Counting Mutual Web Linkage Occurrences
IN3200/IN4200 Home Exam 1

Written by:

Candidate Number
15204

University of Oslo

March 30, 2020

Contents

1 Introduction

The aim of this project is to implement a highly efficient algorithm in terms of computational speed, that highlights important features of web graphs. In addition, we will also implement a parallelised version of the code utilising OpenMP.

2 Theory

2.1 Web graph

The linkage situation among a group of webpages can be illustrated by a web graph, which shows how the webpages are directly connected by outbound links. An example of a web graph can be found below in Figure 1. In this particular example, there are eight webpages, where webpage No. 1 is directly linked to webpage Nos. 2 3, webpage No. 2 is directly linked to webpage No. 4, and so on.

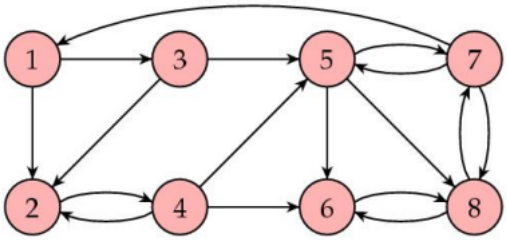

Figure 1: A very simple graphical example of eight webpages that are linked to each other

2.1.1 File format

A web graph as described above can be presented in a file as

# Nodes: 8	Edges: 17
# FromNodeId	ToNodeId
0	1
0	2
1	3
2	4
2	1
3	4
3	5
3	1
4	6
4	7
4	5
5	7
6	0
6	4
6	7
7	5
7	6

where *Nodes* refers to the number of webpages and *Edges* refers to the number of linkages between the webpages. The column *FromNodeId* holds information of the origin node of a outgoing connection, while the column *ToNodeId* holds information about the destination node.
More information on how the file format of a web graph is constructed is shown in the appendix below.

2.1.2 Data storage format

There exists many data storage formats for a web graph, but we will only focus on two methods in this project.

2D Table
The 2D table format method, is the most convenient data storage format for a web graph. It is of dimension $N \times N$, where N denotes the number of webpages, also known as Nodes. The values in the table are either "0" or "1". If the value on row i and column j is "1", it indicates a direct link from webpage j (outbound) to webpage i (inbound). For instance, the corresponding 2D table for the web graph in Figure 1 is as follows:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{1}$$

CRS
When the number of webpages N is large, and if the total number of direct links (denoted as N_links) is relatively small, then most of the values in the 2D table will be zero. This can result in a huge waste of storage, due to the huge number of zero values. Therefore, a different method is called for, and one other method is called the CRS method, which stands for compressed rows. This method consists of two 1D arrays, col_idx and row_ptr. The col_idx array is of length N_{links} = Edges, and it stores, row by row, the column indices corresponding to all the direct links. The row_ptr is of length $N + 1$, which can be used to "dissect" the col_idx values, row by row.

2.2 Mutual web linkage

There are many ways to characterise a webpage group, but one we will focus on is to analyse how likely two webpages are directly linked to a third common webpage. For that purpose, it is necessary to count the number of occurrences for mutual web linkage, that is, two webpages i and j (outbound) are both directly linked to webpage k (inbound), where $i \neq j \neq k$.
Take for instance the web graph in Figure 1. There are in total 13 occurrences of mutual web linkage, more specifically

$$\begin{aligned} &(w1, w3) \Rightarrow w2,\ (w1, w4) \Rightarrow w2,\ (w3, w4) \Rightarrow w2, \\ &(w3, w4) \Rightarrow w5,\ (w3, w7) \Rightarrow w5,\ (w4, w7) \Rightarrow w5, \\ &(w4, w5) \Rightarrow w6,\ (w4, w8) \Rightarrow w6,\ (w5, w8) \Rightarrow w6, \\ &\qquad\qquad\qquad\qquad\qquad\quad (w5, w8) \Rightarrow w7, \\ &(w5, w6) \Rightarrow w8,\ (w5, w7) \Rightarrow w8,\ (w6, w7) \Rightarrow w8. \end{aligned}$$

Figure 2: Illustration of mutual web linkage occurrences from Figure 1

Another statistical quantity we will look into is the number of times that a webpage is involved as outbound for the mutual web linkages. For the web graph in Figure 1, webpage 4 is involved six times, webpage 5 is involved five times, webpages 3 and 7 are each involved four times, webpage 8 is involved three times, webpages 1 and 6 are each involved twice and webpage 2 is never involved.

3 Implementation

Programs used in this project can be found on the github repository https://github.com/Moejay10/IN4200/tree/master/Project1, and the "README.md" explains how to compile and execute the programs.

3.1 Code structure

To keep the program neat, specific parts were placed in specific files. We have divided the program into a serialized version and a parallelised version.

The serialized version
This consists of the files:

- main.c
  - Contains the main part of the project that makes use of all the functions implemented below.
- count_mutual_links1.c
  - Contains the function count_mutual_links1 and test_count_mutual_links1.
- count_mutual_links2.c
  - Contains the function count_mutual_links2 and test_count_mutual_links2.
- top_n_webpages.c
  - Contains the function top_n_webpages and test_top_n_webpages.

The parallelised version
This consists of the files:

- OMP_main.c
  - Contains the main part of the project that makes use of all the functions implemented below.
- OMP_count_mutual_links1.c
  - Contains the function OMP_count_mutual_links1 and test_OMP_count_mutual_links1.
- OMP_count_mutual_links2.c
  - Contains the function OMP_count_mutual_links2 and test_OMP_count_mutual_links2.
- OMP_top_n_webpages.c
  - Contains the function OMP_top_n_webpages and test_OMP_top_n_webpages.

Found in both the serialized and parallelised version of the code

- read_graph_from_file1.c
  - Contains the function read_graph_from_file1 and test_read_graph_from_file1.
- read_graph_from_file2.c
  - Contains the function read_graph_from_file2 and test_read_graph_from_file2.
- functions.c
  - Contains functions used during the project.

3.2 Implementation of reading a webgraph from file

We implemented two functions to read the web graph file as illustrated above.

read_graph_form_file1
This function formats the web graph file as a 2D table. It takes in the filename, an empty variable N and an empty 2D array *table2D* as input. Then it scans the data file, row by row, where it first extracts the number of *Nodes* and *Edges*, where the empty parameter N is assigned the number of *Nodes*.
Thereafter, the file is scanned until the end of the file, and during this procedure the data containing information about outgoing connections is being extracted into the parameter table2D, where the values in the table are either 0 or 1.

read_graph_form_file2
This function formats the web graph file in the CRS format. It takes in the filename, two empty variables N and N_links and two empty vectors *row_ptr* and *col_idx* as input. Then it scans the data file, row by row, where it first extracts the number of *Nodes* and *Edges*, where the empty parameters N and N_links are respectively assigned the number of *Nodes* and *Edges*.
Thereafter, the file is scanned until the end of the file, and during this procedure the column *FromNodeId* is being placed in the vector *col_idx* and the column *ToNodeId* is being placed in *row_ptr*.

3.3 Implementation of counting mutual web linkages

We implemented two functions to count the total number of mutual webpage linkage occurrences, as well as the number of involvements per webpage as outbound for such mutual linkage occurrences.

count_mutual_links1
This function returns the total number of mutual webpage linkage occurrences. It takes in a parameter N, which is the number of nodes, 2D format of the web graph file and an empty vector *num_involvements*. It goes through the 2D table and calculates the the total number of mutual webpage linkage occurrences and assign that value to the return parameter of the function. It also calculates the number of involvements per webpage as outbound and assign that information to the vector *num_involvements*.

count_mutual_links2
This function returns the total number of mutual webpage linkage occurrences. It takes in two parameters N and N_links, which respectively is the number of nodes and edges, two vectors *col_idx* and *row_ptr* which is the CRS format of the web graph file and an empty vector *num_involvements*. It goes through the two vectors *col_idx* and *row_ptr* and calculates the the total number of mutual webpage linkage occurrences and assign that value to the return parameter of the function. It also calculates the number of involvements per webpage as outbound and assign that information to the vector *num_involvements*.

OMP_ount_mutual_links1 & OMP_ount_mutual_links2
These functions are just the parallelised implementations of the functions mentioned above.

3.4 Implementation of finding top webpages regarding involvements in mutual linkages

We implemented a function which finds the top n webpages with respect to the number of involvements in mutual linkages.

top_n_webpages
This function finds the top n webpages with respect to the number of involvements in mutual linkages, and print out these webpages and their respective numbers of involvements. It takes in a parameter *num_webpages*, which is just the number of nodes found in the web graph file, the vector *num_involvements* which contains the number of involvements per webpage as outbound and finally we have the parameter n which indicates the number of top webpages we are looking for. This function goes through the vector *num_involvements*, sorts it, finds the top n webpages and then prints out these webpages and their respective numbers of involvements.

OMP_top_n_webpages
This function is just the parallelised version of the function described above.

4 Results

All the time data presented in the tables and figures was collected by running the code 10 times at each thread setting and the means of those runs is what is presented below.

4.1 8 node web graph

Functions	CPU-time [s]
read_graph_from_file1	$2.68 \cdot 10^{-5}$
read_graph_from_file2	$2.44 \cdot 10^{-5}$

Table 1: Time measurements for the non-OpenMP parallelised functions

	CPU-time [s]							
Functions	**1 Thread**	**2 Threads**	**3 Threads**	**4 Threads**	**5 Threads**	**6 Threads**	**7 Threads**	**8 Threads**
OMP_count_mutual_links1	$3.0 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$2.0 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$7.1 \cdot 10^{-5}$
OMP_count_mutual_links2	$2.0 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$4.0 \cdot 10^{-6}$	$5.0 \cdot 10^{-6}$	$4.0 \cdot 10^{-6}$	$2.0 \cdot 10^{-6}$	$6.0 \cdot 10^{-6}$	$1.79 \cdot 10^{-4}$
OMP_top_n_webpages	$9.2 \cdot 10^{-5}$	$1.02 \cdot 10^{-4}$	$8.8 \cdot 10^{-5}$	$1.09 \cdot 10^{-4}$	$8.6 \cdot 10^{-5}$	$3.6 \cdot 10^{-5}$	$9.2 \cdot 10^{-5}$	$2.33 \cdot 10^{-4}$

Table 2: Time measurements for the OpenMP parallelised functions, where the number of OpenMP threads is varied

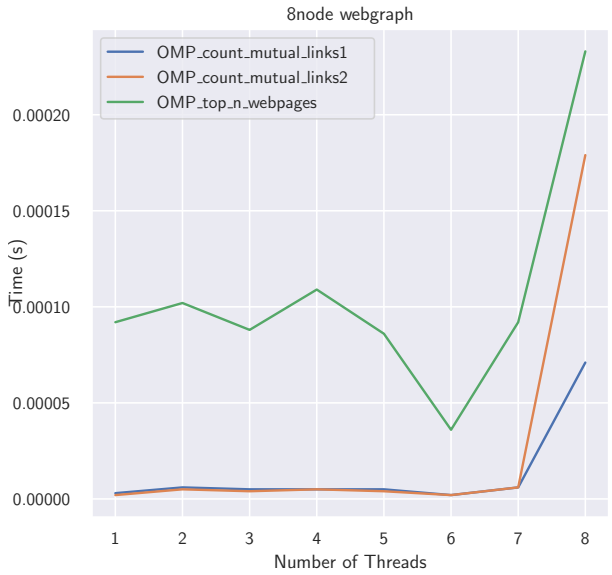

Figure 3: Visualisation of table 2, where the CPU-time consumption of the OpenMP parallelised functions is plotted against the number of OpenMP threads

4.2 Notre-Dame web graph

There is not a time presented for the read_graph_from_file1 in table 3, and that is because the NotreDame webgraph file is too big for the 2D table format. In other words, there was not enough memory to store the entire table in the hardware used in this project. This leads to functions that are dependent on results from read_graph_from_file1 not being able to run, such as function OMP_count_mutual_links1.

Functions	**CPU-time [s]**
read_graph_from_file1	NA
read_graph_from_file2	0.24

Table 3: Time measurements for the non-OpenMP parallelised functions

	CPU-time [s]							
Functions	**1 Thread**	**2 Threads**	**3 Threads**	**4 Threads**	**5 Threads**	**6 Threads**	**7 Threads**	**8 Threads**
OMP_count_mutual_links2	$2.73 \cdot 10^{-3}$	$1.80 \cdot 10^{-3}$	$1.55 \cdot 10^{-3}$	$1.60 \cdot 10^{-3}$	$1.90 \cdot 10^{-3}$	$2.05 \cdot 10^{-3}$	$2.33 \cdot 10^{-3}$	$2.62 \cdot 10^{-3}$
OMP_top_n_webpages	$1.72 \cdot 10^{-3}$	$1.73 \cdot 10^{-3}$	$1.76 \cdot 10^{-3}$	$1.77 \cdot 10^{-3}$	$2.44 \cdot 10^{-3}$	$2.69 \cdot 10^{-3}$	$3.09 \cdot 10^{-3}$	$3.29 \cdot 10^{-3}$

Table 4: Time measurements for the OpenMP parallelised functions, where the number of OpenMP threads

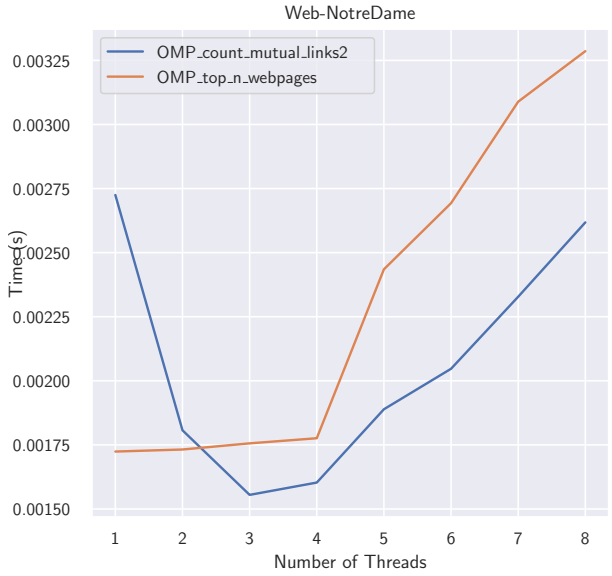

Figure 4: Visualisation of table 4, where the CPU-time consumption of the OpenMP parallelised functions is plotted against the number of OpenMP threads

5 Discussion

5.1 8 node web graph

The results in table 1 shows that there is a slight difference between the two functions that utilises different data storage formats, where read_graph_from_file2 with a CRS format is the one running faster. The reason we are not observing a more distinct difference in CPU-time between the two functions, may be that the 8node web graph file used is to small that we can observe any distinct difference between the two functions.

When the of OpenMP threads was varied, as shown in table 2 and figure 3, we observe that all the functions CPU-time fluctuated. All the functions seemed to have a minimum at 6 threads and the maximum at 8 threads, concerning the CPU-time. The most drastic CPU-time change occur at 8 threads for all functions, and that may very well be because of the OpenMP overhead, which could inhibit certain compiler optimisations, such as preventing loops from being vectorised or shared variables from being kept in registers.

5.2 Notre-Dame web graph

We observe a huge difference in CPU-time between the results from table 3 and table 1, regarding read_graph_from_file2. This is due to the vast amount of data difference between the two web graphs, where the Notre-Dame file contains 325729 Nodes and 1497134 Edges, compared to the 8node file which has only 8 Nodes and 17 Edges.

From the results given in table 4 and figure 4, we observe a different behaviour than we did in the case of 8node web graph file.
The function OMP_count_mutual_links2 exhibits somewhat a x^2 dependency on the number of OpenMP threads. It is obvious that the parallelisation yielded some efficiency, especially for the utilisation of 3 threads, where it has a minimum.
On the other hand, the function OMP_top_n_webpages seems only to become slower when the number of OpenMP threads are increased, which seems to indicate that the OpenMP overhead may inhibit certain compiler optimisations.

6 Conclusion

In this project we have implemented an algorithm that reads a web graph, counts the mutual linkage occurrences between them and finds the top n webpages with respect to the number of involvements in mutual linkages.

We studied the CPU-time of the algorithm as a function of the the number of OpenMP threads. In the case of the small web graph file (8node) we noticed there were fluctuations of the CPU-time when the number of OpenMP threads was varied, and that all functions CPU-time was drastically increased for the maximum number of 8 threads was reached. A possible explanation for this behaviour was believed to be because of the OpenMP overhead that inhibits some compiler optimisations.

In the case of the larger web graph file (Notre-Dame), the function OMP_count_mutual_links2 was observed to have a speedup until the OpenMP threads reached 3, and then it became slower. However, there were no speedup observed for the OMP_top_n_webpages function when the number of OpenMP threads was varied, in fact only a delay of the time was observed when OpenMP threads increased, which is also believed to because of the OpenMP overhead.

Appendices

A File format

The code reads a .txt file which must be formatted in the following way:

```
# Directed graph (each unordered pair of nodes is saved once): web-NotreDame.txt
# University of Notre Dame web graph from 1999 by Albert, Jeong and Barabasi
 # Nodes: 325729   Edges: 1497134
  # FromNodeId     ToNodeId
        0            0
        0            1
        0            2
        ⋮            ⋮
```

- The first two lines both start with the # symbol and contain free text (listing the name of the data file, authors etc.);
- Line 3 is of the form "# Nodes: integer1 Edges: integer2", where integer1 is the total number of webpages, and integer2 is the total number of links. (Here, nodes mean the same as webpages, and edges mean the same as links.)
- Line 4 is of the form "# FromNodeId ToNodeId";
- The remaining part of the file consists of a number of lines, the total number equals the number of links. Each line simply contains two integers: the index of the outbound webpage and the index of the inbound webpage;
- Some of the links can be self-links (same outbound as inbound), these should be excluded (not used in the data storage later);
- Note: the webpage indices start from 0 (C convention).