Introduction

The language we decided to use to implement parallelism for matrix multiplication was Rust. Rust is a low-level functional language whose main selling point is its primary focus being both speed and efficiency. It's also a robust programming language to implement parallelism as it provides functionality to allow concurrent logic to be safe. One reason we picked this language was due to the fact that it offers safety when writing parallel logic. Unlike Rust, C's parallel libraries require the user for the most part to take care of making sure they handle things like data race conditions. However, the design of Rust aims to eliminate these concerns through its code paradigm.

Syntax

Rust on the surface looks much like C, and in fact derives much of its syntax from C++. Once you start digging down deeper into the language, however, the differences become much clearer. Variables in Rust are immutable by default, but the `mut` keyword converts a variable to be mutable. Defining data types works similarly, but the syntax differs. To define the type of a variable after adding a semicolon to the variable name, you identify the type. For example 'f' stands for float and 'i' stands for integer. However you can also define the size by appending a number. For example to make a float of 32 bits, you write f32 and likewise for an integer. It tries to be more human-readable, so there is a reduction in symbol usage throughout the syntax. If statements are commonly written without parentheses, for example. If the final statement in a block lacks a semicolon, it is considered to be a return statement. It uses more words, for example the "as [type]" syntax for specifying data type.

Comparison

Rust in comparison to other parallel libraries we used like OpenMP and POSIX from using it seems to be more powerful. This is because Rust as a language is designed to be able to achieve parallelism, therefore the functionality that supports parallelism is more robust. Rust offers many ways to achieve parallelism. However with Rust, one of its main selling points is safety, so there are a lot of restrictions when trying to implement parallelism.

One way is by spawning threads. This is much similar to the traditional way to support parallelism but because safety is a priority in Rust, code won't compile unless the user abides by the restriction Rust sets in place to ensure that when your code runs, that it will work as intended.

Discussion

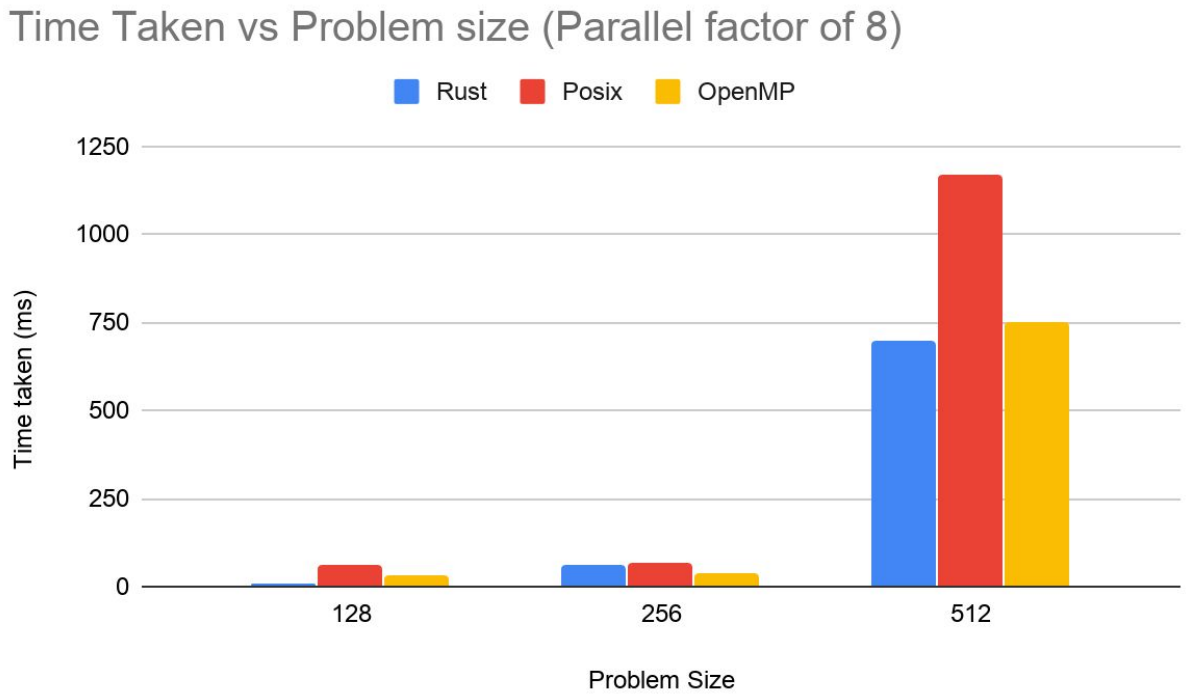
Rust is a very robust language for achieving parallelism. However Nik found using Rust with the use case of solving matrix multiplication to be quite difficult. This is because the matrix multiplication problem using parallelism requires updating a shared vector with computations involving other vectors. Implementing a serial approach was trivial - it was just a matter of getting the syntax down. However he started reaching problems when trying to implement the solution in parallel. In short, implementing the code provided in mmult.c in rust in parallel just using the standard library is impossible. This was concluded after much trial and error of trying multiple different solutions. Nik was able to use threads (which were in the standard library), however because of the restrictions Rust imposes for concurrent threads running, the code he

wrote with threads just ended up being a serial execution which provided even worse performance than a single threaded approach (due to overhead of spawning and joining threads).

On the other hand, Nik really enjoyed using OpenMP as he thought it was very intuitive and did not require writing too much additional code to achieve parallelism. However with enough practice with Rust he could actually see myself enjoying using Rust as the language of my choice when writing parallel programs. This is because Rust includes a lot of functions that can be applied to data types (like vectors) that automatically execute the logic in parallel (e.g. par_chunks_mut). In terms of overall performance Rust is very comparable to C. Rust is a lower level language so it allows you to define memory allocation (to an extent). However the limitations Rust imposes on the user programming the language is a result of how Rust abstracts data to help itself achieve parallelism. While it does allow for a good amount of memory allocation, Rust does not have much freedom as C to allocate memory. However when talking about performance in terms of safety, this means Rust has a clear advantage over C. Rust is a more safe language than C. This is because if programmed the “Rust" way, there is both type safety and memory safety. The Rust compiler is very smart at catching out logic that compromise the safety of the program and even though compiling errors become more frequent, it ensures that when your code runs, it will run as the programmer intended. However because of this, parallel programs can be more difficult to write. This all depends on the use case however. Writing logic that applies a function to every element in a vector, or writing reduce functions over a vector is very easy and the built in functions can distribute the work for you. However if the parallel logic you need to write is less straight forward (e.g. for matrix multiplication) where you need to update a single shared variable and require indexing of other,

out of scope vectors, then parallelism can be much more difficult. This once again has to do with Rust making sure that the parallel code you write does not have any data conditions and that synchronization of resources is safe.

Above is a graph indicating the performance of parallelism of Rust in matrix multiplication. The different parrelization factors are 1, 8, and 16 and while there isn't a huge difference in performance between a parrelization of 8 and 16, there is significant improvement over a single threaded application. The gap grows with the problem size, so even larger input would yield in more performance gains.



The above graph compares Rust in the context of C. It is noted that Rust outperforms Posix and OpenMP approaches across the board. This is not surprising because Rust makes it really easy to write efficient code. Because of the smartness of the compiler, compiled Rust code makes it really easy for Rust to know which memory to allocate and free up, as well as how to

synchronize data across a distributed workflow. While in C this is possible too, it is up to the user to optimize his/her code and the failure to do this could lead to slow/faulty/error-prone code. So one possibility why Rust is outperforming my implementations of POSIX and openMP could be due to the fact that my C programs weren't written to be as optimal as possible. However part of the reason for suboptimal code is due to the difficulty there is to write fully optimal C code was the compiler doesn't do much to look out for suboptimal code. This however wasn't the case in the beginning. When first testing the Rust code, the performance was drastically worse than even the C non parallel approach. While according to the graph, the time it takes for matrix multiplication to execute is a little over 1 second (>1000ms), the first trials the time taken was around 26 seconds. However after inspecting this terrible performance, it was discovered that a non-optimized version of the project was being run. This is because to build and run the Rust project, *cargo build* & *cargo run* were being run. However by including the *--release* flag, Rust optimizes the code when building the project and will use the optimized build when running the project as well.

While Nik thinks this language is a really good language to learn for writing parallel programs, he believes that there are other frameworks that should be taught in class instead (along with C). This is because Rust overall is pretty similar to C other than syntax and its ability to ensure safety. Good technologies to teach that are very popular and borrow from similar ideas but are also in its completely own class are Hadoop/Spark. This is because these are technologies that are supposed to run distributed on nodes rather than processes/threads. Rust like C uses threads as an option to achieve parallelism so it wouldn't be worthwhile to teach in his opinion.

Alex had a slightly different reaction to Rust. While its syntax was different from C, he felt that it adopted some of the better qualities from other recent languages. Like many more modern tools, Rust is very closely tied to rustup and crate, in the same way that PHP has composer or (and this is perhaps the stronger comparison) NodeJS is tied to npm. The tight integration makes it more difficult to write simple, unintegrated code and compile it, but for someone who might actively develop in Rust over a longer period in time, Alex thinks this would be a significant advantage.

Under the hood, Rust brings support for other ideas that were new or that hadn't been developed in the early days of C. As a functional language it encourages immutability of data, of course, and reduction of side effects. It specifies ownership of data--the scope of a value is the same as the scope of its owner, which provides clarity for the lifetime of an object and gives the compiler tools for memory-safing code.

Alex has had some good and bad experiences trying to adopt toolchains over the years (PostCSS comes to mind as one of the bad experiences) but Rust's toolchain was amazingly easy to install. Granted, this is on a *nix box (specifically, Ubuntu 19.10) but installation was performed entirely by running:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

(with the exception of temporarily adding the cargo path to the environment variable)

Similarly, the language does have some divergence from C-like syntax, but the syntactic drift is minor. The benefits were not--compared to C, where simply implementing parallelism required multiple, sometimes dozens, lines of setup, Rust's built-in support was excellent and the the

crates (packages, really) available through Cargo made things even easier. Alex was able to write the entire parallel portion of the Reimann sum in what was essentially one line--he expanded the closure to make it readable, but if the other people reading the code could be expected to be professional Rust programmers, there wouldn't be any reason for the expansion.

That simplicity argues against Rust as a teaching tool, however. Alex had similar feelings to Nik in terms of his recommendation, although for somewhat different reasons. He grew up learning low-level languages--BASIC, C, C++--and only came to Java later in life, to find that basic concepts like pointers were mysteries to more contemporary students. While he recognizes that pain shouldn't be carried from one generation to the next, there's value in learning the details of why a programmer might take one approach over the other--or why one implementation might take that approach over a competitor. Rust successfully takes the pain out of concurrency, but in so doing it also obfuscates many of the concepts that the class is attempting to teach. Rather than focus on Rust, Alex agrees with Nik's recommendation that other tools might bring more to the table.

Conclusion

Rust is a very useful language to use when writing parallel programs. While some use cases (as with matrix multiplication) seem to challenge the ease at which a parallel program can be written in Rust, the language offers many functions/libraries that make implementing parallelism a breeze. However I learned that in order to really be able to masterfully program in Rust, a solid understanding of its fundamentals needs to be understood, fundamentals that do not exist in C.

Because of this there is an initial steep learning curve when using Rust. However as the analysis shows, being able to become adept at Rust will lead to very efficient code.