Aristotle's University of Thessaloniki
Polytechnic facility
Electrical and Computer Engineering
Electronics and Computers Section

Parallel and Distributed Systems

First Project

Project of
Isidoros Tsaousis-Seiras
AEM: 10042

Code for this project can be found here

December 5, 2022

Parallel and Distributed Systems
First Project

Ισίδωρος Τσαούσης-Σειράς
isidtsao@ece.auth.gr

December 5, 2022

Contents

Intro

This report presents the results of a study on the performance of different implementations of the SCC algorithm as described in [1], in three different threading environments: openCilk, openMP, and pThread. The study was conducted to compare the of this algorithm in multi-threaded environments.

In short, he algorithm works by assigning a unique color to each vertex in the graph, then updating in parallel the colors of each vertex based on the colors of its neighboring vertices. Once all the vertices have reached a stable coloring, the algorithm performs a parallel breadth-first search on each set of vertices with the same color to find the strongly connected components. The algorithm then removes these strongly connected components from the graph and repeats the process until the graph is empty. The algorithm works because the propagation of the colors happens in reverse of the BFS, and thus intersecting sets of the two are SCCs.

A graph is represented by its adjacency matrix. This way vertices are represented by their 'id' $\in \mathbb{N}$ and edges are represented by the contents of the matrix:

$$\begin{aligned} A_{i,j} &= 1 \text{ if vertex i} \rightarrow \text{j} \\ &= 0 \text{ otherwise} \end{aligned} \tag{1}$$

	OpenMP	OpenCilk	pThread	serial
celegansneural	0,793	0,338	0,385	0,045
foldoc	2,437	0,985	3,099	1,922
language	53,735	22,915	44,028	69,656
eu-2005	225,287	174,017	327,257	333,57
wiki-topcats	1570,04	1375,14	3062,255	4554,656
sx-stackoverflow	901,109	745,481	1310,731	1161,978
wb-edu	21725,658	17163,897	24606,245	42960,992
indochina-2004	14084,655	11086,592	21082,211	16468,548
uk-2002	21335,535	16139,5	20208,73	35609,951
arabic-2005	14164,525	12265,853	18311,773	25717,268
uk-2005	85555,417	71959,539	99837,427	129612,258

Figure 1: Full times of the algorithm on the experimental setup

Chapter 1

Part 1

1.1 C++ Implementation

1.1.1 Graph Representation

Due to the small amount of neighbors of each node, the matrix is extremely sparse. So we used a Compressed Sparse Column formatted matrix (CSC) to store it. This was implemented as the following struct:

```
struct Sparse_matrix {
    size_t n;
    size_t nnz;
    std::vector<size_t> ptr;
    std::vector<size_t> val;

    enum CSC_CSR {CSC, CSR};
    CSC_CSR type;
};
```

This resulted in very easy and fast access to the neighbors of each vertex: the neighbors that point towards a vertex `j` are the rows with non-zero elements in column `j`, which is just `val[ptr[j]], val[ptr[j]+1],..., val[ptr[j+1]]`.

If we compress the matrix according to its rows instead of its columns we get a CSR matrix, which we can use in the same way to get the neighbors that `j` points towards.

Using both of those methods means we have an $O(1)$ way of finding both incoming and outgoing neighbors, that is very space efficient and since their `id`s are stored contiguously, we also minimize cache misses.

In practice, because only one Compressed Sparse Matrix could fit in the experiment's setup memory at a time, the algorithm was tweaked to only require one of them to function properly.

1.1.2 SCC representation

Each vertex belongs to some SCC, so each vertex was given a variable `scc_id` that recorded which SCC it belonged to. The id of an SCC is the id of the smallest vertex it contains.

1.1.3 Trimming

A substantial part of the vertices in most graphs form trivial SCCs by themselves. We can detect such vertices by counting their incoming and outgoing neighbors and removing them if they have none. While this creates more potential trivial vertices, in practice one non-recursive run of this algorithm gives the best performance. In this implementation a non-recursive version was used.

Because outgoing neighbors sets are not readily available, a list of the vertices with no outgoing neighbors can be constructed by keeping track of the vertices that do not appear as any vertex's incoming neighbor. Then any vertex that either is in that list or is found to have no incoming neighbors is discarded. A check is required to determine if some neighbor has been removed or not.

Handling the special case of size one SCCs this way leads to a dramatic speedup both in the serial and the parallel implementation.

The time complexity of this step regardless of which of the versions is used is $O(vertices\_left)$, and what makes this improve the algorithm is that those vertices will not affect the coloring or propagation stages at all.

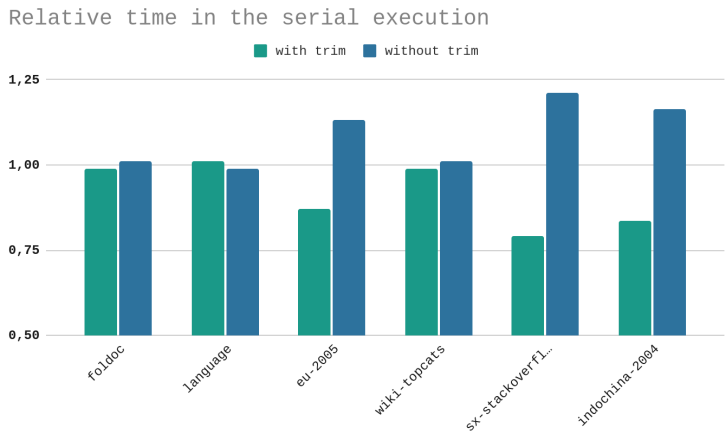

Figure 1.1: Execution time relative to the average between the two versions.

1.1.4 Color Propagation

A vector of all vertices not yet added to some SCC, named `vleft` is kept in order to reduce lookups. In the start of each "coloring round" the algorithm goes through all vertices left, so by using `vleft` we avoid going through all vertices in every iteration. The performance gain of this however is questionable, since the overhead of removing entries might surpass the gains.

1.1.5 BFS

After colors are stable, a BFS is initiated in every color group, starting from the vertex that gave the group its color. In the beginning this was implemented as a matrix multiplication as per [2], but this was later scrapped due to performance reasons, and instead BFS was implemented using a standard queue method. The assignment of `scc_id`s is handled inside the BFS function so that the value of `scc_id` can serve as an indicator of removed/visited vertices, as well as avoiding the iteration that would be necessary for the assignment of the `scc_id`s was it to simply return the `id`s of visited vertices.

1.1.6 Experimental Setup

All tests were run on an AMD® Ryzen 5 5600h processor with 12 cores

1.1.7 Serial Times

Dataset	Time(ms)
celegansneural	0,045
foldoc	1,922
language	69,656
eu-2005	333,57
wiki-topcats	4554,656
sx-stackoverflow	1161,978
wb-edu	42960,992
indochina-2004	16468,548
uk-2002	35609,951
arabic-2005	25717,268
uk-2005	129612,258
twitter7	195393,672

Figure 1.2: Execution time for the various datasets in the sequential algorithm

1.2 Parallelization

All three major subroutines (Trimming, Color Propagation, BFS) are parallelizable.

Trimming

Parallelizing this statistically leads to the same number of operations as the serial version, since all that changes is the examination order of the vertices, which was arbitrary to begin with. This however becomes non-deterministic, which leads to much harder debugging and harder to measure performance. That being said, simply changing the variables to `std::atomic` takes care of race conditions, and allows a trivial parallelization.

Color Propagation

The same is true for the colors, however due to some overwriting happening a 20-30% increase in loops was observed, which was counterbalanced by the speedup.

BFS

This part is embarrassingly parallel, as by design it acts on mutually exclusive sections of the graph for every iteration. The clean linear speed up this offered was apparent in the BFS-heavy dataset `wiki-topcats`

1.2.1 OpenMP

Simple `pragma omp parallel for` clauses where used. This was the easiest multithreading environment to integrate, and required the least amount of changes to function properly.

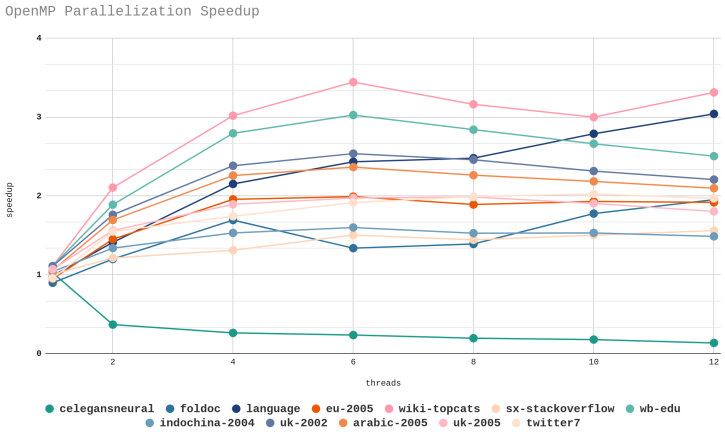

Figure 1.3: Speedup with OpenMP compared to the serial algorithm

1.2.2 OpenCilk

Harder to integrate than OMP due to a lack of documentation. However substituting `cilk_for` in the appropriate places worked. A weird bug that poped up was that a 'Flag' variable, that would only be accessed to write `true` into, sometimes kept a wrong value at the end of the loop. After investigation it was discovered that this is due to the way OpenCilk handles looping. Changing the type of the flag to atomic worked here.

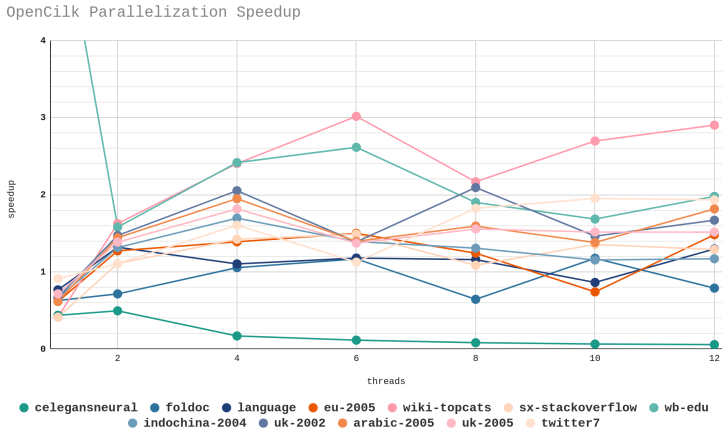

Figure 1.4: Speedup with OpenCilk compared to the serial algorithm

1.2.3 pThread

This was the hardest and messiest version to implement, perhaps due to a lack of time and planning. The entirety of the functions needed to be specially interfaced using intermediate functions to support multithreading. All loops needed to be scheduled in advance, and while this offered great flexibility it did not prove useful compared to the other versions performance-wise.

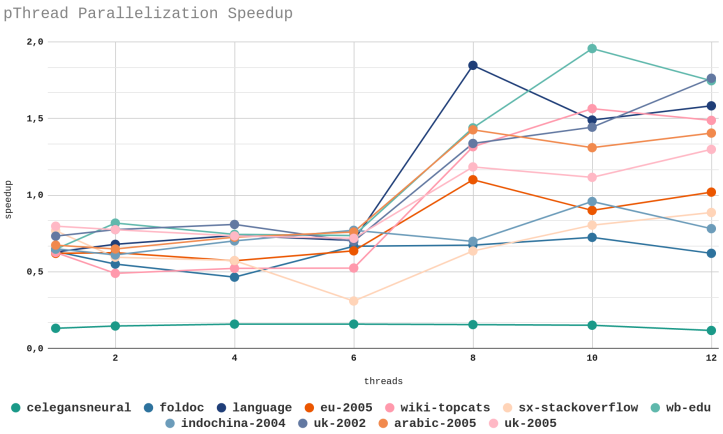

Figure 1.5: Speedup with pThread compared to the serial algorithm

1.3 Conclusion

OpenCilk gave the best performance overall, and was easy enough to implement, however the combination of it's lack of documentation and non-intuitive design created a bug that was quite dificult to figure out. OpenMP was the best "out of the box" solution. Pthread required much longer than the other two to implement, and it's lack of abstraction left all scheduling on the programmer, which did not do a very good job compared to OpenMP and OpenCilk schedulers.

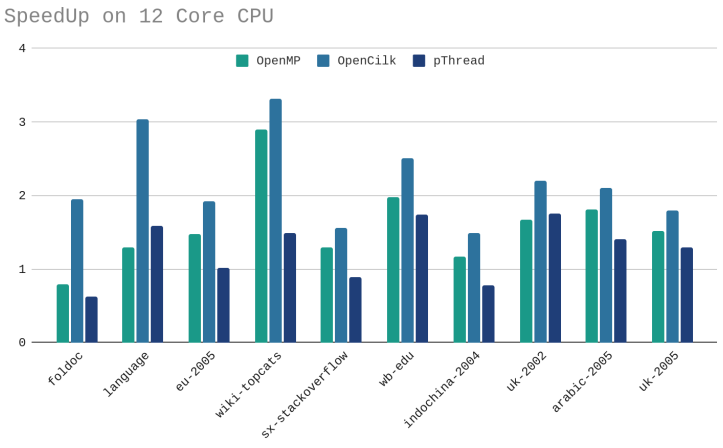

Figure 1.6: 12 Core Speedup using the different implementations

Bibliography

[1] Slota, G., Williams, L., Weimer, M., Li, H., and Zhang, S. (2014). Parallel Strongly Connected Component Algorithms in the Wild. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 289-300. https://www.cs.rpi.edu/ slotag/pub/SCC-IPDPS14.pdf.

[2] Aydın Buluç. In *Parallel breadth-first search on distributed memory systems*, https://people.eecs.berkeley.edu/ aydin/talks/SC11-BFS.pdf