IN4200 Home Exam 1

Bruce Chappell: Candidate 15209

14 April 2020

1 Introduction

In this project we will explore different methods for reading web graph files and extracting quantities of interest from web graph files. These two methods are based on full matrix storage and CRS storage of the data. The quantities we are interested in are total mutual links and the number of mutual links each page is involved in as an outbound page.

2 Method

2.1 Full Matrix Storage

For this method a $M_{N\times N}$ matrix is used to store the web linkages where N is the number of web pages in the graph. $M_{i,j} = 1$ when node j is an outbound link to node i and 0 otherwise. The general algorithm goes as follows:

1. Count the number of non zero entries per row i
2. Add $non\_zero\_entries - 1$ to the number of involvements for each non zero j in row i
3. Add $non\_zero\_entries - 1$ to $mutual\_links$
4. Repeat for each row
5. Divide final $mutual\_links$ by two since we counted links for $i \to j$ and $j \to i$

To parallelize this algorithm, we add a `#pragma omp parallel for` to the outer row loop. This divides the work by chunks of rows for each worker. We need to make use of the `reduction+:` to get the sum of $num\_involvement[:N]$ and $mutual\_links$ between the threads.

2.2 CRS Storage

Here we allocate space for a row pointer of size $N_{pages} + 1$ and a column index pointer of size N_{links}. The row pointer starts at zero and $row\_ptr_{i+1} - row\_ptr_i$ tells you how many links are on row i. While the *col_idx* pointer gives you the column index values of the corresponding links on row i. The counting algorithm goes as follows:

1. Save $row\_ptr_{i+1} - row\_ptr_i$ as the number of involvements
2. loop from row_ptr_i to $row\_ptr\_i + 1$ and add the previously calculated involvements to each corresponding column node involved.
3. Add number of involvements to *mutual_links*
4. Divide final *mutual_links* by two since we counted links for $i \to j$ and $j \to i$

As with the full matrix storage method, to parallelize this we need to add `#pragma omp parallel for` to the outer row loop to split the rows up among workers. Again, we use `reduction+:` to get the sum of variables across threads.

2.3 Counting Top Webpages

To find the top N webpages in terms of mutual links involved in, we implement the following algorithm:

1. Create an array of zeros of size N to store top values
2. Loop over list of number of involvements
3. Loop over top webpages array
4. If a values in *num_involvements* is greater than a value in *top_n*, move the trailing values in *top_n* one space down and insert the *num_involvements* value.

This method is not as simple to parallelize as the previous two. To achieve this, we split the data up by the amount of threads and each thread finds its corresponding *top_n* values for its chunk. Then outside of the parallel block we find the "total" *top_n* values by repeating the sorting algorithm but with the sets of *top_n*'s from each chunk as the data set.

3 Results

The following results were obtained using the `p2p-Gnutella30.txt` file from `https://snap.stanford.edu/data/index.html`. This data set has 36682 nodes and 88328 links. The machine used was a 2019 Mac-Book Pro with an Intel Core i5 processor and 8GB of RAM.

	Read(s)	Count(s)	Top N(s)
Full Matrix	2.957	4.408	6.69×10^{-4}
CRS	6.793	6.88×10^{-4}	6.39×10^{-4}

Table 1: Run times for serial implementations of counting algorithms.

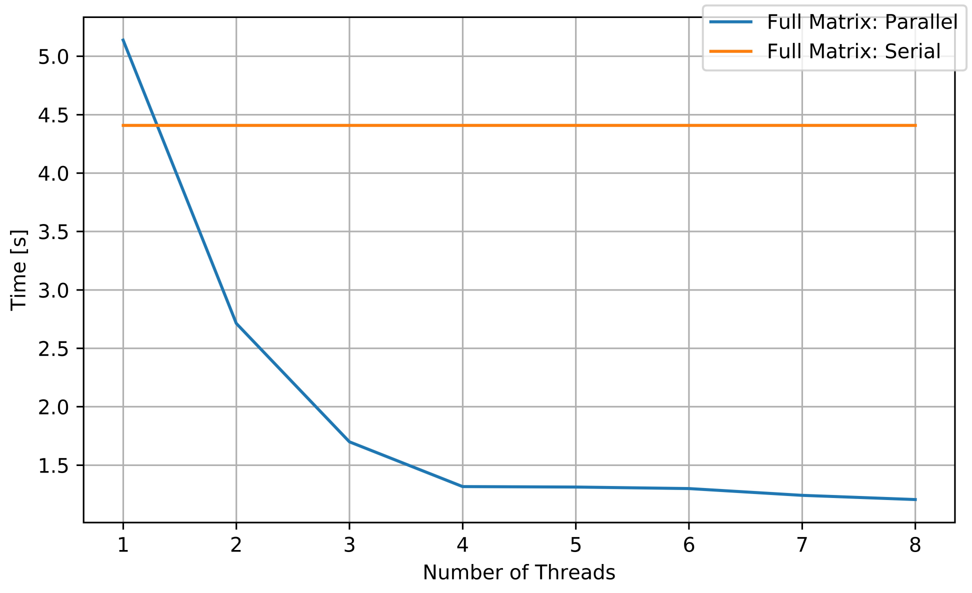

Figure 1: Run time results for the Full Matrix implementation of the counting algorithm.

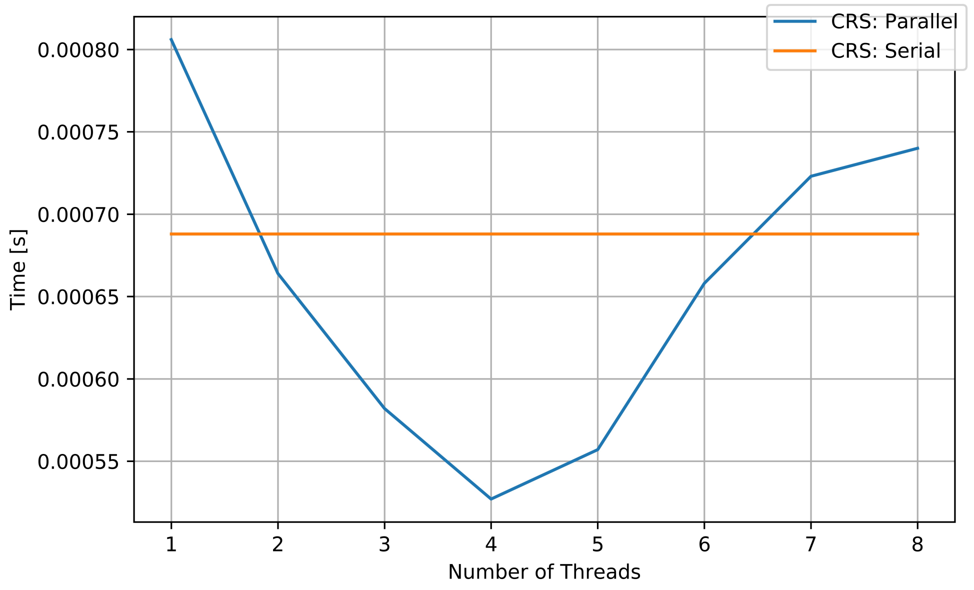

Figure 2: Run time results for the CRS implementation of the counting algorithm.

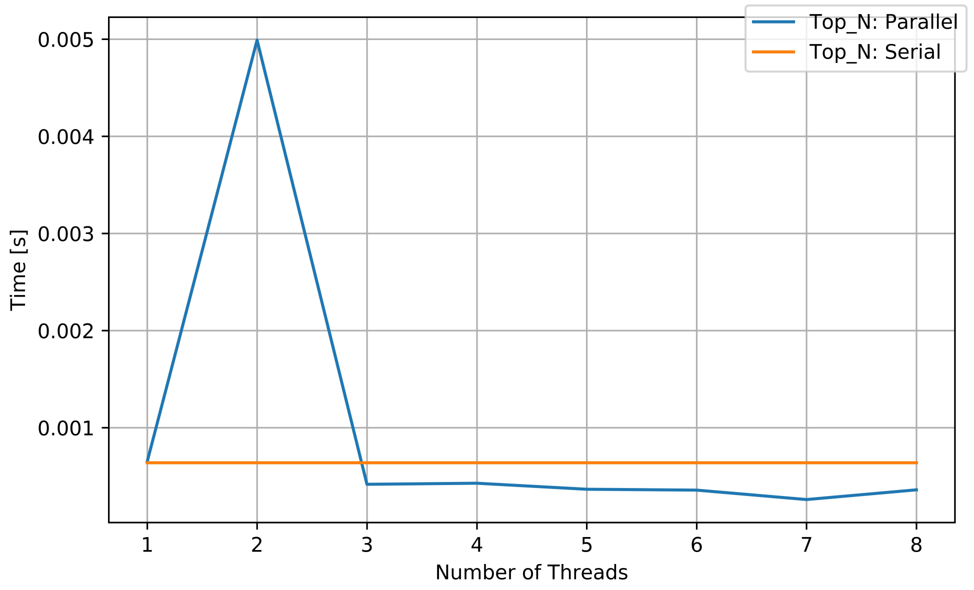

Figure 3: Run time results for the Top N webpages counter.

3.1 Discussion and Conclussion

Beginning with Table 1, there is a serious increase in the performance of the counting algorithm when switching from full matrix storage to CRS storage with the later being about ~ 6395 times faster than the former. As to be expected, the *top_n* counter takes about the same time since it functions independently of storage method. One thing to note is that the read time increase by a factor of 2.3 when using the CRS method.

Figure 1 shows the effects of parallelizing the full matrix counting algorithm. These results appear to show what one might expect when running a program in parallel; with one thread the time is roughly that of the serial implementation and as threads increase the run time drops off. Here we see an exponential looking dropoff that levels off around 4 threads.

Figure 2 shows the effects of parallelizing the CRS counting algorithm and we see a trend noticeably different than the one seen in Figure 1. The performance gradually increases until the computer is using four threads, and then gradually increases as you approach eight threads. This trend is not what I expected to see and could be explained by cache mismanagement or the hardware of the machine.

After initial instability, Figure 3 shows that the *top_n* algorithm relaxes to a run time of about 0.0005s. This runtime remains steady as thread number is increased.

In conclusion, for the selected experimental data set and machine used, running the programs in parallel with 4 threads appears the give the best results overall. However, if you only want to run the full matrix version, 8 threads would give slightly better performance. The full matrix version is limited by the `malloc` function in C and therefore cannot handle larger datasets. Therefore, if you have a large dataset, running the CRS version with 4 threads would be optimal.