafe threads unlike C only reveals unsafe code during run-time with data-race and deadlocks, which is why OpenMP is not widely utilized and scaled across various applications. So, this also means that Rayon and Rust will be more favoured for embedded systems software development compared to OpenMP and C for upcoming new projects back in my company.

4.4 Rayon's Advantage Over Custom Thread-Based Approach

Rayon is an easy-to-use high-level Application Programming Interface (API), that eliminates the need for low-level thread management, making it simple to introduce parallelism via work-stealing into most code whereas custom thread-based management requires additional manual code to create, synchronize, and manage threads, which is challenging and error prone i.e., data-race.

5 Performance - Results

This section of the report shows the actual implementation of what was learnt for Channels and Rayon. The first step was to use "cargo build –release" that puts the resulting binary in the directory "./target/release", and the code will run faster compared to compiling in the default debug mode. To obtain the run-time of the Grep tools, the "time" command was used. In the benchmarks, the execution times of searching for all matches of the regex "[mM]icrodevice" in the Linux source code were recorded. The lesser time taken means faster performance in the run-time of the code.

In Table 2, we measured and recorded the run-time of the GNU grep command from Linux. Following that, a test was done on our naive implementation of the self-implemented multi-threaded mygrep tool. Following which, the team decided to take on the bonus challenged of implementing the mygrep tool in Rayon. Table 2 shows the average of 10 results for the 3 different methods Grep tools, one Linux grep -r and the other two that were implemented based on the assignment's requirements. From the results shown, custom thread-based mygrep is faster than Linux grep -r, whereas the Rayon-based mygrep is the fastest of them all.

Searching in 1 Copy of Linux Source Code			
	grep -r	mygrep(Base)	mygrep(Rayon)
1	1.610	1.024	0.514
2	1.606	1.091	0.470
3	1.622	1.094	0.471
4	1.602	0.943	0.449
5	1.625	0.905	0.472
6	1.575	0.971	0.464
7	1.603	1.017	0.474
8	1.597	1.021	0.472
9	1.591	1.116	0.474
10	1.588	1.010	0.461
Avg	1.602	1.019	0.472

Table 2: Benchmark Using 1 Copy of Linux Source Code (Execution Time in Seconds)

In Table 3, the same test was performed, but instead of running it on one copy of the Linux source code, the team decided to test it on ten copies of Linux source code. The results in Table 3 also showed the same ranking where Rayon-based mygrep is faster than self-implemented multithreaded mygrep, and the GNU grep is the slowest.

We believe the reason why the GNU grep is slower than the two multi-threaded version is that the use of multi-threading parallelises the process of searching in the file system and finding matches in the files. However, reading from files is an IO bound operation. Although the use of multi-threading can help to saturate the disk bandwidth and to accelerate the process of traversing the directory tree, there should be an upper limit of how much we can improve Grep using the

Searching in 10x Linux Source Code			
	grep -r	mygrep(Base)	mygrep(Rayon)
1	76.116	21.757	17.906
2	74.387	21.042	18.420
3	74.691	23.181	19.716
4	74.709	23.200	19.039
5	74.529	21.744	18.726
6	74.700	23.374	19.864
7	75.049	22.110	19.663
8	74.868	22.640	19.805
9	74.553	22.053	17.787
10	74.952	22.960	18.723
Avg	74.855	22.406	18.965

Table 3: Benchmark Using 10 Copies of Linux Source Code (Execution Time in Seconds)

technique of parallelisation.

As for the difference in performance found in the two multi-threaded versions, it can be due to the fact that the Rayon-based version adopts a better thread management strategy. The Rayon-based implementation utilises the work-stealing strategy while keeping the total number of threads unchanged. In our base implementation, we keep track of the number of active threads, and a thread exits when it finishes its job. When a thread exits, a new thread with a new job will be spawned. Comparing to the the Rayon-based implementation, our base implementation does not reuse threads and brings overheads when a new thread replaces an old one.

6 Conclusion

In conclusion, a thorough study of concurrency was done, mainly in the research of Channels and the Rayon library. The base implementation of multi-threaded MyGrep without Rayon is faster than the GNU Grep tool in Linux, and the implementation of MyGrep using Rayon is the fastest. The speedup based on the code written by the team is 1.572x faster for the self-implemented multi-threaded MyGrep compared to GNU Grep during the search in 1 Linux_Code File. The speedup is even more significant for 10 Linux_Code Files - 3.351x faster for the self-implemented MyGrep and 3.947x for Rayon-based MyGrep when compared to GNU Grep tool.

References

[1] Rust
https : *//doc.rust* − *lang.org/std/sync/mpsc/index.html*
"std::sync::mpsc - Rust," (accessed Nov. 17, 2023)

[2] Rust
Rust − *lang.org*, 2019.*https* : *//doc.rust* − *lang.org/book/ch*16 − 00 − *concurrency.html*
'Fearless Concurrency: The Rust Programming Language' (accessed November 17, 2023)

[3] A. Nadiger
https : *//www.linkedin.com/pulse/channels* − *rust* − *amit* − *nadiger/*
"Channels in Rust," www.linkedin.com, May 26, 2023. (accessed November 17, 2023).

[4] Tokio
https : *//tokio.rs/tokio/tutorial*
"Tutorial — Tokio - An asynchronous Rust runtime," tokio.rs. (accessed November 17, 2023).

[5] Tokio
https : *//users.rust* − *lang.org/t/channel* − *or* − *mutex*/18106/2
'Channel or mutex', The Rust Programming Language Forum, June 15, 2018. (accessed November 17, 2023)

[6] N. Matsakis
https : *//docs.rs/rayon/latest/rayon/*
'Rayon - Rust', docs.rs, 06 September 2019. (accessed Nov. 17, 2023)

[7] S. T. R.
https : *//github.com/trsupradeep/*15618 − *project*
"Comparison of Multi-threading between C++ and Rust (OpenMP vs Rayon)," GitHub, Nov. 04, 2023. (accessed Nov. 17, 2023).

[8] K. Chauhan
https : *//kartik* − *chauhan.medium.com/parallel* − *processing* − *in* − *rust* − *d*8*a*7*f*4*a*6*e*32*f*
'Parallel Processing in Rust', Medium, 23 Jan. 2023. (accessed Nov. 17, 2023).