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

Parallel algorithm implementation in C++

1st assignment for 2022-2023 Parallel and Distributed Systems course, Aristotle University of Thessaloniki,
Panagiotis Syskakis
Github repository: https://github.com/Pansysk75/pds

Useful utilities:
A few structs were implemented to represent graphs (in file graphs.hpp) :
DirectedEdge is a pair of integers that indicate the start and the destination of a directed edge.
DirectedGraph is a list of DirectedEdges and represents a graph.
GraphCSR and **GraphCSC** are graph representations that mimic the structure of Compressed Sparse Row and Compressed Sparse Column matrices. They are more efficient in memory usage and indexing speed (finding all edges that point to some node OR all edges that start from a node, depending on which is used).

In addition, Tarjan's SCC algorithm (scc_algorithms/tarjan.cpp) and a SCC equality test algorithm (scc_algorithms/testing.cpp) were implemented to test performance and correctness of this implementation.

Implementing the SCC Coloring Algorithm:
For representing the graph, a CSC representation was chosen, which given a node k can very quickly (O(1)) give us all edges that point to k.This is important because both the "color propagation" step and the "backwards BFS" step of the algorithm start from some node and navigate the graph backwards.

Firstly, we initialize two vectors of integers, **scc_ids** and **colors** (The color of node k will be stored in colors[k] etc). If scc_ids[k] = -1 , it means that node k hasn't been assigned to an SCC yet.When the algorithm finishes, nodes that form a SCC will have the same scc_id assigned to them.

We also make the vector **queue**, which at any moment holds the ids of nodes that haven't yet been assigned to a SCC. At the end of every iteration, any node which has been assigned to a SCC (ie scc_ids[k] != -1) is removed from the queue vector.
Additionally, a trivial trimming algorithm was implemented, which finds all trivial single-node SCCs (nodes that have 0-out or 0-in degree).

Other than that, the SCC Algorithm implementation directly follows the implementation in *BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems (*Algorithm 2) .

Coloring SCC Algorithm

```
1: procedure COLORSCC(G(V,E))
2: while G = ∅ do
3:    for all u ∈ V do Colors(u) ← u
4:    while at least one vertex has changed colors do
5:       for all u ∈ V in parallel do             //Color propagation loop
6:          for all u, v ∈ E do
7:             if Colors(u) > Colors(v) then
8:                Colors(v) ← Colors(u)
9:    for all unique c ∈ Colors in parallel do    //Backwards BFS loop
10:      Vc ← {u ∈ V : Colors(u) = c}
11:      SCVc ← BFS(G(Vc, E(Vc)), u)
12:      V ← (V \ SCVc)
```

In Parallel:

This algorithm was devised with parallelization in mind and thus can be easily parallelized. In line 5, we have the “coloring” for-loop. Iterations of this loop can run simultaneously or in any order, and the algorithm will produce the correct result*. Thus, we can in essence partition this for-loop into pieces, and distribute that work to many processing units. The same is true for the for-loop in line 9.

In OpenCilk and OpenMP, parallelizing these for loops was easy to do with the appropriate keywords (*cilk_for* and *#pragma omp parallel for*). The same approach was followed for the PThreads implementation, where a for_loop_partitioner class was created.

There are other parts that could be done in parallel, such as trimming, vector initializations at the start, as well as the reduction of the queue vector at the end of each iteration. Due to time constraints, however, they were not implemented.

```
*Note about data-races:
In line 8, Colors(v) might be assigned a value while another thread is accessing Colors(v)
for the comparison in line 7. This introduces a data race, which seems unimportant, since
any wrong color propagation will be corrected in the next iteration.
Similarly, in order to know if "at least one vertex has changed colors" (line 4), one would
use a "flag" boolean, which is initialized false, and changed to true if any thread performs
a color propagation. This simultaneous write is also, technically, a data-race, even though
all threads are writing the same value.
Even in these cases, there are no guarantees by the language, and compiler optimizations
might (and did, in the OpenCilk compiler) lead to wildly unexpected behavior. So, if 100%
certainty about the correctness of a program is needed, unprotected data-races should be
avoided.
```

PThreads implementation:

Firstly, I chose to abstract away from directly using the PThread facilities (error prone and more verbose). A Thread class was made to provide a simpler and more programmer-friendly interface.

The following remarks are also important:

- Launching (many) PThreads is costly
- Statically assigning work to a small number of PThreads may lead to reduced parallelization, because if the workload is unevenly distributed, the bulk of the work might end up being done by only a few of those threads.

Thus, the following approach was chosen:

When we want to parallelize a task, a thread-pool is created, which spawns a small number of PThreads. Then, the available work can be broken up and "packaged" into self-contained functions (using the std::function facility), and is loaded into work-queues.

Each PThread in the thread-pool acquires work from its own work-queue and executes it.

When a thread finishes its own work queue, there is also the option to steal work from neighboring work queues.

Advantages of this approach:

- Implementation is hidden away from the user, and writing code to run in parallel can be non-intrusive.
- There is a distinction between threads (which are costly to create) and “tasks” (small lightweight units of work).
- The use of queues allows for dynamic distribution of the workload, because if there is any work available in a work-queue, any thread can claim it.

Limitations of this approach:

- Currently, every call of for_loop_partitioner creates a new thread pool. An approach that might work better is to reuse the same thread pool throughout the whole program, and use condition variables to signal the threads when there is more work added to the work queues.

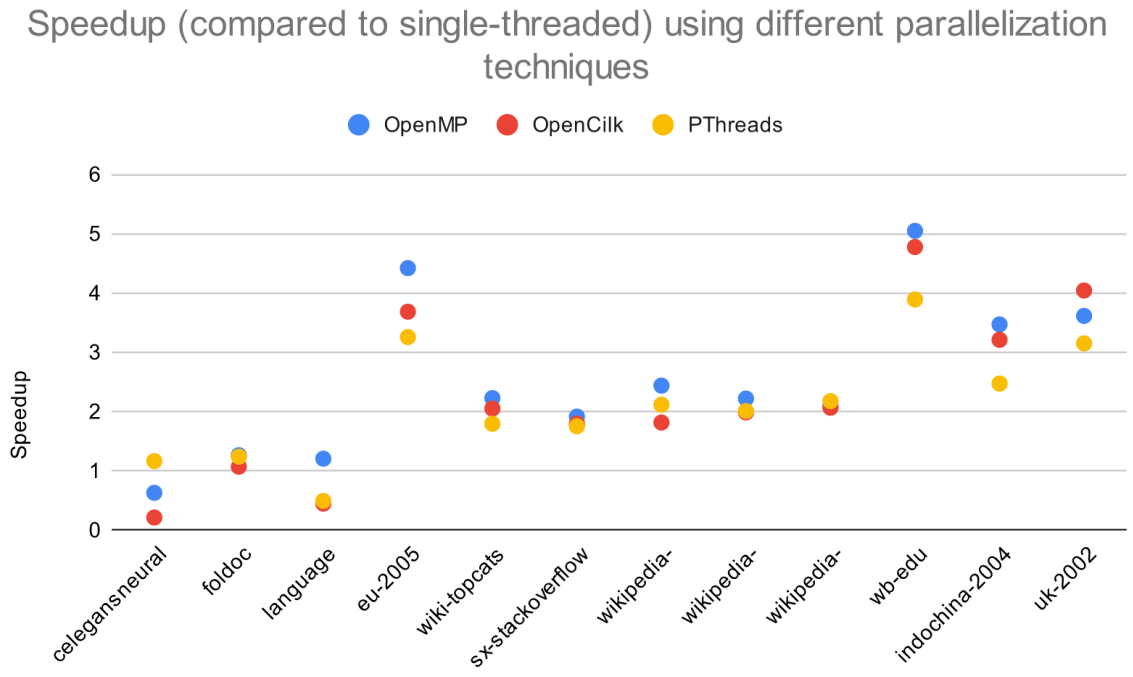
Performance measurement:

The execution time of the SCC Coloring algorithm (including the trimming process) was measured. In order to get consistent results, each measurement was taken at least 10 times and averaged. The measurements were done on a laptop 4-core Intel i5-8250U 1.60GHz CPU. Graphs larger than uk-2002 were not possible to measure, due to system memory limitations.

We see that all parallel implementations perform similarly, with OpenMP implementation having the most consistent results, and the PThreads implementation falling behind in larger graphs.

	celegansneural	foldoc	language	eu-2005	wiki-topcats	sx-stackoverflow
Sequential	0.048960 ms	1.9138 ms	67.881 ms	2032.39 ms	1787.54 ms	4365.21 ms
OpenMP	0.077653 ms	1.5131 ms	56.272 ms	459.75 ms	802.194 ms	2278.38 ms
OpenCilk	0.228012 ms	1.7856 ms	151.352 ms	551.51 ms	871.409 ms	2429.39 ms
PThreads	0.041985 ms	1.5432 ms	137.398 ms	623.88 ms	995.483 ms	2489.83 ms

	wikipedia-20060925	wikipedia-20061104	wikipedia-20070206	wb-edu	indochina-2004	uk-2002
Sequential	4576.2 ms	4866.2 ms	6300.1 ms	33138 ms	95302 ms	204988 ms
OpenMP	1876.2 ms	2191.8 ms	2995.8 ms	6563 ms	27456 ms	56709 ms
OpenCilk	2519.5 ms	2449.4 ms	3049.6 ms	6937 ms	29676 ms	50705 ms
PThreads	2161.9 ms	2420.2 ms	2893.5 ms	8514 ms	38525 ms	65045 ms



Additionally, the SCC Coloring algorithm was compared to a recursive implementation of Tarjan's algorithm, which was about one order of magnitude faster for the smaller graphs (larger graphs could not be tested, as the recursive function overflowed the memory stack).

Sources and interesting material:
1: BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems
2: How to miscompile programs with "benign" data races, Hans-J. Boehm