Parallel and Distributed Systems
Assignment 4 – Reverse Cuthill Mckee Algorithm

- **Author:** *Papadakis Charalampos AEM:9128 mail:papadakic@auth.gr*

Source code can be found in the links below:

- https://www.dropbox.com/s/tg8zjlgzlp8dw96/RCMalgorithm.tar.xz?dl=0

Problem Description

The RCM algorithm is used to reduce the bandwidth of a sparse symmetric matrix. The "sparse" term defines that more of the half elements of the given matrix have the value zero. I define the "sparsity" of the matrix in the beginning of the source coude. All in all, the algorithm gets as input the first adjacency matrix, and after checking the neighbors of every node and reordering the matrix, returns to us a new adjacency matrix with smaller bandwidth as we will see in the results below. In this assignment I was asked to implement the algorithm using both sequential and parallel programming. For the parallel version I used the openMP library, with which I was familiar from the first assignment of the course.

Description of each implementation

- Sequential Version

Starting my sequential implementation I define the size of the sparse symmetric matrix and also its sparsity. Using them I am able to create and initialize a matrix (sparseSymmetric()) , on which I will use the algorithm and see If it is working. In this version we calculate the degree of every node , and we initialize the array that will help us reorder the adjacency matrix and also the queue that will hold the nodes before entering the array I mentioned above. The first node that enters the queue is the one with the smallest degree (fewest neighbors) , and we added without checking anything else. After that we dequeue nodes from the queue, until it is empty. However, we sort the not visited neighbors of each node we extract from the queue in ascending order and we add them to the queue. After sorting the neighbors we add the dequeued node in the array R. If the queue is empty , but the array R is not full yet , we find the node with the smallest degree and follow the same operation. When we are done adding nodes , we reverse the array R, and we are ready to reorder the adjacency matrix .

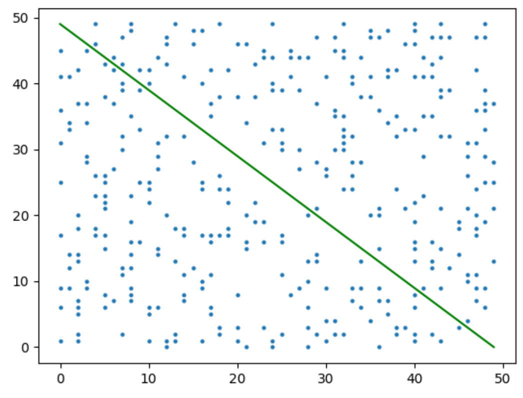
- Parallel Version

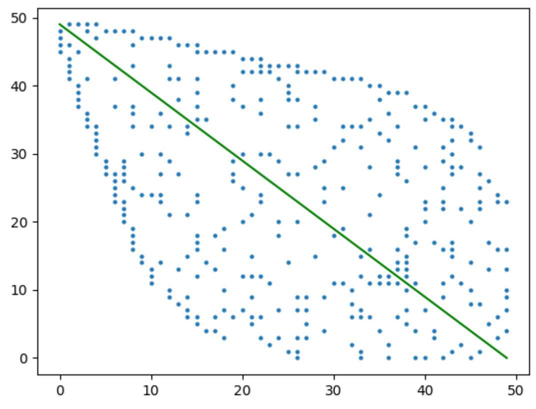
For the parallel version I used the openMP library as I mentioned above, to provide acceleration in specific tasks of the problem. Indeed, as it will be shown in the results afterwards the program ran a lot faster with the parallel implementation. In this version I also defined the THREADS that I used every time that I ran the program, and I changed it manually between 2 and 4. The tasks that are done parallel in my implementation are the calculation of each node’s degree , the finding of the node with the minimum degree and also we find the index of every connected node to the one we added to R and we change their status to visited. As you will be able to see in my source code, there are used some mutexes/locks in specific parts of the code to provide the correctness of the problem.

Testing

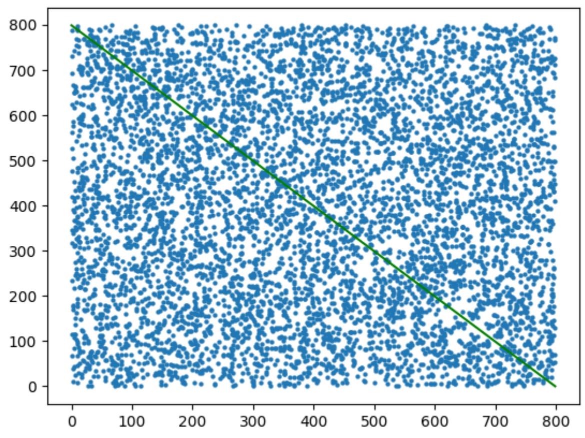
To test if the program runs properly and provide correct results I used a python script for some small arrays and I visualized the results for bigger matrices to see if the bandwidth of the array was indeed decreased. As we can see in the pictures below the program provides great results.

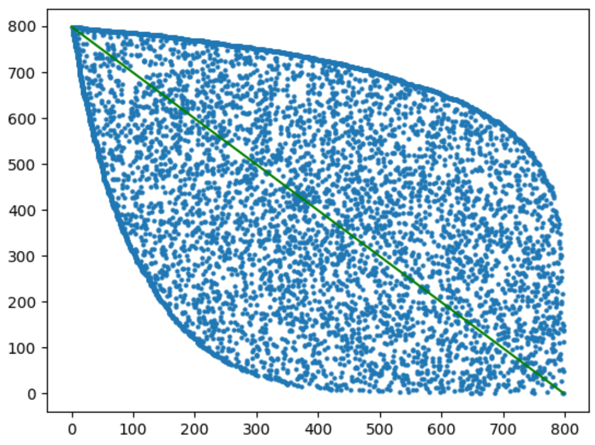
The first two pictures are from a 50x50 matrix:





The last two pictures are from a 2000x2000 matrix:





Results

Matrix Size(N)	Sequential (sec)	Parallel – 2 Threads (sec)	Parallel – 4 Threads (sec)
500	0.0068	0.0017	0.0021
2000	0.017	0.0127	0.113
10000	0.434	0.23	0.18
20000	1.937	0.7	0.606
30000	4.407	1.407	1.257
35000	9.34	1.988	1.872

As I can see in the time results represented in the table below , as the adjacency matrix gets bigger the parallel implementations show better and better time improvement. For threads above 4, the program began being slower, that’s why I decided not to show the results for 8 threads. Also, the sparsity I used running the program was 0.7. Overall, I am really pleased by the results my program provided.