

Home exam 1 - IN3200

Candidate nr: 15129
(Dated: March 18, 2020)

I. INTRODUCTION

In this report we look into the main algorithmic aspects of the code implementations and present time measurements of the serial and parallelized codes.

II. BACKGROUND

Webgraphs

Webgraphs describe hyperlinks between webpages i and j . Each webpage is represented by a node and the hyperlink between them is known as an edge. In this report I'll define N as the number of nodes and N_{links} as the number of edges in a *directed* webgraph. By directed, we mean that the hyperlink $j \rightarrow i$ is distinct from $i \rightarrow j$.

Hyperlink matrix representation of webgraphs

Webgraphs can be represented by what is called a hyperlink matrix A . An element $A_{ij} = 1$ if there exists a hyperlink $j \rightarrow i$. If no such link exists, $A_{ij} = 0$. In the implementation described in this report, we allow no self-links $i \rightarrow i$ such that $A_{ii} = 0$ for all i .

Compressed row storage (CRS) of the hyperlink matrix

For large webgraphs, it's expected that the hyperlink matrix will mainly consist of zeros. Since the storage in the matrix format scales with N^2 , storing the data becomes a bottleneck. The CRS format is based on two different arrays. There's the *row pointer* which stores information about how many row elements the webgraph has. Let r denote the row pointer array. The following equation shows the general form of this array.

$$r = (0, r_0, r_0 + r_1, \dots, r_0 + r_1 + \dots + r_{N-1}), \quad (1)$$

where r_i is the number of row elements of row i . The row pointer thus has length $N + 1$. To extract how many row elements there are on row i , we take the difference $r_i - r_{i-1}$. The second array is called a *column index* array and is of length N_{links} and stores the column indices for each row.

Mutual web linkages and number of involvements

Mutual web linkages is defined as the number of times to outbound nodes i and j are directly linked to a inbound node k , with $i \neq j \neq k$. That is, it's a count of how many times there exists a hyperlink $i \rightarrow k$ and $j \rightarrow k$ simultaneously. The *number of involvements* a given node has is defined as the number of times a given node is involved as an outbound node. That is how many times a node i is involved in $i \rightarrow k$ with $j \rightarrow k$.

III. IMPLEMENTATION

Reading the Webgraph from file

1. Hyperlink matrix storage

To extract the data to store in a hyperlink matrix is straight forward. The following code snippet shows how I did it.

```
int FromNodeId, ToNodeId;
for (int k = 0; k < N_links; k++){
    fscanf(fp, "%d %d", &FromNodeId, &ToNodeId);
    (*table2D)[ToNodeId][FromNodeId] = (char) 1;
}
```

2. CRS storage

To extract the data from the file into CRS storage, we must first sort the arrays. I chose to sort this using the *shellsort* algorithm with $\text{gap} = N/2$. The following code snippet shows the implementation.

```
int tmp1, tmp2, i, j, gap;
for (gap = *N_links/2; gap > 0; gap /= 2){
    for (i = gap; i < *N_links; i++){
        tmp1 = row_elems[i];
        tmp2 = (*col_idx)[i];
        for (j = i; j >= gap && row_elems[j-gap] >
             tmp1; j -= gap){
            row_elems[j] = row_elems[j-gap];
            (*col_idx)[j] = (*col_idx)[j-gap];
        }
        row_elems[j] = tmp1;
        (*col_idx)[j] = tmp2;
    }
}
```

Here, `row_elems` is just there to temporarily store row node ids. The row pointer is simply made using an array counting how many elements each row has consistent with eq. (1). The following code demonstrates this.

```

*row_ptr = (int*)calloc(*N+1, sizeof(int*));
int count = 0;
for (int i = 0; i < *N; i++){
    count += row_count[i];
    (*row_ptr)[i+1] = count;
}

```

Counting mutual web links

3. Counting mutual web links with the hyperlink matrix

To count the number of web links that each node is an outbound participant is in principle a simple matter when we use the hyperlink matrix representation. It is simply to check when $A_{ij} = A_{ik} = 1$ with $i \neq j \neq k$. However, to make the code implementation efficient, it's important to do this economically. The algorithm goes as follows: For a given node i , we check every j and only if $A_{ij} = 1$ should we count the mutual linkages and number of involvements. One way to do this is to insert an if test and check if $A_{ik} = 1$ and add 1 to both variables. However, since A_{ik} is either 0 or 1, we might as well just add the matrix element itself and avoid the if-test entirely. The following code demonstrates how this can be implemented. It also counts the total number of web linkages which is just the total number of times mutual web links occur.

```

int counter;
for (i = 0; i < N; i++){
    for (j = 0; j < N; j++){
        counter = 0;
        if (table2D[i][j] == 1){
            for (k = j+1; k < N; k++){
                counter += table2D[i][k];
                num_involvements[k] += table2D[i][k];
            }
            num_involvements[j] += counter;
            total_mutual_web_linkages += counter;
        }
    }
}

```

4. Counting mutual web links with the CRS format

To count the total number of web links in this format, we can reduce the number of floating point operations by computing the sum analytically. To find the contribution a given row i has to the total number of web linkages, let's first define m_i to be the number of elements on row i such that $m_i = r_i - r_{i-1}$. Then

$$S = \sum_{k=0}^{m_i-1} k = \frac{m_i(m_i-1)}{2}, \quad (2)$$

where the standard formula for the sum of the m_i-1 first integers were used. The following code snippet shows the implementation.

```

for (int i = 0; i < N; i++){
    tmp = row_ptr[i];
    row_elems = row_ptr[i+1]-tmp;
    for (int j = 0; j < row_elems; j++){
        num_involvements[col_idx[j+tmp]] +=
            row_elems-1;
    }
    total_mutual_web_linkages += (row_elems)*(
        row_elems-1);
}
total_mutual_web_linkages *= 0.5;

```

As can be seen from the code, a tmp variable is defined to avoid loading row_ptr[i] in j-dependent loop. In addition we use the sum formula instead of computing the actual sum for each i manually.

Finding top webpages

First of all, by top webpages we mean the webpages or nodes that are most involved as an outbound node in mutual linkages. To find the n top webpages in a webgraph, I've used shellsort again, but this time sorting in descending order. For small n this isn't the wisest choice since the algorithm is only dependent on how many nodes there are. However, for larger n 's it's an efficient choice. The code implementation is as follows.

```

int gap, i, j, tmp1, tmp2;
for (gap = num_webpages/2; gap > 0; gap /= 2){
    for (i = gap; i < num_webpages; i++){
        tmp1 = num_involvements[i];
        tmp2 = webpage_number[i];
        for (j = i; j >= gap && num_involvements[j]
            -gap < tmp1; j -= gap){
            num_involvements[j] = num_involvements[j]
                -gap;
            webpage_number[j] = webpage_number[j-gap]
                ];
        }
        num_involvements[j] = tmp1;
        webpage_number[j] = tmp2;
    }
}

```

Parallelization of count_mutual_links1

To parallelize this function is straight forward by applying the following OpenMP directive on the outer loop

```

#pragma omp parallel for private(i, j, k)
reduction(+: total_mutual_web_linkages,
num_involvements[:N])

```

This is because we need to make the indices i , j and k private to each thread. To handle the possibility of race conditions when reading and updating the total_mutual_web_linkages and elements in num_involvements, we must also use the reduction clause.

Parallelization of count_mutual_links2

This function is also readily parallelizable, but also here we must avoid race conditions. Here, however, we can use the `#pragma omp atomic` directive instead (simply because it provided more speedup than using the reduction clause on the array elements). The following code shows the parallelized version.

```
int tmp, row_elems;
#pragma omp parallel for private(tmp,
    row_elems) reduction(+:
    total_mutual_web_linkages)
for (int i = 0; i < N; i++){
    tmp = row_ptr[i];
    row_elems = row_ptr[i+1] - tmp;
    for (int j = 0; j < row_elems; j++){
        //Insert atomic to avoid race conditions
        when updating num_involvements.
        #pragma omp atomic
        num_involvements[col_idx[j+tmp]] +=
            row_elems-1;
    }
    total_mutual_web_linkages += (row_elems)*(
        row_elems-1);
}
```

IV. RESULTS

Timing of serial codes

The measured time of the serial implementations of the various functions are shown in table I.

Function name	Time in seconds
read_graph_from_file1	0.179096
count_mutual_links1	1.799104
read_graph_from_file2	0.356536
count_mutual_links2	0.001885
top_n_webpages	0.019348

TABLE I. The table shows the measured time using `clock()` from the Ctime-library. `read_graph_from_file1` and `count_mutual_links1` was applied to a web-graph containing $N = 5000$ nodes and $N_{\text{links}} = 146823$ edges as found in the file `test_webpages.txt`. The data in this file was extracted from `web-NotreDame.txt`. `read_graph_from_file2` and `count_mutual_links2` was applied directly to the web-graph contained in `web-NotreDame.txt`. This file contained $N = 325729$ nodes and $N_{\text{links}} = 1479143$ edges.

Parallelized version of count_mutual_links1

Using OpenMP to parallelize `count_mutual_links1` gave the time measurements shown in figure 1.

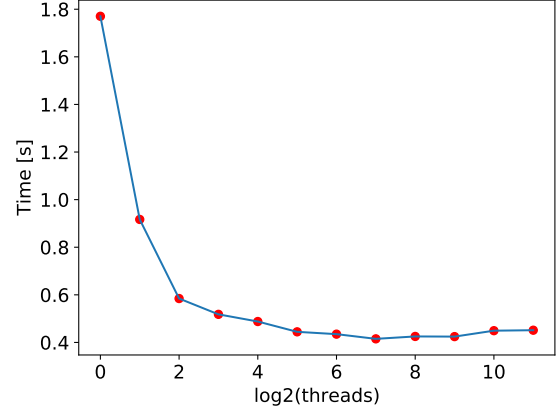


FIG. 1. The figure shows the time used in seconds by `count_mutual_links1` on a webgraph consisting of $N = 50000$ nodes and $N_{\text{links}} = 146823$ edges. The webgraph was extracted from `web-NotreDame.txt`. The red points show the actual measured datapoints.

The speedup relative to one thread is shown in figure 2.

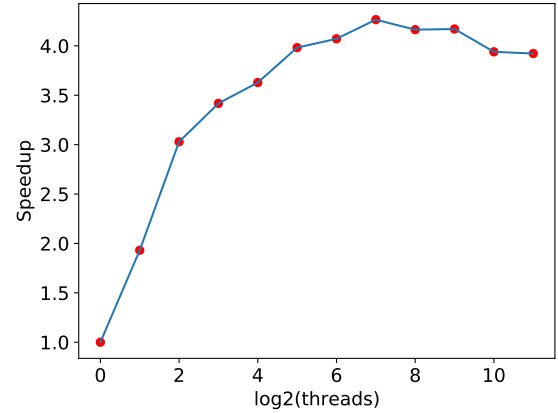


FIG. 2. The figure shows speedup relative to a single thread by `count_mutual_links1` on a webgraph consisting of $N = 50000$ nodes and $N_{\text{links}} = 146823$ edges. The webgraph was extracted from `web-NotreDame.txt`. The red points show the actual measured datapoints.

Parallelized version of count_mutual_links2

Using OpenMP to parallelize `count_mutual_links2` and measuring the time used by the function for different number of threads yielded the results shown in figure 3.

V. CONCLUDING REMARKS

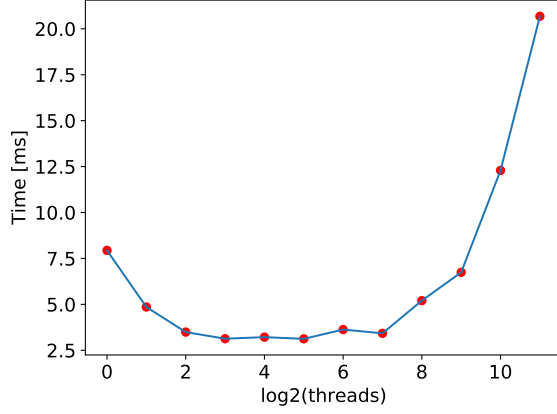


FIG. 3. The figure shows the time used in seconds by `count_mutual_links2` on a webgraph consisting of $N = 325729$ nodes and $N_{\text{links}} = 1479143$ edges. The webgraph is found in the file `web-NotreDame.txt`

The speedup relative to a single thread is shown in figure 4.

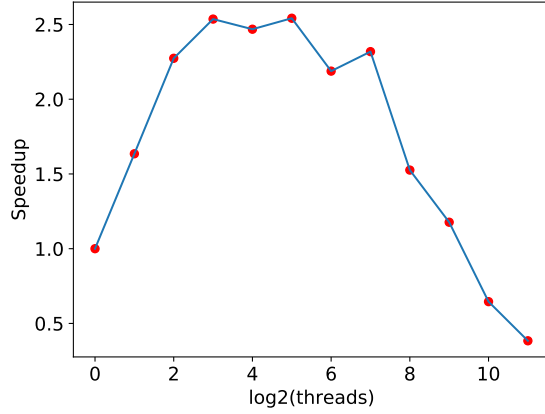


FIG. 4. The figure shows speedup relative to a single thread by `count_mutual_links1` on a webgraph consisting of $N = 325729$ nodes and $N_{\text{links}} = 1479143$ edges. The webgraph is from the file `web-NotreDame.txt`. The red points show the actual measured datapoints.