IN4200 Home Exam 1, spring 2020: Counting Mutual Web linkage Ocurrences

CANDIDATE: 15224

April 14, 2020

Abstract

This report contains the most important algoritmic information and results from the functions made during this home exam. We did get a speedup from parallelizing the functions, and we did get seemingly reasonable results for the top n webpages.

1 Introduction

In this exam we were asked to program functions in C to read web graph files, count mutual web linkage ocurrences and rank the top n webpages based on number of involvements per webpage. We were asked to test our functions and benchmark them, make a makefile and a readme. This report will document algorithmic information and general information on how we solved these tasks.

2 Method: Algorithmic information

To read the web graph and count mutual links we used two different methods. First we used a matrix format for an easy to comprehend and convenient to program (at least in principle) solution to the problem presented. This method was fine for smaller web graphs like test_graph.txt. However for a larger problem size like for web-NotreDame.txt we simply do not have enough memory on a standard laptop to allocate a matrix. Assuming each element is 1 byte (char) we would need a 325729x325729 matrix which naively would mean more than 106GB of memory. This calls for a different format, and because we have a sparse matrix we can use the CRS format. This format is based on storing the information in the matrix in three arrays. One array for the values of each non-zero element, however this is not needed in our case since all non-zero elements is ones. The second array is for storing the column indices of each non-zero element in the matrix. The last array is a row pointer that stores the indices at which new rows start in the column index array. With this we can easily represent the web graph on a standard laptop.

2.1 Matrix Format

2.1.1 read_graph_from_file1.c

The web-graphs all have the same structure, two columns containing which webpage links to another. Webpage j in one column links to webpage i in the next. This means that the matrix may be allocated using calloc to fill the matrix with zeroes. Then we read the web graph file line by line and fill the matrix with ones according to the equation below:

$$M_{i,j} = 1 \tag{1}$$

This means there will be a 1 for each number of links (edges). The matrix will be of size $N \times N$ where N is the number of nodes.

2.1.2 count_mutual_links1.c

Now that we have a matrix containing the links we can start counting the number of mutual links and involvements. To do this we need to go through every matrix element row by row. We count up each 1 in a row and store it in a counter value called "links_to" (there will be one of these for each row). Then we through each element in a row and check if it is a one. If yes then we add up the current links_to at that location like this:

$$num\_involvements[k] = num\_involvements[k] + links\_to - 1 \tag{2}$$

Which gives us every time location k was involved and how many (links_to - 1) times it was involved at that location. After going through each row we will know how many time every single location k was involved in mutual linkage.

Now we have almost completed one row but we also need to tally up the number of total mutual links on that row. This follows the pascal triangle number pattern, but does not start at one node, instead it starts after the first node. The total number of mutual links is then the number from the previous rows plus the number on the current row:

$$T = T + \frac{(l_t)(l_t - 1)}{2} \tag{3}$$

Here T is the total number of mutual links and l_t is the links_to value.

2.2 CRS Format

2.2.1 read_graph_from_file2.c

Since we do not want to use the matrix format for larger web graphs we need to create the correct vectors for the CRS format instead. Here we start out by reading the file line by line and storing the information in two arrays. In addition we can here count up each one in a row which is used for the row

pointer later. When we have a vector of the number of links per row which we can use to sum over all the rows and get the row pointer we want, with values between 0 and the number of links:

$$row\_pointer[k+1] = row\_pointer[k] \tag{4}$$

Where k is the current row. We now want to iterate over all links and find each column index. However there are several column indices per row so we make a counter array that offsets each index in the column index array per value on a row so that we can count up all column indices at the correct locations. We then use the row pointer and the counter as locations in the index array. The values of the index array are given by the first column in the web graph file. The computation is then given by:

```
int *counter = calloc((*N), sizeof(*counter));
for(size_t i = 0; i < *N_links; i++){
    (*col_idx)[(*row_ptr)[ToNodeId[i]] + counter[ToNodeId[i]]] = FromNodeId[i];
    counter[ToNodeId[i]]++;
}
```

2.2.2 count_mutual_links2.c

The algorithm of this is very similar to the one counting links with the matrix format however its even simpler. In this case we can find the number of links in a row k by taking $links[k] = row_{start} - row_{end}$. From this we can find total number of mutual links using (3). And find the number of involvements by using

$$num\_involvements[k] = num\_involvements[k] + links[k] - 1 \tag{5}$$

Which is actually the same in the previous program. Note that in the actual program count_mutual_links2.c $links[k]$ is called "counter".

2.3 top_n_webpages.c

The algorithm for finding the top n webpages is actually pretty simple. We take a copy of the num_involvements array to avoid destroying the original array. Then we find the max element in the copy array. We store that element in a top_results array (which is n long), and we set that value to -1 to avoid finding the same value twice.

2.4 OpenMP

To parallelize the count_mutual_links1.c and count_mutual_links2.c we just need to use pragma parallel for, and make sure we do not have any false sharing by having pragma private and pragma reduction(+:). However parallelizing top_n_webpages was not a matter of one line of code with the right pragmas.

Here we needed to split the work up between each thread. Every thread finds their own top n webpages. We then use the master thread to sort the list of top n webpages found by each thread. To do this we had to copy the num_involvements array and set the values found to -1 once for the parallel code and once for the single thread code.

3 Results

All results were generated on a thinkpad model L590 with 16GB of RAM, and a intel i7-8665U running on the latest version of ubuntu:

This is the output of running main.c:

```
Web graph table:
| 0 0 0 0 0 0 1 0 |
| 1 0 1 1 0 0 0 0 |
| 1 0 0 0 0 0 0 0 |
| 0 1 0 0 0 0 0 0 |
| 0 0 1 1 0 0 1 0 |
| 0 0 0 1 1 0 0 1 |
| 0 0 0 0 1 0 0 1 |
| 0 0 0 0 1 1 1 0 |

Number of involvements per webpage:
|  2  0  4  6  5  2  4  3 |

Total number of mutual links: 13
n cannot be larger than num_webpages, n is set equal to num_webpages 8
Ranking: 1, Webpage number: 3, Number of involvements: 6
Ranking: 2, Webpage number: 4, Number of involvements: 5
Ranking: 3, Webpage number: 6, Number of involvements: 4
Ranking: 4, Webpage number: 2, Number of involvements: 4
Ranking: 5, Webpage number: 7, Number of involvements: 3
Ranking: 6, Webpage number: 5, Number of involvements: 2
Ranking: 7, Webpage number: 0, Number of involvements: 2
Ranking: 8, Webpage number: 1, Number of involvements: 0

N = 325729, N_links = 1497134

Total number of mutual links: 251035302
Ranking: 1, Webpage number: 7137, Number of involvements: 57115
Ranking: 2, Webpage number: 260552, Number of involvements: 42175
Ranking: 3, Webpage number: 236095, Number of involvements: 42175
Ranking: 4, Webpage number: 260725, Number of involvements: 42171
Ranking: 5, Webpage number: 260726, Number of involvements: 42171
Ranking: 6, Webpage number: 260727, Number of involvements: 42171
```

```
Ranking: 7, Webpage number: 260728, Number of involvements: 42171
Ranking: 8, Webpage number: 260729, Number of involvements: 42171
Ranking: 9, Webpage number: 260730, Number of involvements: 42171
Ranking: 10, Webpage number: 260731, Number of involvements: 42171
```

3.1 Testing

We wanted to test the validity of our algorithm. This was done by hardcoding the expected results for each and checking the results of our algorithms. Below you can see the output of running the test.c program:

```
read_graph_from_file1 took 0.280000ms to complete on test_graph.txt
Test passed. Expected number of nodes: 8, Found: 8.
Test passed. Expected number of egdes: 17, Found: 17.
Test passed. Matrices are identical.

count_mutual_links1 took 0.006000ms to complete on test_graph.txt
Test passed. Expected number of total mutual links: 13, Found: 13.
Test passed. Number of involvements vector is identical to the expected vector.

read_graph_from_file2 took 0.054000ms to complete on test_graph.txt
Test passed. Expected number of nodes: 8, Found: 8.
Test passed. Expected number of links: 17, Found: 17.
Test passed. Row pointer vectors are identical.
Test passed. Column index vectors are identical.

count_mutual_links2 took 0.003000ms to complete on test_graph.txt
Test passed. Expected number of total mutual links: 13, Found: 13.
Test passed. Number of involvements vector is identical to the expected vector.

calc_top_n_webpages took 0.005000ms to complete on test_graph.txt, without printing.
Test passed. Ranking vector is identical to the expected vector.

Tests passed: 12/12
```

3.2 Benchmarking

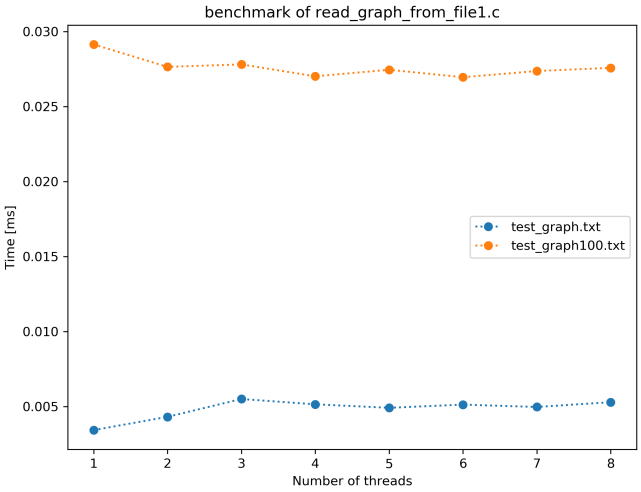

Figure 1: Average time benchmarking read_graph_from_file1.c plotted against number of threads used.

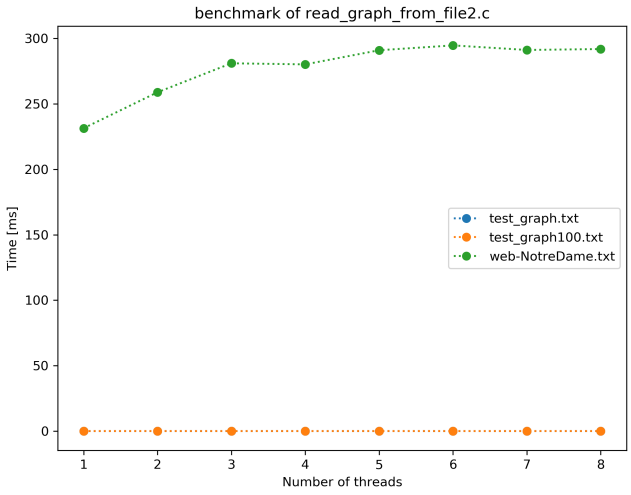

Figure 2: Average time benchmarking read_graph_from_file2.c plotted against number of threads used.

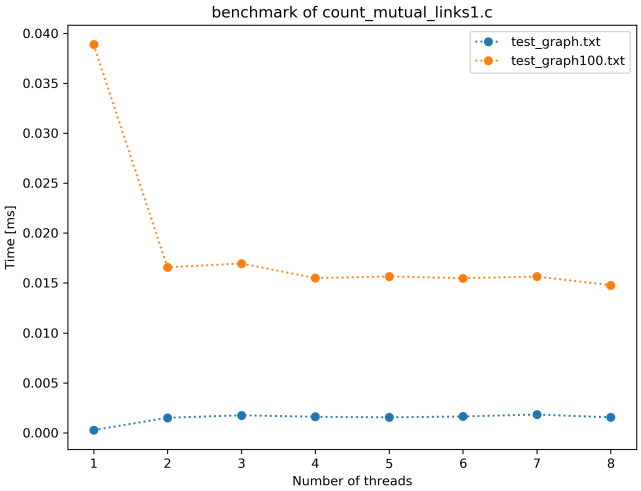

Figure 3: Average time benchmarking count_mutual_links1.c plotted against number of threads used.

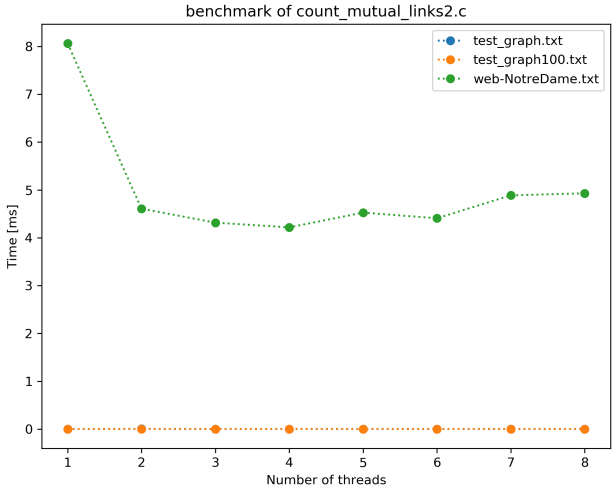

Figure 4: Average time benchmarking count_mutual_links2.c plotted against number of threads used.

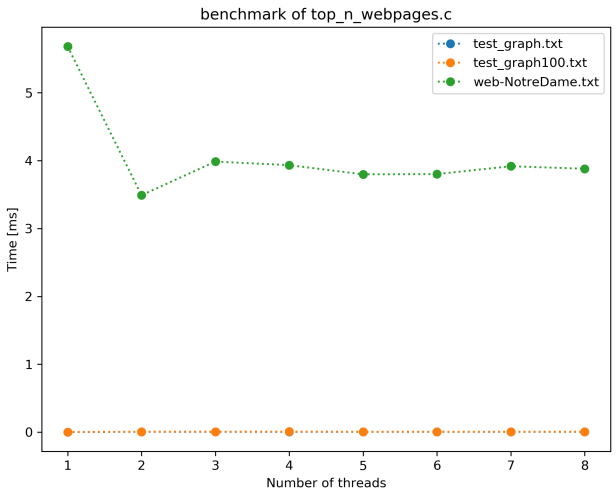

Figure 5: Average time benchmarking top_n_webpages.c plotted against number of threads used.

4 Discussion and Conclusions

As we can see we get a clear speedup when changing from serial to parallel. However theoretically we should have been able to get a speedup from increasing the number of threads. This is surprising because our algorithm for finding the top n webpages should have benefited more at least for larger web graphs like the Notre Dame dataset. At the very least from the results of our testing program we can see a substantial speedup switching from the matrix format to the CRS format. Anything else would have been worrying since we are doing less operations with CRS than with a matrix. The most important point about the CRS format however is probably the memory efficiency, since we were outright not able to run the Notre Dame dataset with the matrix format. A last important note is that our algorithms seem to have less than consistent results on different operating systems. The programs all ran at ifi, however since the version of OPENMP was different our pragmas did not work there. Last time i tested the program at an ifi workstation the results did seem consistent with the ones on my own system.