SCC Coloring Algorithm - C vs Rust

Ioannis Stefanidis - 9587

Aristotel University of Thessaloniki

28 April 2023

In this paper I'm going to compare the **performance of the Coloring Algorithm** for finding the Strongly Connected Components (SCC) in a directed graph, written in **C and Rust**.

Introduction

In a few words, the coloring algorithm for finding strongly connected components (SCC) is a depth-first search-based algorithm that assigns unique colors to vertices during the search, and then identifies SCCs based on the colors assigned.

The data structure of the graphs and the method of parallelization is discussed in a previous paper for this assignment *SCC Coloring Algorithm in parallel with C*[1] . In this paper I'm going to compare the fastest implementation of the C code (OpenCilk) with both safe and unsafe Rust code.

Parallelism in Rust

Rust has a very good support for parallelism. It has a built-in library for parallelism called `std::thread` and a third party library for parallel iterators called `rayon`. In this paper I'm using rayon which uses a work-stealing scheduler similar to OpenCilk.

Why use unsafe Rust?

In the C implementation, there are some benign race conditions that do not affect the result of the algorithm, but are necessary for its speed. These race conditions were discussed in the *Coloring* section of the original algorithm paper.[2]

In Rust, the borrow checker prevents mutably accessing the same data across threads without synchronization mechanisms such as locks or atomics. To ensure a fair comparison between the C and Rust implementations, I first implemented the algorithm using only safe Rust and then added unsafe blocks where necessary for performance reasons.

---

[1] Stefanidis Ioannis, *SCC Coloring Algorithm in parallel with C* (2022). link: https://github.com/johnstef99/scc_madness/blob/master/report/scc_report.pdf.

[2] Slota, George M, Rajamanickam, Sivasankaran, and Madduri, Kamesh, *BFS and coloring-based parallel algorithms for strongly connected components and related problems,* IEEE (2014).

Results

At the following table we can see all the graphs that were used for testing and the number of vertices, edges and SCC for each one.

Graph's Name	Vertices	Edges	SCC
celegansneural	297	2345	57
foldoc	13356	120238	71
language	399130	1216334	2456
eu-2005	862664	19235140	90768
wiki-topcats	1791489	28511807	1
sx-stackoverflow	2601977	36233450	953658
wikipedia-20060925	2983494	37269096	975731
wikipedia-20061104	3148440	39383235	1040035
wikipedia-20070206	3566907	45030389	1203340
wb-edu	9845725	57156537	4269022
indochina-2004	7414866	194109311	1749052
uk-2002	18520486	298113762	3887634
arabic-2005	22744080	639999458	4000414
uk-2005	39459925	936364282	5811041

Figure 1: Table of graphs used for testing (sorted by the number of edges).

Now let's take a look for each implementation of the coloring algorithm how much time was needed to find all the Strongly Connected Components:

Graph's Name	OpenCilk	Safe Rust	Unsafe Rust
celegansneural	1.46	2.55	1.39
foldoc	2.73	7.16	2.58
language	29.37	93.56	44.02
eu-2005	192.66	483.15	208.97
wiki-topcats	1601.57	5.73	1993.44
sx-stackoverflow	654.95	938.18	627.43
wikipedia-20060925	2674.3	37205.44	3584.63
wikipedia-20061104	2038.97	42293.73	2558.36
wikipedia-20070206	2314.81	63242.29	2650.71
wb-edu	19273.45	37350.83	16305.11
indochina-2004	8448.95	27135.79	11044.77
uk-2002	20127.72	29735.58	13231.55
arabic-2005	17951.26	30642.27	13168.97
uk-2005	62696.62	123428.41	51449.39

Figure 2: Table of execution time in milliseconds of each implementation for each graph (machine used: Macbook Pro M1 16GB 1TB).

In Fig.3 the graph shows the relative execution time of each implementation (relative to the slowest one) for each graph. We can see that the slowest implementation is always the safe Rust one and C and unsafe Rust are competing very closely, with both of them in average finish in 1/3 of the time that safe Rust needs.

In graphs like the wikipedia-20060925, wikipedia-20061104 and wikipedia-20070206 the safe Rust implementation is 10+ times slower than the other two. This implies that a lot of coloring cycles are happening in these graphs and the safe Rust implementation is paying the cost of the atomic operations and the cloning of some vectors to avoid race conditions that are happening.

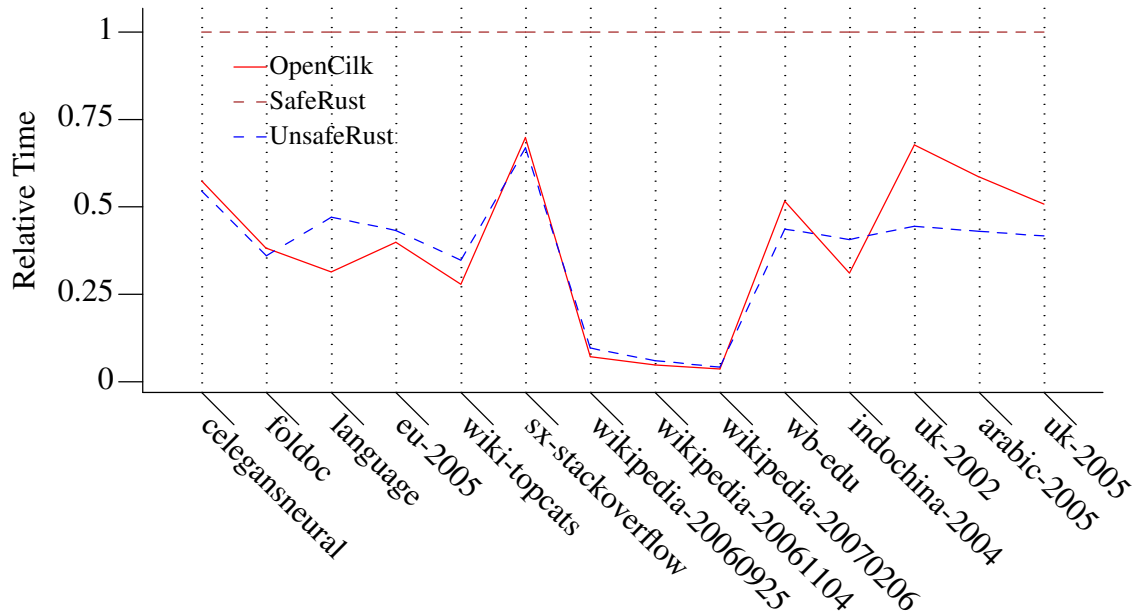

Figure 3: Relative time graph for the parallel implementations.

Conclusion

I really wasn't expecting Rust to perform so close to the OpenCilk implementation. Writing Rust takes more time than writing C, especially when you need to do something unsafe. But knowing that your code is safe anywhere else except inside the unsafe blocks is a great feeling. Having all that safety without sacrificing any performance is even better.

The source code for this assignment is available at the rust branch in this repo:
https://github.com/johnstef99/scc_madness