Finding the Strongly Connected Components of a directed graph in a shared memory multiprocessor

A strongly connected component (SCC) is a maximal subgraph of a directed graph in which there is a path from each vertex to any other vertex in the subgraph. In this homework, I will be implementing a parallel strongly connected components algorithm to identify the number of SCCs in large directed graphs and compare its performance between 3 different Parallelization tools, Openmp, Opencilk an Pthreads.
The algorithm that I will be implementing is a coloring SCC algorithm that work in 2 stages:

- In the first stage we check the Node's parents, if there is a parent with smaller number than the Node's, then the current number of the Node changes to the smaller one. In this way we can see how far a root Node can "reach".

- In the second stage we start from the root Node and check if its parents have the same color (Number), and if they do, we insert them in the SCC and repeat the process. This way we can see what Nodes that the root Node can reach, reach back to it.

My Serial Implementation

The first thing before creating the algorithm was to read the matrices, and I did that using the available code in https://math.nist.gov/MatrixMarket/mmio-c.html . After reading the file I proceed to transform it into CSC & ROWS format. Next step is creating the algorithm.

The first step of the algorithm is the "coloring" process. I pass through all Nodes and in every Node I check if there is a parent with smaller number. Each Node's parents are given from between CSC[i] and CSC[i + 1] where i is the current Node. The process is repeated until no colors are changed.

The second step is to check what Nodes lead back to the root Node. I did that recursively using a function that takes as input the root Node's color and Index. Every time it's called the root color remains the same but the index changes as we move deeper in the tree. Every time a Node is called, it's color changes to a negative number specifying the SCC's index. That ensures that the Node will not be called again during the rest of the algorithm. All Nodes with the same negative number belong in the same SCC.

After the algorithm is finished, the total number of SCCs is refreshed and it checks if there are more Nodes left in the tree. The process is repeated until there are no more Nodes left.

Serial to Parallel

My approach to this problem was that I wanted to make parallelization as easy as possible, so the first thing I had to account for was the race conditions.

First step was to create an identical array to the coloring, but it wouldn't change until the end of the coloring process. That way every Node stores the minimum of the parents' colors, including its own, and "gives" the color that it has before the coloring process. This will require some more repetitions of the coloring process but this way there won't Be any race conditions. All there is left is to implement a parallel for at the start of the process.

The second step was to check all the unique colors after the coloring process has finished and store them in an array. Then we start for each color we start at the root and repeat the Step as serial. We encountered some problems in this step that we will talk for in the implementations.

Parallel Implementations

Opencilk/Openmp:

The first step worked as planned and by implementing a parallel for at the start of the coloring process I managed to parallelize it for both Opencilk and Openmp.

I tried to repeat the same for the second part of the algorithm by trying run the recursive function in parallel for different colors. I added some restrictions to the function and also added the second array with temporary colors so there would be no race conditions. It worked for the first small matrix but not for the rest. I debugged it with cilksan, the cilk debugger for race conditions, and manually. It kept stopping in the same depth, so my conclusion is that threads do not allow the same recursive depth as serial. The segmentation fault that I was getting was most likely because of the depth, as it seemed to work in the smaller matrix. I did manage to make it work by using tasks in openmp and using Cilk_spawn/cilk_sync for opencilk, but the time needed was almost identical, if not worse, than before making the change. The reason that I believe that happened is because of the single Node SCCs. Most of the matrixes that I tested usually had one or two Huge SCCs and some smaller or single Node. So, in the end one thread was dealing with the big one while the others were simply performing miniscule tasks and probably the time tasks and cilk_spawn/cilk_sync take is larger than the time they save. In the end I decided to not include this in my code.

Pthreads:

With pthreads parallelizing the coloring process is trickier as it doesn't have easy tools like the two mentioned above, but the plan remained the same.

In order to make it work I had to manually break the graph in the number of Threads that I wanted to use. To do that I created a function that the pthreads would use and pass as arguments the number of the thread.

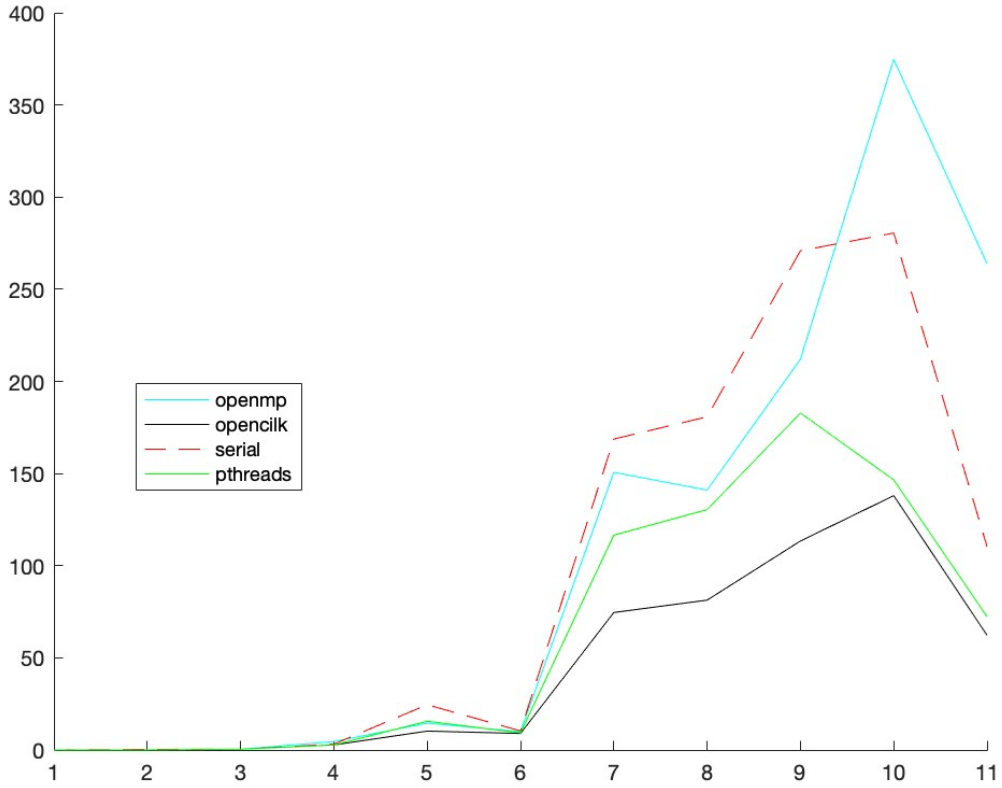
In the function that I created I repeated the same thing as in openmp/cilkl, but instead of going through all the Nodes with the build in parallel for, I was going through 1/num_of_threads of the total Nodes. By using the Thread number, I managed to create a for loop that would run in parallel but with different starts and ends. Of course, there were not any race conditions as I repeated the same strategy as with the other two ways.

The only thing that I had to do more in order to achieve that was to set all the arrays and parameters that the function would use to global, as there is no other way to access them. The second part of the algorithm remained the same.

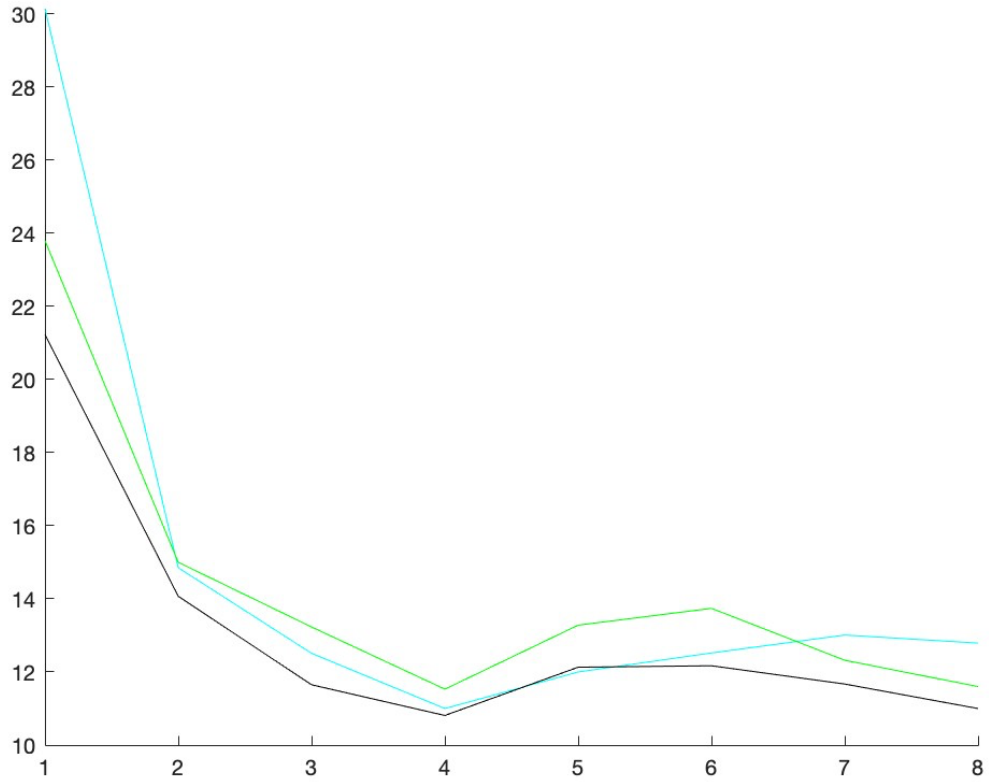
All the resource code, the instructions to compile the code and the matrices list will be given in Notes at the end of the report.

Results:

The graph below shows the time needed for vertices with growing edges:



As we can clearly see the more the graph grows, the better the parallel performance is, what we expected, expect from the last 2 of Openmp. I couldn't identify why that happens, As it's clearly not the expected result. All parallel implementations use 4 threads.



The graph above shows the same matrix, in this case wiki-topcats, but given different number of threads every time. As we can see in all the cases the time needed minimizes at around 4 threads and after that it is almost the same or a little bit higher. From the experiments that I did and the information I exchanged with my colleagues, this number (in this case 4) seems to be the number of physical cores to the system. That is however not proven but my hypothesis through my experiments.

All the results above contain the read matrix time.

Notes:

As I explained before, I use a recursive part in my algorithm. A lot of systems use a maximum limit to the depth that a function like that can reach. Because the depth can be very big, u need to run:

ulimit -s unlimited

That will allow the function to reach the required depth.

You need to compile the code with the mmio.c file that I have in the repository so it can read the matrices. The command is:

gcc -flag <<The program you want to compile>> mmio.c

For the opencilk you need to compile it with the clang compiler in your system.
The -flag is the specific flag for each version (-fopenmp foe opemp etc..)

To run the program:
./a.out <<matrix>>

Where instead of <<matrix>> you add the matrix you want to read.

Be aware that the programs I wrote read 3 columns (i/j/value). In order to run matrices that have 2 columns (most of them after languages matrix) you need to remove the %lg and &val from the fscanf in the source code.(openmp line 166, opencilk line 169, pthreads line 182, serial-to-parallel line 171, serial line 147)

All of my programs were compiled with the -O3 optimization flag.

The link to all of the source files, including the header the program to read the matrices
https://github.com/themis1234/Parallel-and-distributed

All the graphs I used are in order:

Graph	n	m	ncc
Newman/celegansneural	297	2345	57
Pajek/foldoc	13356	120238	71
Tromble/language	399130	1216334	2456
LAW/eu-2005	862664	19235140	90768
SNAP/wiki-topcats	1791489	28511807	1
SNAP/sx-stackoverflow	2601977	36233450	953658
Gleich/wikipedia-20060925	2983494	37269096	975731
Gleich/wikipedia-20061104	3148440	39383235	1040035
Gleich/wikipedia-20070206	3566907	45030389	1203340
Gleich/wb-edu	9845725	57156537	4269022
LAW/indochina-2004	7414866	194109311	1749052

Arvanitis Themistoklis