SCC Coloring Algorithm in parallel with C

Ioannis Stefanidis - 9587

Aristotel University of Thessaloniki

29 November 2022

For this assignment we were asked to find all the **Strongly Connected Components (SCC) in a directed graph**. The goal was to implement the Coloring Algorithm, which can be executed in parallel. The algorithm was implemented in C with 3 different frameworks/libraries OpenCilk, OpenMP and pthread.

Data Structure

The graphs were provided to us by *SuiteSparse Matrix Collection*[1] in a sorted COO matrix format. Given the above, I stored the graph in CSC format (Compressed Sparse Column) by reading the file line by line. By doing that and not storing the Adjacency Matrix to create the CSC or the CSR matrices I managed to read graphs with file-sizes much greater than my available RAM.

Having only the CSC left me with 2 options. The first one was to transpose the CSC to get the CSR, which I did and it worked well for graphs with a relatively small amount of edges. But doing that for graphs with 1.4B edges was too time-consuming and also meant that I would have to allocate 2 times the memory of the CSC. So I ended up with the second option which was to modify the coloring algorithm to only use the in-degree edges of a vertice.

For the graph representation (`struct Graph`) I decided to use the CSC matrix and 2 arrays with `number_of_vertices` length to keep track of the scc_id of each vertice and which vertices I have removed during the trimming and the coloring.

Parallelization

Trimming in parallel

To trim the graph having only the in-degree edges of each vertice meant that I had to go through each edge $(u, v) \in E' \forall\, u \in V$ and mark that vertice u has an in-degree edge and vertice v has an out-degree edge (E' is the set of incoming edges and V are all the graph's vertices). After doing that I had to go through each vertice and check if it has zero in or out edges to trim it. Both of these loops can be executed in parallel one after the other. Although at the first loop we can encounter a race condition in the case where there

[1] *SuiteSparse Matrix Collection.* link: https://suitesparse-collection-website.herokuapp.com/.

are edges (u_1, v_1), (u_2, v_1) which for both we are going to try to write to the v_1 position of `has_out[]` array, there isn't any real problem with that because the information that the vertice v_1 has an out-degree edge is going to be saved.

Coloring and BFS in parallel

The original coloring algorithm[2] that we were asked to implement uses the outgoing edges to determine if the color of a vertice should change. On the other hand, I only use the incoming edges, so the color of a vertice u can be determined by this function $color(u) = min(color(u), color(v)) \, \forall (u, v) \in E'$. This function should run in loop until there are not further change to colors, to attain that, I kept a boolean `color_changed` outside of the while loop. Propagating the colors in parallel results in a race condition on the `color_changed` boolean, but once again, there is no problem with that race because the loop will run again as intended, no matter which thread will turn the `color_changed` to true (for OpenCilk reducers had to be used to avoid *false sharing*).

Continuing with running the BFS in parallel for each unique color, we aren't going to encounter any race condition here. Because every BFS is going to have a different starting vertice u_{entry} and it's going to write at `removed[v]` $\forall v$: $color(v) == color(u_{entry})$.

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
twitter7	41652230	1468365182	8044728

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

[2] Slota, George M, Rajamanickam, Sivasankaran, and Madduri, Kamesh, *BFS and coloring-based parallel algorithms for strongly connected components and related problems,* pp. 550-559, IEEE (2014).

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	Serial	OpenCilk	OpenMP	Pthread
celegansneural	0.101	1.465	1.901	1.172
foldoc	2.034	2.735	3.125	3.408
language	74.904	29.379	35.710	34.884
eu-2005	456.182	192.661	222.757	221.364
wiki-topcats	5480.700	1601.571	1963.331	1913.419
sx-stackoverflow	1015.416	654.957	696.468	637.399
wikipedia-20060925	5409.858	2674.38	4023.632	4014.157
wikipedia-20061104	4307.067	2038.979	2887.391	2906.421
wikipedia-20070206	4312.090	2314.812	3053.161	3074.681
wb-edu	75944.155	19273.456	23687.500	21796.886
indochina-2004	25178.474	8448.959	11949.300	11550.546
uk-2002	56302.432	20127.728	21624.896	21219.984
arabic-2005	49314.916	17951.261	18427.179	18349.330
uk-2005	178898.744	62696.625	61866.468	69041.439
twitter7	538467.272	365122.715	414258.941	446038.842

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

In most of the cases the fastest implementation is **OpenCilk** (see figure 3) and that's reasonable for the following reasons. It's obvious that is going to be faster than the serial implementation because we get more workers doing the same amount of work, without any mutaxes, so no thread needs to wait for an other to finish. Why though OpenCilk runs faster than OpenMP and Pthread for the majority of the graphs?

To get the answer we need to understand how OpenMP and my Pthread implementation manage the distribution of work between their threads. In both implementations each time we need something to be done in parallel we spawn a new group of threads to divide the work W to W/P where P is the number of threads, we then wait for all the threads to finish (sync) and we exit the threads. That means in case of a for loop with V amount of iterations, each thread is going to run $T = V/P$ amount of iterations, so the first thread is going to run for $i \in [0: T-1]$ the second for $i \in [T: 2T-1]$ etc.. At this point the problem is clear, what happens if a thread finishes all it's work? Well it just going to do nothing until all the other threads are finished.

OpenCilk on the hand, uses what they call a *Work-Stealing Scheduler*.[3] What that does is, after the initial distribution of work to each thread, if a thread finishes all it's work it selects randomly an other thread and "steals" some work from it. So basically OpenCilk does a dynamic load balancing at runtime. That's why on the largest graph *twitter* OpenCilk can get ahead for a whole 1 minute.

[3] *Cilk's Work-Stealing Scheduler,* p. 13. link: https://www.opencilk.org/img/opencilk-pact-2021.pdf.

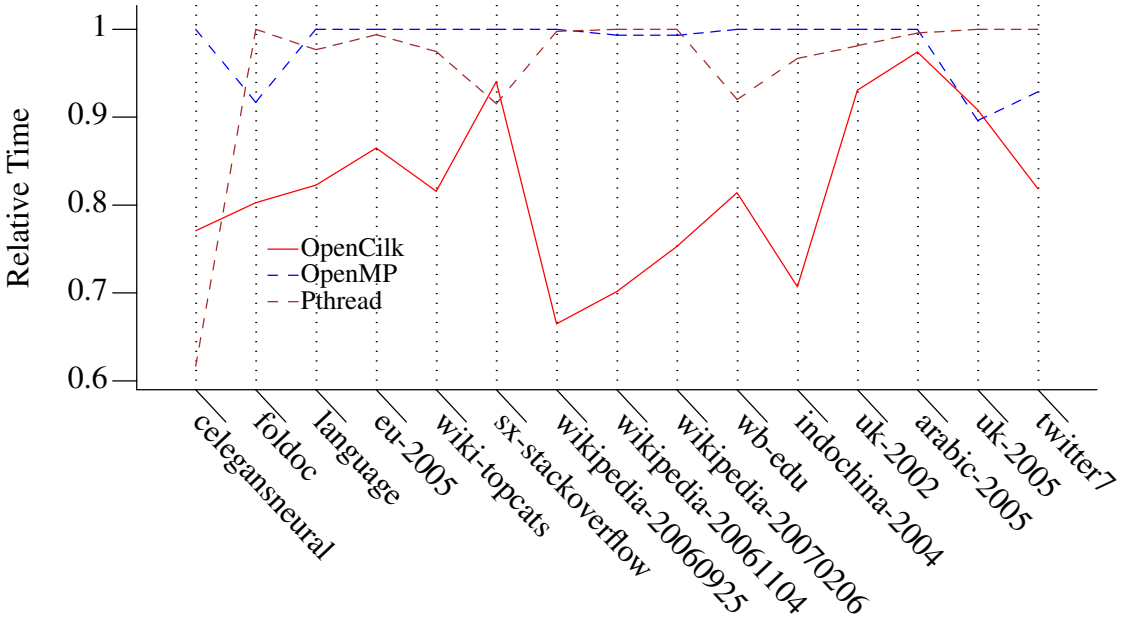

Figure 3: Relative time graph for the parallel implementations.

Finally something worth mentioning is that for small graphs like the first two, doing the extra work of creating threads and distributing work, get us worst performance than running it in a single thread. Or in the case of the graph *foldoc* we can see that there is an optimal number of cores (see figure 4).

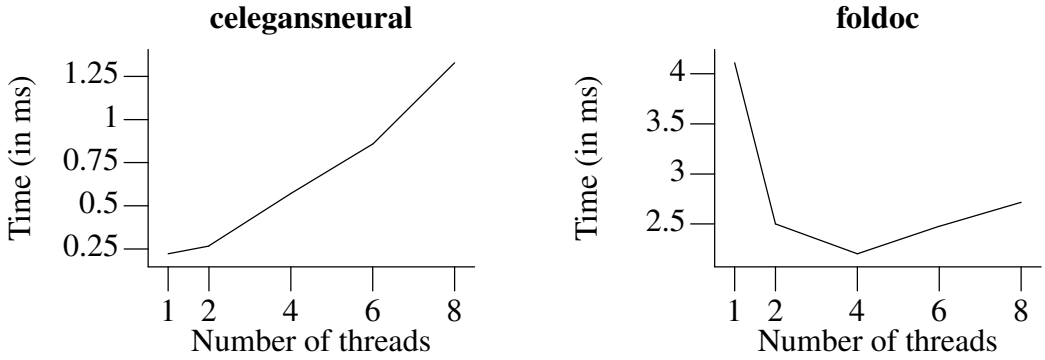

Figure 4: Execution time per number of threads used.

The source code for this assignment is available at:
https://github.com/johnstef99/scc_madness