IN3200 Counting mutual web linkage occurrences

15120
Institue for Physics
University of Oslo
Norway
April 14, 2020

I. READ GRAPH FROM FILE 1

Initially a skip lines function is defined, which loops through each caracter of a line until *nn*n or EOF for a given number of lines. This is simply a useful tool to skip lines, since we know there are a few lines in the beginning which are not useful for us.

The main read graph form file function skips the first two lines then extracts the number of nodes and edges from the file, before skipping yet another line.

The 2D array is allocated using calloc, so that all elements are zero, and then a loop over all number of links use the indexes in the file to set each index with a link to 1.
the file is then closed.

II. READ GRAPH FROM FILE 2

At the top of the file a struct called pairs is made, this will be used later to pair from and to idexes given by the file.

A function pair comp is also made. This is used with qsort later to sort the lists by both collumn and row! The 'to' elements are sorted first, however if the 'to' elements are the same the 'from' elements will be sorted aswell. This function is not used until the call to qsort on line 60.

The main function does the same as read graph form file 1 as in it opens and skips the two first line using the skip lines function located in the file for read graph from file 1. It then extracts the edges and nodes and skips the remaining lines until there is only data left.

at this point a struct called 'pairs' is made with calloc. This struct basically pairs the row and column index of each line, so that these stay together when sorting. The space allocated is enough to store the whole array of row and column elements.

a new array called row num stores the number of elements on a given row. This is then later used to make the condensed row storage since the first element on a given row is stored as a separate value. A loop over the number of all links is then started, which reads each line, and stores the indices from the file to the struct pairs. Since the right column 'to' in the file is the row this is used to index the number of elements in a given rows on line 54. I also count the number of self links, that's links that just goes to itself.

I then use the function qsort to sort throught the struct using the function pair comp described earlier. Since the function is meant to return a col and row array i allocate these, and find first the row array, which is simply cummulative number of elements of each row. the column elements are simply the column index of each element, however since we have a number of links which goes to itself i.e. self links we want to exclude these in the final col array.

In the end the i assign the row and col arrays to row ptr and col idx, since these are the arrays that are passed to read graph from file 2 by reference.
I then free the arrays i allocated in the function.
I ensured no arrays are left allocated using valgrind, so that no leaks or lost blocks are left.

III. COUNT MUTUAL LINKS 1

Since this code was parallelized it is all enveloped in an if else block checking if openMP is defined. This means when the Makefile compiles the 'single.exe' version which does not include -fopenmp this if check will render false (NOTE, chech readfile for info on compiling and running

this code).
For the parallelized code version, i use two for loop which loop throught collumn and row throught the whole matrix. These are preceded with a pragma omp for reduction call, which indicates to openMP that there is a summation to be done on num involvments. Inside theese i have a check to see if the given element is 1, i.e. if there is a link there. if this is the case it goes to yet another for loop which loops though the remaining elements in the row and checks how many ones there is. There isn't a specific if block to see if there is a 1, but by adding each element to the num invlovements array this is taken care of.

A very similar code is to be found in the else section of the initial if check however this does not include the pragma omp reduction call.

IV. COUNT MUTUAL LINKS 2

As with count mutual links 1 this code is made to work with both openMP and without it, as the code is duplicated in an if else block where on has the pragma omp for reduction initialization.
The main code loops over the number of webpages, which is also the number of rows. The codes extracts the number of elements on the given row, which is simpy the row ptr for one row, minus the previous row since row ptr is cumulative number of elements. the number of involvments for a given column index is then the number of row elements minus one.
The same codes goes for the un parallelized version, which is simply the same code but without the pragma omp call.

V. TOP N WEBPAGES

Now that we have the number of involvements listed from count mutual links we simply have to sort this list in order of number of involvements. We also need to keep track of the webpage number, or index, since this is the only way for us to keep track of what webpage is which.
Again i use a struct to pair webpage index, or webpage number if you like, and the number of involvements, which we know from count mutual links. A similar sorting function is used as in count mutual links 2, however this one simply sorts based on number of involvments.
The code itself is simply a loop over all webpages assigning webpages and number of involments together in the struct, then a call to qsort as earlier to sort the given arrays.

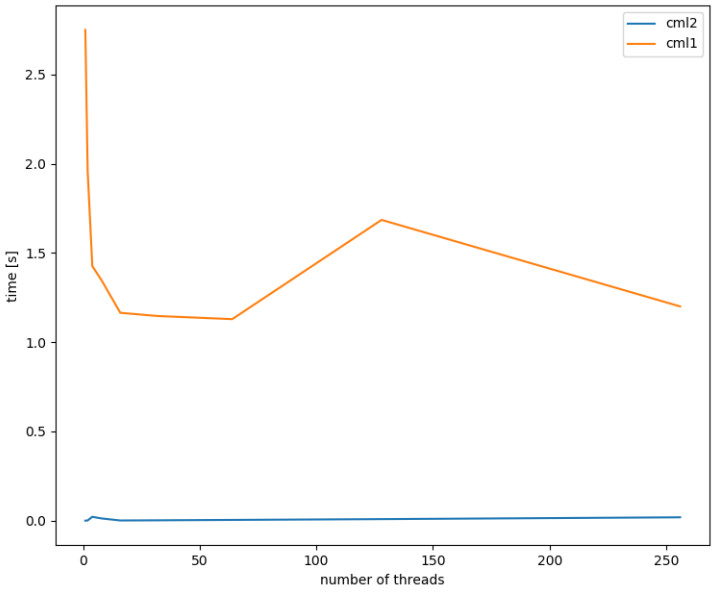

Figure 1. both count mutual links 1 and 2 parallelization using an increasing about of threads with the web-NotreDame datafile cut down to 50 000 webpages. This is because count mutual links 1 can't process the whole web-NotreDame data file

VI. TESTFILE

The test file is essentially just a program like main, however the time it takes for each function call is included, which also takes into a count wether openMP is paralelized the code. As per the problem set it also takes the filename of the data file that is read from the command line. As of right now i don't do much with the times recorded. In order to generate the figures seen below i had to alter the code to allow for the number of threads to be passed to the function as an argument. Since this means writing the function definition in a way that is different from what the problem gives us i removed it before turning it in. This means that the code given cannot generate the plots below.
All i did however was loop over number of threads, and write the times to a file which i then read in python in order to plot the times.

VII. SPEEDUP FROM PARALLELIZATION

One large drawback to count mutual links 1 is that since a whole 2D array is allocated it very easily runs out of space. On my computer the largest datafile i can run on this function is 50 000 webpages, which is not nearly as large as the whole web-NotreDame file. As can be seen by the plots above there is also a very large difference in speed between the two. Even thought the file run in the second plot is a lot larger than the first plot it is heaps faster, so much so that the two can't

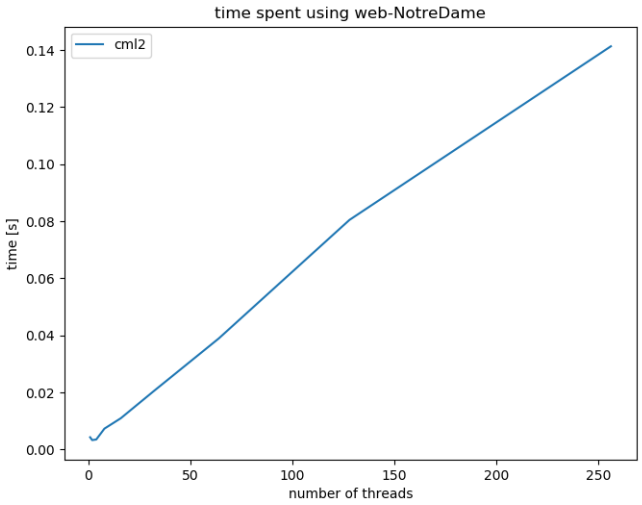

Figure 2. only count mutual links 2 with the web-NotreDame datafile.

even be plotted in the same plot while still being able to read both graphs.

In figure 2 we can see that as the number of threads increase so does the time it takes to run the code. This can be explained by the fact that allocating and distributing all the threads comes with a bit of overhead. Since the data file is no sufficiently large, and the number of cores on my computer is not high enough, the distribution of threads does prove any more efficient.

In the first figure we can see that there is a performance improvement for the larger number of threads. This might be explained by the fact that the 2D array from read graph from file 1 which is used in count mutual links 1 is indeed much larger than that in read graph from file 2. This may mean that the higher amounts of threads allocated actually leads to a performance increase.

APPENDIX