Home exam HPC spring 2019

Candidate: 15231

University of Oslo, Norway

Mar 3, 2019

Author's comment: *The program is not producing the expected results, and appears to be faulty. Writing the code took a lot more time than I had thought it would, mostly due to a combination of constraints from other mid term exams and my inadequacies in c. This has made me learn a lot of c basics - which I am very grateful for, but it has unfortunately taken focus away from optimization and unit testing. More on this under "**Program Discrepancies**".*

1 Scope of program

The program compiled with c from the files submitted with this report is built to read a file containing webpage linkage information and process that information. Provided a filename, a dampening constant d, a convergence threshold ϵ, and the number n of top ranked webpages to display, the program first reads through the linkage information, storing the linkage pattern in a hyperlink matrix using the Compressed Sparse Row (CRS) form. Furthermore, the program finds which webpages classify as dangling webpages (webpages with no outgoing links), and subsequently utilizes the PageRank algorithm with d and ϵ, to find the true PageRank of each webpage. Lastly, the program sorts the webpages according to their PageRank, and displays the top n ranked pages and the total run time of the program.

2 Program flow

Initiation: The code is centered on the main file *PE_main_15231.c*, in which calls to functions from *PE_functions_15231.c* is made. The main program first reads command line arguments, then declares array pointers and variable pointers, as well as global variables. Having declared variables, the program starts a clock in order to obtain run time, using *omp_get_wtime()*. After printing the name of the file to the terminal, the main function then makes a call to the *read_to_crs*. After CRS arrays are filled in, the number of nodes, total number of links and the number of dangling pages are printed to terminal. Then, the main function calls the *PageRank_iterations* function in order to compute the page rankings, before these are sorted using the *top_n_webpages* function. The resulting top n functions are printed to terminal along with the dampening factor d and the iteration threshold ϵ. Lastly, the final run time is computed and printed to terminal before all array pointers are de-allocated.

read_to_crs function: The array pointers used in the CRS representation of the hyperlink matrix, an array pointer to store the indices of any dangling pages variables, as well as pointers to variables for number of nodes, total number of links, and the number of dangling pages is passed to the read_to_crs function. The function allocates arrays for the number of inbound links, outbound links, and the number of selflinks for each page/node. These pointers are filled in on the go as the function iterates the columns in the data file. Using the number of inbound links, the row pointer (CRS) is filled in, before the column pointer is assigned values - initially as first encountered per hyperlink matrix row. As the column index increases whilst traversing nodes per row, the initial column pointer values are sorted using quick sort per row, such that they correspond to the hyperlink matrix. The value pointer is then filled in, using the number of outgoing links per column as normalization factor. Lastly, each column is summed up, and any webpage corresponding to a column summing to zero is marked as dangling.

PageRank_iterations function In this function, arrays to store initial pagerank guesses ($1/(\#nodes)$) and newly computed pageranks are allocated. For a data set corresponding to no dangling pages, W is pre-calculated as it is fixed for all iterations. The function then enters a while loop, which is exited at $max(\delta) < \epsilon$, where δ is absolute difference between the newly computed rank of each node and it's former value. Inside the while loop, each new rank is computed according to the PageRank algorithm, employing the CRS provided from the main function. In an attempt to speed up these calculations, shared memory parallelization is implemented using OpenMP on the node iteration, with a critical point inserted at the δ vs ϵ step, before current rank is set to the newest value.

top_n_webpages function The function first allocates an array to store the sorted values of page ranks, and the correspondingly sorted node indexes. Iterating through the nodes, a struct named object holding rank and index is then made. The struct is then sorted using qsort on the underlying object ranks. The struct is then unpacked, and rank and indexes unloaded onto the arrays, which are then handed off to main before being de-allocated.

3 Program Discrepancies

Throughout all program testing, gcc 7.3.0 for Ubuntu has been used.

The program is not producing the expected results when tested with 100-node web graph from the course web page. I noticed this test set slightly too late, and had only been testing on the the 8-webpage example provided through the exam paper while writing the code- on which the program produced the expected results.

If I had more time, I would go through the handling of dangling pages and self linking more carefully, as I expect that the fault may lie here or/and in the PageRank algorithm implementation. Regarding dangling pages, it is somewhat unclear to me whether or not nodes with no outgoing or ingoing links should be counted or not. In the exam paper, dangling pages are defined as pages corresponding to hyperlink matrix columns summing to zero. However, the hyperlink matrix is a matrix representing linkage between linked pages, and such pages as mentioned above are not after all linked.Thus further work on the program would for my part require a better understanding of the PageRank algorithm in addition to the implementation of unit testing.

The program runs through the "web-NotreDame" data set in ≈ 2.7 seconds, using $d = 0.85$ and $\epsilon = 1 \cdot 10^{-6}$, when run on an Intel i7 @3.20Ghz. Pulling on the 4 core/ 8 logical processors I get approximately the same time, which indicates that the program has a faulty PageRank algorithm implementation. For this reason examination of run time for the function is regrettably omitted. Further decreases in ϵ results in only marginally increased run times, which also indicates implementation discrepancies or counting errors.

As I have devoted most of the time close to the deadline in trying to rectify the faulty behavior, I have not been able to optimize fully. There are numerous code parts in which this should have been done, such as the pre-calculation of parts of the terms involved in finding w in the case of dangling pages in the PageRank_iterations function, and the rewriting of several of the loops used in both read_to_crs and PageRank_iterations.