IN4200 Project1

Candidate Nr. 15238

April 2020

1 Introduction

A few explanations of the algorithms implemented for the first project in IN4200 at the University of Oslo are presented. The time measurements of the OpenMP parallelised functions for various OpenMP threads are also included. To compile the file `main.c`, the following command is recommended:

```
gcc main.c -o test.x -fopenmp -O3
```

And the following command is recommended to subsequently run the executable `test.x`:

```
./test.x -d
```

This indicates to the program that you would like to use the data directory `data/web-NotreDame.txt`. Alternatively, you can insert your own data directory:

```
./test.x YourDirectory
```

To test the function for errors with regards to some testing data, run the executable using the `[-t]` flag:

```
./test.x -t
```

This requires specific testing data which has well defined results to be in the directory `data/testingdata.txt`.

2 Exercise Implementations

Miscellaneous

The functions were implemented using some miscellaneous functions. These were functions which were either used often or were unappealing in code (e.g. nested loops or using many lines). A central aspect of the project is having good user command line arguments for execution of the code. One of the miscellaneous functions implemented was titled `arg_parser`, and was designed to handle any flags and command line arguments. The flags implemented are as follows:

- `[-t]` - A flag which initializes the testing utilities of the functions. This is mostly used for debugging and function assertion purposes. The program tests using the `data/testingdata.txt` data directory.
- `[-d]` - A flag which tells the program to use the default data directory `data/web-NotreDame.txt`.
- `[-p]` - A flag which indicates to the program that you would like to parallelize the processes using *OpenMP*.

The argument parser function expects that you pass a filename in as a command line argument. Exceptions are when the `[-t]` or `[-d]` flags are passed. The function also raises an error if there is more than one filename (or unexpected flag) which is passed into the command line.

Exercise 3.1: Reading a web graph from file

1. `read_graph_from_file1.c`

The first function was made to create a matrix containing the website linkages. This was as simple as adding the value 1 to specific locations in the `table2D` array. The locations of the ones are dictated by the `FromID` and `ToID` values from the file which is read in. The function implemented writes in the data as `table2D[ToID][FromID] = 1` for each of the linkages. The function only does this if the values are different and not dead. A dead link is if ether the To- or From ID is has a larger value than the total number of nodes.

2. `read_graph_from_file2.c`

The second function was made to store the website linkages and store them in a CRS format. This algorithm was the most tricky to implement and optimize, as the `col_idx` array is difficult to build efficiently. Firstly, the `ToID` and `FromID` `row_ptr` arrays are built by reading the file in a similar fashion as previously (avoiding self-linkages and dead-links). The `row_ptr` array simply counts how many other nodes link to the current node, so with each valid linkage, the element `row_ptr[ToID+1]` (the plus one is due to convention) is incremented +1.

A quick cleanup is necessary after this to format the `row_ptr` array and to allocate less space. Due to dead- and self-links, the `N_links` value which is passed in overestimates the storage space which is needed.

To build the `col_idx` array, we should only need one loop over all of the `FromID` arrays, and be clever enough to place them in the right place in the memory. I thought of this in terms of the `table2D` array from the previous exercise, where the `col_idx` array tells us which column the value is. The other information which we have is the `row_ptr` array, which tells us how many elements there have already been before the current row. So, to add a new element which is in row i into the `col_idx` array, we need to be careful to place it in the `row_ptr[`i`]`th spot.

Then the only information which is needed is how many elements there have already been previously in the current row. This is needed to add elements in between the indices `row_ptr[`i`]` and `row_ptr[`$i+1$`]`. For this, there was a new array created called `prev_row_ids`, which has length N. For each time an element was added to row i, the element `prev_row_ids[`i`]` was incremented. This assures that each element in the `col_idx` array is eventually filled with a value.

Exercise 3.2: Counting Mutual web linkages

1. `count_mutual_links1.c`

For this exercise, the mutual links were not too complex. One simply needed to formulate the problem in a computational graph sense. Mutual links are essentially two 1's that share a common row. The algorithm was then simple to implement.

Have an outermost loop through all the columns j. For each of these columns, loop through all the row elements i. If, for any combination `table2D[`i`][`j`]` equals 1, then loop through the columns again and count all the other ones in that row. Any additional 1's in that row that are not in row j will be counted as mutual links to node j.

2. `count_mutual_links2.c`

Much of the same philosophy was used in the second implementation of the function. Only this time, the CRS formatted arrays had to be used. What is important once again is formulating the problem in a more simple sense, relating to the `row_ptr` and `col_idx` arrays.

The `row_ptr` array tells us how many non-zero elements there are from one row to another. In that sense, if `row_ptr[`$i+1$`]` - `row_ptr[`i`]` is larger than one, we know that there is at least one mutual linkage in row i. We simply need to extract which columns are linking to each other. This is where the `col_idx` array becomes useful. We know this array is dictated by `row_ptr` array. The `row_ptr` array describes how many elements there have already been, such that `col_idx[row_ptr[`i`]]` should return the column index of the first element in row i.

This is why a new loop is set for j to loop through all elements between `row_ptr[`i`]` and `row_ptr[`$i+1$`]`, as `col_idx[`j`]` gives us all the columns which are mutually linked in row i.

Exercise 3.3: OpenMP parallelisation of the two `count_mutual_links` functions

1. `count_mutual_links_OpenMP1.c`

This exercise was accomplished by using the

```
#pragma omp parallel for reduction(+:total_links, num_involvements[:N])
```

command. This indicates to the `omp.h` library that the for loop jobs should be evenly distributed to all available nodes. The reduction argument causes all nodes to be able to contribute to the counting of the total links variable. This argument also indicates to the arrays that there is cooperation in manipulating the `num_involvements` array. If not for the `num_involvements[:N]` addition, some strange errors can appear.

Otherwise, the code is pretty much identical to the unparallelized version.

2. `count_mutual_links_OpenMP2.c`

Similarly here, the parallelized implementation implemented an additional `omp.h` command line prior to the for-loop which states:

```
#pragma omp parallel for reduction(+:total_links, num_involvements[:N])
```

The line accomplishes the same functionality as described previously.

Exercise 3.4: Finding top webpages regarding involvements in mutual linkages

These exercises

1. `top_n_webpages.c`

The `max` function designed found each element of the top n webpages array individually. This is based on a minimalist approach, where the whole `num_involvements` array is looped through to find the maximum value. This is done for all top n webpages. If an element is picked out to have the maximum value, then it is set to -1. This is to assert that the webpage is not picked out again on other iterations. Following is a pseudo-code of the function:

```
for $i$ in top_n:
—       maxidx, maxval = max(num_involvements)
—       num_involvements[maxidx] = $-1$
—       top_n[$i$] = maxval
```

2. `top_n_webpages_OpenMP.c`

The work was distributed amongst the nodes in the following fashion:

Given the number of webpages is $N_w = 14$, the number of nodes we have is $N_n = 4$, and we want to find the top $n = 6$ webpages, the algorithm would aim to distribute the work in the following fashion.

- Calculate the integer division $N_w/N_n = 3$ (disregarding the 'rests').
- Have each node loop through their respective section of the webpages array, and build their own top 6 webpages array using the `max` function described previously:
  - Node 1 loops through $[0, 3)$,
  - Node 2 loops through $[3, 6)$,
  - Node 3 loops through $[6, 9)$,
  - Node 4 loops through $[9, 12]$,
- This leaves us with four candidate arrays for the top 6 webpages as well as any rests N_w **mod** $N_n = 2$
- Finally, using one node, loop through all 4 top 6 webpages arrays, as well as the rests values using the `max` function.

Exercise 3.5: Test program

The testing function was implemented in two batches for readability. Both functions are contained within the same `testing_functions.c` file. The testing functions have utilized the data in the directory `data/testingdata.txt`, which should contain the following text file:

```
# 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
```

The testing function simply consisted of the 'true values' of the various arrays (such as the `table2D`, `col_idx` and `row_ptr` arrays), and compared them to the arrays produced by the functions. If any values were found to mismatch, then the testing function would count that as an error. Finally the function returns the total amount of errors found.

3 Timing analysis

The program was altered slightly (an additional loop at the beginning) to calculate several times of execution. The top 100 webpages were found 100 times using both the parallelized- and the regular code.

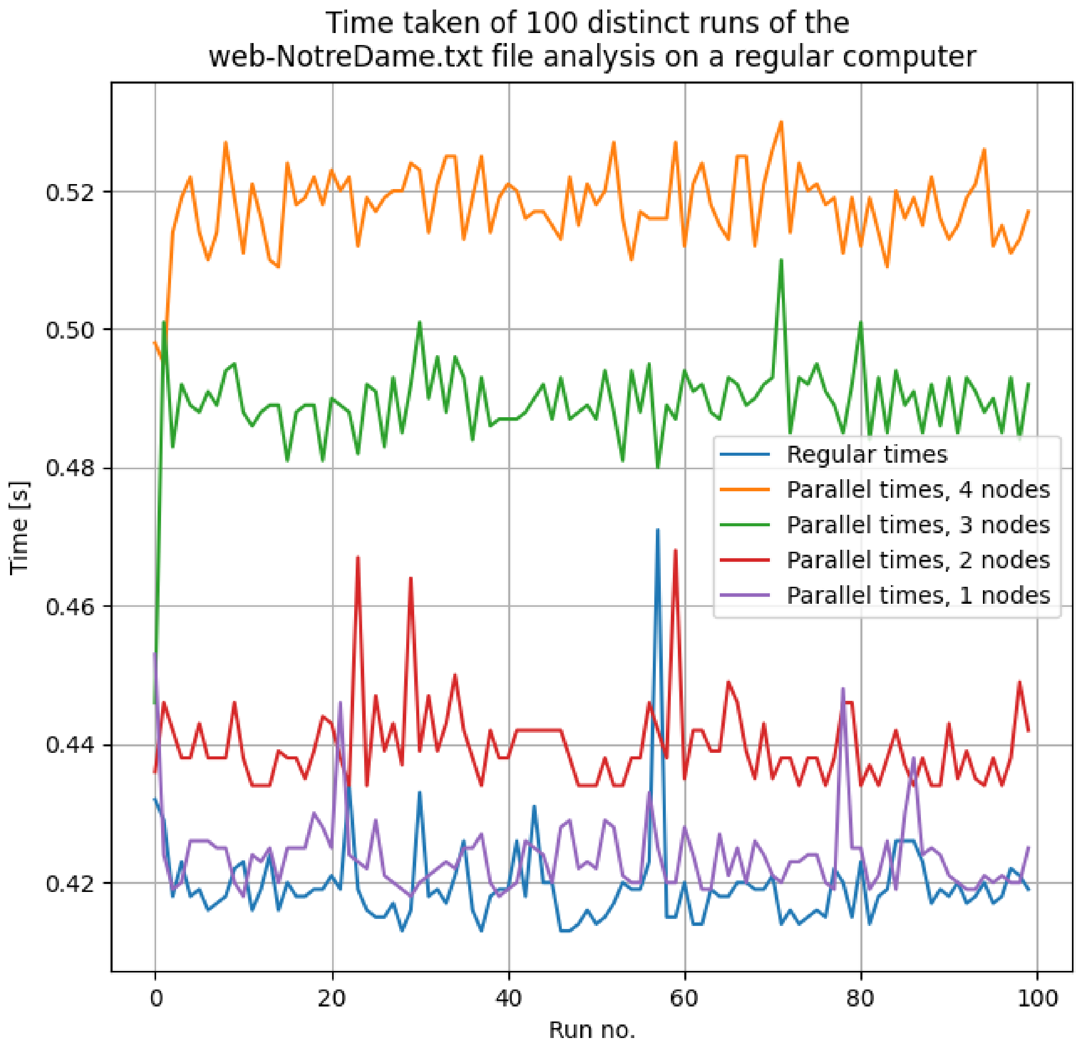

Figure 1: 100 run times of the top 100 webpages of the `data/web-NotreDame.txt` data directory. This analysis was done on a laptop computer. The mean-values and standard deviations of these runs can all be found in table 1

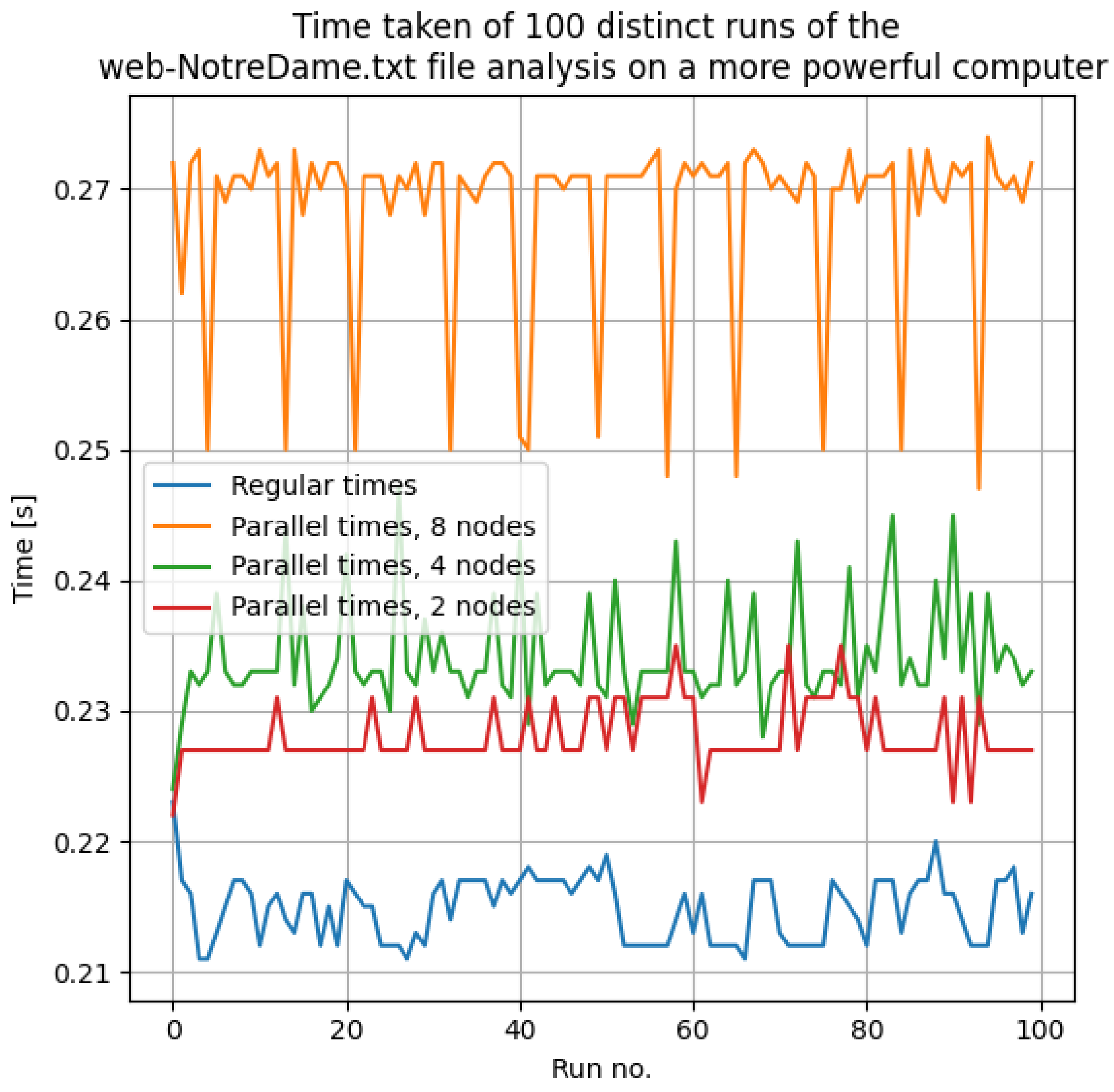

Figure 2: 100 run times of top 100 webpages of the `data/web-NotreDame.txt` data directory. This analysis was done on a desktop computer, resulting in the one node execution being nearly half as quick as one node execution in figure 1. The mean-values and standard deviations of these runs can all be found in table 1

Run description	Mean time [s]	Standard deviation [s]
Regular Computer		
Without parallelization	0.420	0.007
Parallelized, one node	0.424	0.006
Parallelized, two nodes	0.440	0.006
Parallelized, three nodes	0.489	0.006
Parallelized, four nodes	0.518	0.006
More Powerful Computer		
Without parallelization	0.215	0.002
Parallelized, two nodes	0.228	0.002
Parallelized, four nodes	0.234	0.004
Parallelized, eight nodes	0.268	0.007

Table 1: Mean- and standard deviation values of the times taken.

4 Conclusion

The code execution was found to be incredibly fast with the algorithms implemented. There are several ways of treating the exercises at hand which are far less efficient than the ones presented, and those were certainly all explored. It was quite surprising that more nodes resulted in a slower analysis. This may be due to the inherently short execution time of one node. Complicating that process with more nodes seems to slow it down. It may be that if the initial one node configuration took longer, that the parallelized speed-up may be noticeable.

There was a strange bandwidth pattern that showed up in the eight node configuration of figure 2. This caused this time to have one of the largest time standard deviations of $0.007s$, tied only with the laptop computer without parallelization run, which also had strange spikes.