UiO : Department of Informatics
University of Oslo

IN4200: High-Performance Computing and Numerical Projects

Home Exam 1

Counting Mutual Web Linkage Occurrences

Candidate 15244

April 14, 2020

Contents

1 Introduction **1**

2 Methodology **2**

2.1 File Format of a Web Graph 2

2.2 Read and Store Web Graph as a 2D Table 3

2.3 Read and Store Web Graph in the CRS Format 4

2.4 Count Mutual Web Linkage Occurrences 5

2.4.1 With 2D Table as Storage Format 5

2.4.2 With CRS as Storage Format 6

2.5 Ranking the Top Webpages 6

2.6 Parallelization 6

3 Results and Discussion **7**

3.1 Read and Store Web Graph 7

3.2 Count Mutual Web Linkages 8

3.3 Top Webpages 8

4 Conclusion **9**

References **10**

1 Introduction

Directed links between a group of webpages can be illustrated by a *web graph* in which each webpage is a node and each hyperlink a directed edge that connects node i to node j if there is an outbound link in node i referring to node j. An example of a web graph can be seen in Figure 1.1

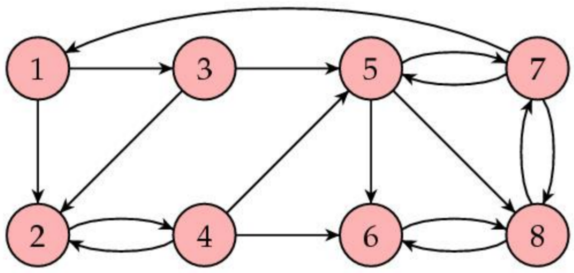


Figure 1.1: *Graphical representation example of eight webpages that are linked to each other. The figure is retrieved from the project description.*

A statistical quantity of interest is the number of occurrences for *mutual web linkage*, that is, two webpages i and j (outbound) are both linked to webpage k (inbound), where $i \neq j \neq k$. In the linkage situation shown in Figure 1.1, there are in total 13 occurrences of mutual web linkage, more specifically,

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

Another statistical quantity of interest is the number of times that a webpage is involved as outbound for the mutual web linkages. Table 1.1 tabulates the number of involvements for the situation above.

Table 1.1: *The number of times that a webpage is involved as outbound for the mutual web linkages for the linkage situation in Figure 1.1.*

Webpage	Number of involvements
w4	6
w5	5
w3	4
w7	4
w8	3
w1	2
w6	2
w2	0

2 Methodology

2.1 File Format of a Web Graph

The web graph text file is assumed to have the following format:

- Line 1 and 2 both start with # and contain free text
- Line 3 is of the form "# Nodes: integer1 Edges: integer2", where integer1 is the total number of webpages, denoted N, and integer 2 is the total number of links, denoted N_{links}
- Line 4 is of the form "# FromNodeId ToNodeId"
- Remaining file consists of a number of lines where each line contains two integers; the index of the outbound webpage and the index of the inbound webpage
- Each web link is uniquely listed
- Webpage indices start from 0 (C convention)
- Self-links may occur (same outbound as inbound) and these should be ignored
- If one of the nodes in an edge has an index that is larger or equal to the number of nodes, ignore it

For instance, the web graph file that contains the linkage information shown in Figure 1.1 is as follows:

```
# Directed graph (each unordered pair of nodes is saved once): 8-webpages.txt
# Just an example
# 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
```

Table 2.1 lists the web graphs to be analyzed in this project.

Table 2.1: *Web graphs that are analyzed in this projects. The vital statistics in this project are the number of nodes, N, and the number of edges, N_{links}. The last three web graphs can be retrieved from [1].*

Web graph	N	N_{links}	**Description**
8-webpages	8	17	Simple web graph for testing and verifying the implementation.
100nodes_graph	100	250	Slightly larger web graph than the one above. Also used to verify the implementation.
web-NotreDame	325 729	1 497 134	Web graph of University of Notre Dame from 1999.
web-Stanford	281 903	2 312 497	Web graph of Stanford from 2002.
web-BerkStan	685 230	7 600 595	Web graph of Stanford and UC Berkely from 2002.

2.2 Read and Store Web Graph as a 2D Table

A 2D table of dimension $N \times N$ is a convenient storage format for a web graph. The values in the table are either "0" or "1". If the value in row i and column j is "1", it indicates a direct link from webpage j (outbound) to webpage i (inbound). For example, the corresponding 2D table for the web graph in Figure 1.1 is:

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

The programmatic approach to store the web graph in this format is as follows:

1. Load the web graph file
2. Skip the first two lines
3. Extract the number of webpages N (integer1) from the third line
4. Skip the fourth line
5. Allocate a 2D table of dimension $N \times N$ initialized with zeros
6. Read rest of the file line by line:
   a) For each line, store the outbound index j and inbound index i
   b) If the table element (i, j) not is a self- or illegal link, set it as 1

2.3 Read and Store Web Graph in the CRS Format

A 2D table element is only set to 1 if there is a direct link. In cases where N is large and N_{links} is relatively small, most of the values in the 2D table will be zero. This can result in a huge waste of storage. By adopting the compressed row storage (CRS) format, the waste can be avoided. The idea is that two 1D arrays of integer values are enough to store the data. The *col_idx* array of length N_{links} stores, row by row, the column indices corresponding to all the direct links. The *row_ptr* array of length $N + 1$ contains the indices at which new rows start in *col_idx*. Usually there is also a val array of length N_{links} which contains the values, but since values are 1 if there is a direct link and zero otherwise, it is not needed [2].

The procedure for reading of the web graph is similar to the 2D table case, however, algorithms for extracting *row_ptr* and *col_idx* is needed. The following algorithms assumes that the arrays are allocated properly, illegal links taken into account.

Algorithm for finding *row_ptr*

1. Set the first entry to 0. The following entries are set by counting the number of direct links row by row.
2. Next, the indices at which new rows start in *col_idx* is found by the following cumulative summation
$$row\_ptr_i = row\_ptr_i + row\_ptr_{i-1} \quad \text{for } i \in [1, N]$$

For instance, the web graph in Figure 1.1 gives

$$row\_ptr = \{0, 1, 3, 1, 1, 3, 3, 2, 3\}$$

in step 1. The cumulative summation then yields

$$\begin{aligned}
row\_ptr_1 &= 0 + 1 = 1\\
row\_ptr_2 &= 3 + 1 = 4\\
row\_ptr_3 &= 4 + 1 = 5\\
row\_ptr_4 &= 5 + 1 = 6\\
row\_ptr_5 &= 6 + 3 = 9\\
row\_ptr_6 &= 9 + 3 = 12\\
row\_ptr_7 &= 12 + 2 = 14\\
row\_ptr_8 &= 14 + 3 = 17
\end{aligned}$$

such that

$$row\_ptr = \{0, 1, 4, 5, 6, 9, 12, 14, 17\}$$

Algorithm for finding *col_idx*

col_idx stores, row by row, the column indices corresponding to all the direct links. By using *row_ptr* to access *col_idx*, the following scheme can be utilized:

1. Store the (valid) inbound and outbound webpage indices in e.g. arrays *ToNode* and *FromNode*.

2. *row_ptr* can then be used to access the row corresponding to a direct link in *col_idx* by

$$col\_idx[row\_ptr[ToNode[i]]]$$

However, to ensure that no values in *col_idx* is overwritten, indices must be offset if already accessed. This can be achieved by allocating a counter array of length N and implementing the following procedure:

```
do i = 0, N_links:
    col_idx[row_ptr[ToNode[i]] + counter[ToNode[i]]] = FromNode[i]
    counter[ToNode[i]] += 1
enddo
```

With this procedure, the *col_idx* array for the web graph in Figure 1.1 becomes

$$col\_idx = \{6, 0, 2, 3, 0, 1, 2, 3, 6, 3, 4, 7, 4, 7, 4, 5, 6\}$$

2.4 Count Mutual Web Linkage Occurrences

Next on the agenda is to implement functions for counting the total number of mutual webpage linkage occurrences, as well as the number of involvements per webpage as outbound for such mutual linkage occurrences.

The approach for achieving this differ slightly with the different storage formats. In both cases a *num_involvements* array of length N, for storing the number of involvements per webpage, is assumed to be already allocated.

2.4.1 With 2D Table as Storage Format

With a 2D table as storage format the following procedure is adopted:

- Initialize a counter, C.
- For each row in the table:
  - Set $C = 0$
  - Traverse each column on the row. Add 1 to C if column j has a direct link
  - The number of involvements of the webpages is then found by traversing the columns on the same row again. If the table element in column k is 1, add $C - 1$ to the *num_involvement* array at index k
  - Calculate the contribution to the total number of mutual links. The summation formula is given by Pascal's triangle shifted down one row:

$$\sum_{c=1}^{C}(c-1) = \frac{1}{2}(C-1)C$$

2.4.2 With CRS as Storage Format

With the CRS as storage format the following procedure is adopted:

- Initialize a counter, C
- For $i \in [0, N)$
  - The counter for the current row can be extracted from the *row_ptr* array by the following
    $$C = row\_ptr_{i+1} - row\_ptr_i$$
  - The contribution to the total number of mutual links is again calculated by the formula given by Pascal's triangle shifted down one row:
    $$\sum_{c=1}^{C}(c-1) = \frac{1}{2}(C-1)C$$
  - By using *row_ptr* to extract a row in *col_idx*, the following indexing scheme can be used to tally the number of involvements to the *num_involvements* array:

    For $j \in [row\_ptr_i, row\_ptr_{i+1})$
    - $$num\_involvements_{col\_idx_j} = num\_involvements_{col\_idx_j} + C - 1$$

2.5 Ranking the Top Webpages

The next procedure to be implemented is to find the top n webpages with respect to the number of involvements in mutual linkages. This procedure is the same for both storage formats, and goes as follows:

- For $i \in [1, n]$
  - For $j \in [0, N)$
    - Find the index (corresponds to webpage node) with the maximum number of involvements in the *num_involvements* array
  - Store or print the result
  - Set the number of involvements to -1 for the recently found top webpage in the *num_involvements* array so that the same webpage is not found as top again

2.6 Parallelization

The code has been parallelized by using the following code structure:

```
#ifdef _OPENMP
  ...
#else
```

```
    ...
#endif
```

This allows for keeping the code both with and without OpenMP specifics in the same program.

For both procedures to count the mutual web linkages, the serialized and parallelized code are the same. They only differ in that the latter have a declared parallel OpenMP region. The pragma omp parallel is used to fork additional threads to carry out the work enclosed in the construct in parallel. To avoid false sharing, the counter variable is declared as private. The tallying of the total number of mutual links and number of webpage involvements must be summed, which requires the corresponding variables to be declared in a reduction identifier.

The algorithmic approach is the same in both the serialized and parallelized version of the ranking of top webpages. However, here a simple pragma declaration is not enough to parallelize the code. Instead, the work must be split manually between threads. With this approach, all threads find the top webpages within their respective part of the list. Procedures for checking that the whole list is ranked if it could not be divided fully amongst the team must also be in place. The list of top n webpages is in the end deduced from a list, of length $n\times$ number of threads, where each worker has stored their top results.

3 Results and Discussion

3.1 Read and Store Web Graph

Table 3.1 tabulates the benchmark results for the reading and storing web graph procedures.

Table 3.1: *Benchmark of reading and storing the web graphs. The table tabulates which web graph that was used, which program (or function), the number of function call repetitions, N_{rep}, and the elapsed time. The time is the elapsed wall clock time given by the omp_get_wtime routine averaged over N_{rep} function calls.*

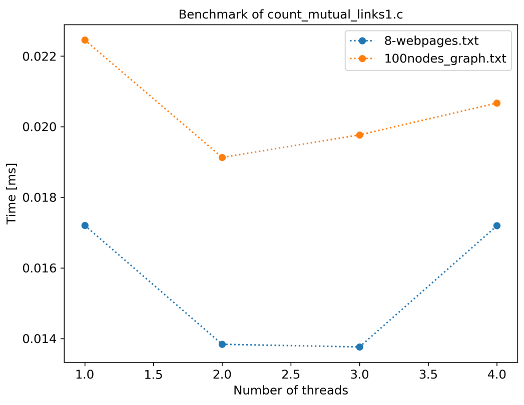
Web graph	Program	N_{rep}	Time [ms]
8-webpages	read_graph_from_file1.c	10 000	0.150754
8-webpages	read_graph_from_file2.c	10 000	0.181131
100nodes_graph	read_graph_from_file1.c	10 000	0.211546
100nodes_graph	read_graph_from_file2.c	10 000	0.236447
web-NotreDame	read_graph_from_file2.c	20	244.277152
web-Stanford	read_graph_from_file2c	20	415.670213
web-BerkStan	read_graph_from_file2.c	10	1231.467176

As seen in the table from the benchmarks on the smaller web graphs, the read and store method with CRS as storage format is slightly slower than the one with a 2D table. This is expected for smaller web graphs as the procedure of the former is slightly more complex. However, for large web graphs the 2D table format is unfeasible on most computers. The web-NotreDame graph, for instance, would require approximately 106 GB of memory if stored as a 2D table.

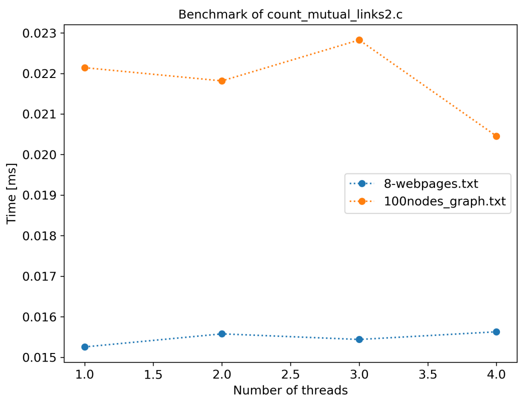
The time it takes to read and store larger web graphs, as seen in the table, indicates that this procedure will be the bottleneck when ranking the top webpages.

3.2 Count Mutual Web Linkages

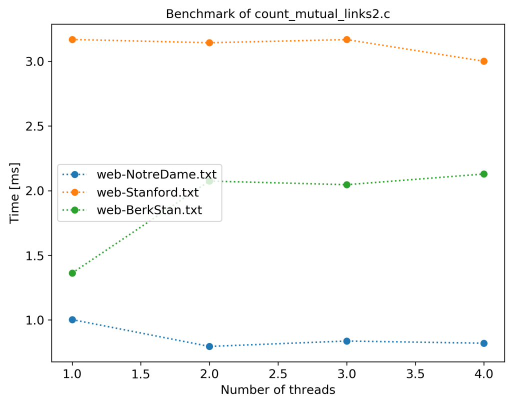
Figure 3.1 shows the benchmark results for the counting of mutual web linkage occurrences procedures.



(a)



(b)



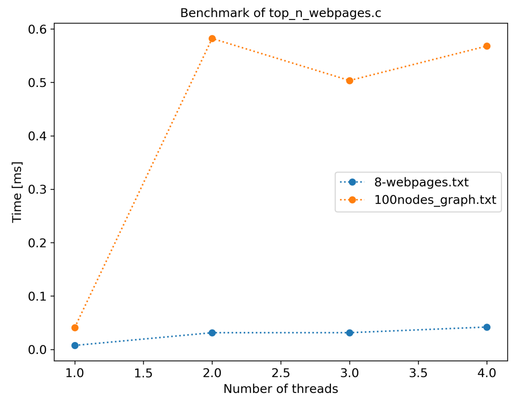
(c)

Figure 3.1: *Benchmark of the counting of mutual web linkages procedures with varying number of threads. The time is the elapsed wall clock time given by the omp_get_wtime routine averaged over* ***(a)*** *10 000,* ***(b)*** *10 000 and* ***(c)*** *100 function calls.*

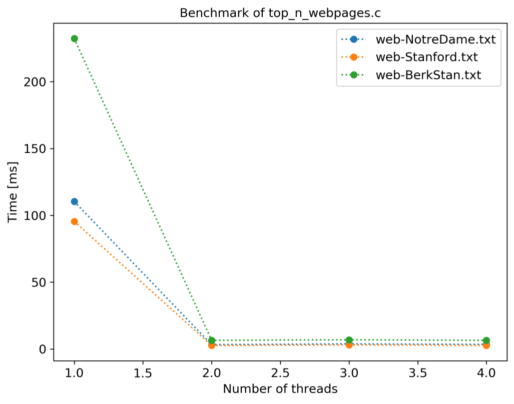
Figure 3.1 shows that there is a speedup with parallelization. The optimal number of threads, however, is not necessarily the maximum available, as it can be seen that idling threads may occur, i.e. overhead is encountered.

3.3 Top Webpages

Figure 3.2 shows the benchmark results for the ranking of the top n webpages procedure.



(a)



(b)

Figure 3.2: *Benchmark of the top n webpages procedure with varying number of threads. When there is only 1 thread, the serialized version of the code is run. Otherwise, the parallelized version is run. The time is the elapsed wall clock time given by the omp_get_wtime routine averaged over **(a)** 1 000 and **(b)** 100 function calls.*

Figure 3.2(a) shows that for the web graphs used to test and verify the implementation, there is much overhead. However, for the real-world web graphs in Figure 3.2(b) there is a substantial speedup with the parallelized code. This illustrates the importance of making sure that there is enough work in order to don't encounter overhead. Interestingly, the speedup achieved by only 2 threads in the latter case, is nearly identical to the speed achieved with both 3 and 4 threads.

4 Conclusion

Due to having a smaller memory footprint, using compressed row storage (CRS) method as storage format for the web graph is a preferable. When tallying the number of mutual web linkages and webpage involvements, this storage format is also superior as it is faster to access the data for larger web graphs. The procedures mostly benefit from parallelization, although only if there is enough work to divide among the team to ensure no overhead is encountered. However, the most time consuming part of ranking the top webpages in a web graph is to read and store the graph. Alas, the gain from parallelization in the overall procedure is not significant.

References

[1] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 3).

[2] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. 1st. USA: CRC Press, Inc., 2010. ISBN: 143981192X (cit. on p. 4).