Exercise 1
Parallel & Distributed Computer Systems
Παπακωνσταντίνου Χρήστος 8531
Code Link : https://github.com/krits123/RCM-OpenMP

Introduction

For this exercise we implemented a parallel version Reverse Cuthill McKee algorithm by using OpenMP. The rcm algorithm produces a premutation for the input, which is a symmetric square sparse matrix, which we refer to as “A” . The input matrix is treated as an adjacency matrix of an undirected non-weighted graph ,so each non-zero element A(i,j) represents an edge from node i to j. The algorithm then produces a relabelling of the nodes which results in a matrix with a reduced bandwidth. The relabelling of the nodes is represented by a premutation vector, ”p”, such that p(i) is the new label of the i_{th} node.

Representing a Sparse Matrix

Before we talk about the implementation, we must talk about how a Sparse matrix is stored in memory. For this version we used the Coordinate list to represent the matrix which typically consists of a list of (row, column, value) tuples. The entries are sorted first by row index and then by column index, to improve random access times. We have an entry in the least for each non-zero element and since we are talking about adjacency matrixes we can assume, without loss of generality, that the value is always equal to one and therefore store only two vectors representing the rows and columns of the non-zero elements. We also use the fact that the entries are sorted in order to increase the efficiency when searching for elements. Example : We can represent the sparse matrix

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \text{ with the vectors } row = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \\ 5 \\ 4 \\ 2 \\ 3 \end{pmatrix} \quad col = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 5 \\ 5 \end{pmatrix}$$

Sequential and Parallel Implementation

First, we will briefly describe the rcm sequential algorithm. Initially we must do is calculate the degree of each node. To do that we use the fact that the col vector is always sorted and the number of appearances of each node in it represents its degree. We can use a modified version of the binary search to find the indexes of the first and last occurrences of each label and calculate the degree for each node. We also store in another array the index of the first appearance for each node to be used later. Then we create an empty Queue, and an empty vector R to store the resulting premutation. First, we select the node with the minimum degree and append it to R. Then we add to the Queue every node adjacent to the first by increasing order of degree. Then, while the Queue is not empty, we pop an element from it, find the adjacent nodes that have not been visited and add them to the Queue ,ordered by degree. If R isn’t full, we repeat the process. When R is full, we reverse it and return.

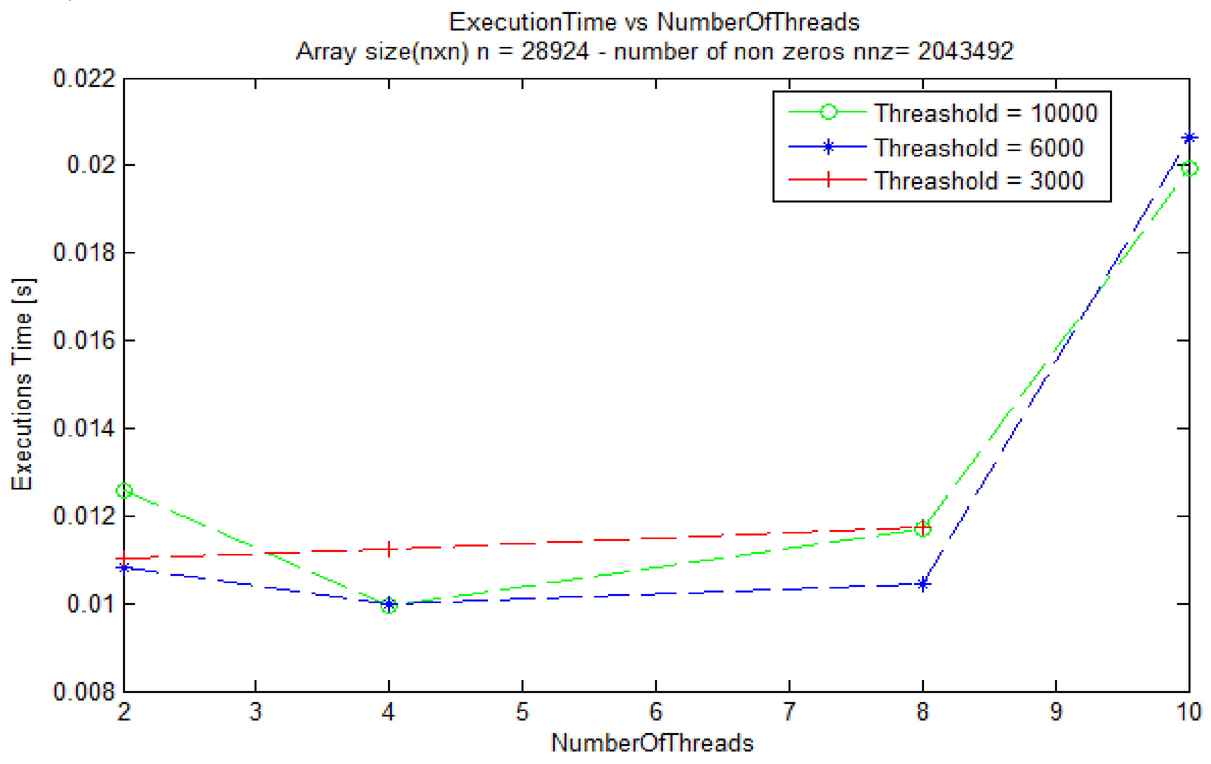
By looking at the above process we notice that the are a few things we can do in parallel :

1. Calculate the degree of each node in parallel.
2. Look for adjacent nodes in parallel and then merge the found nodes in one vector.
3. Sort the newfound nodes, by degree, in parallel.
4. Reverse R in parallel.

5. When every node has been visited many nodes are in the Queue and we only need to append them in R, without doing a search when an element is popped from the Queue. Instead of popping the elements one by one and appending we can just copy them from the Queue in parallel when we reach this point.
   We also use a threshold value and execute a routine in parallel only when its input size is bigger than that threshold. Here we note that the number of elements that need sorting in each loop is small and in rare cases doing thinks in parallel is worth doing. For that reason, we implemented the quicksort algorithm in parallel which is executed in parallel for a small depth and then sequentially.
   Finally, we note that the execution time depends on the size of the matrix (nxn) and the number of non-zero elements (nnz) in contains.

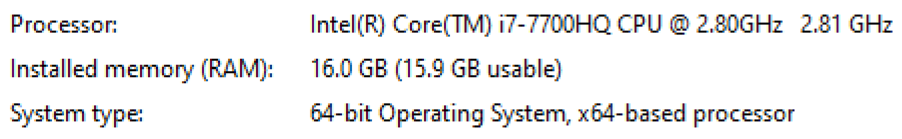
Speed Up and Results Verification

First, we used a Sparse Matrix with fixed size and varied the number of threads used and the threshold value in order to find the optimal values for our system. The matrix size is relatively small and around that size the parallelism is starting to be utilized in every parallel routine. The execution time is plotted bellow.



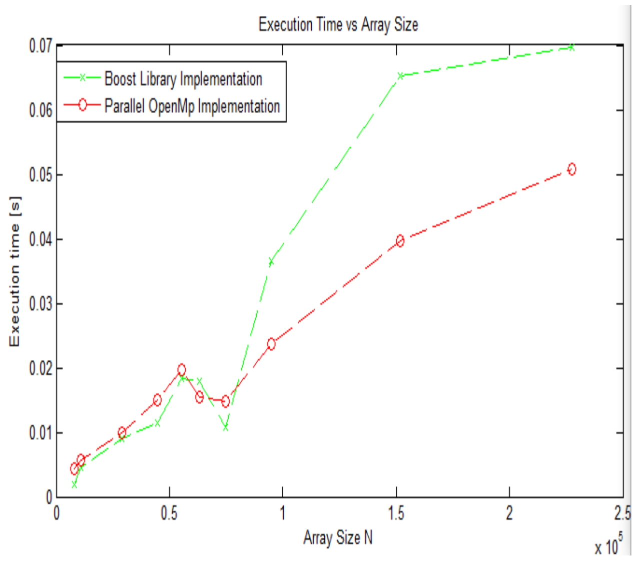
We can see that optimal thread amount is 4 and the Threshold value 6000.

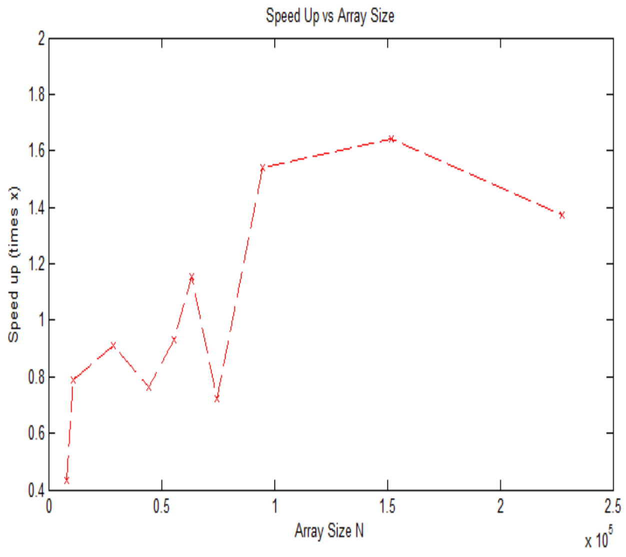
For the experiments we used Sparse matrixes of different sizes and densities from the UF Sparse Matrix Collection and compared our execution time with that of an implementation of rcm in C++ from the Boost library. More info about the implementation and the matrix collection can be found at my repository. Finally, in order to validate our results, we used MATLAB. The premutation is considered valid if it contains each node label only once and if the permuted matrix it produces has a reduced bandwidth. The experiments were run on Cygwin in a windows machine with the following specs:



Processor:	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz
Installed memory (RAM):	16.0 GB (15.9 GB usable)
System type:	64-bit Operating System, x64-based processor

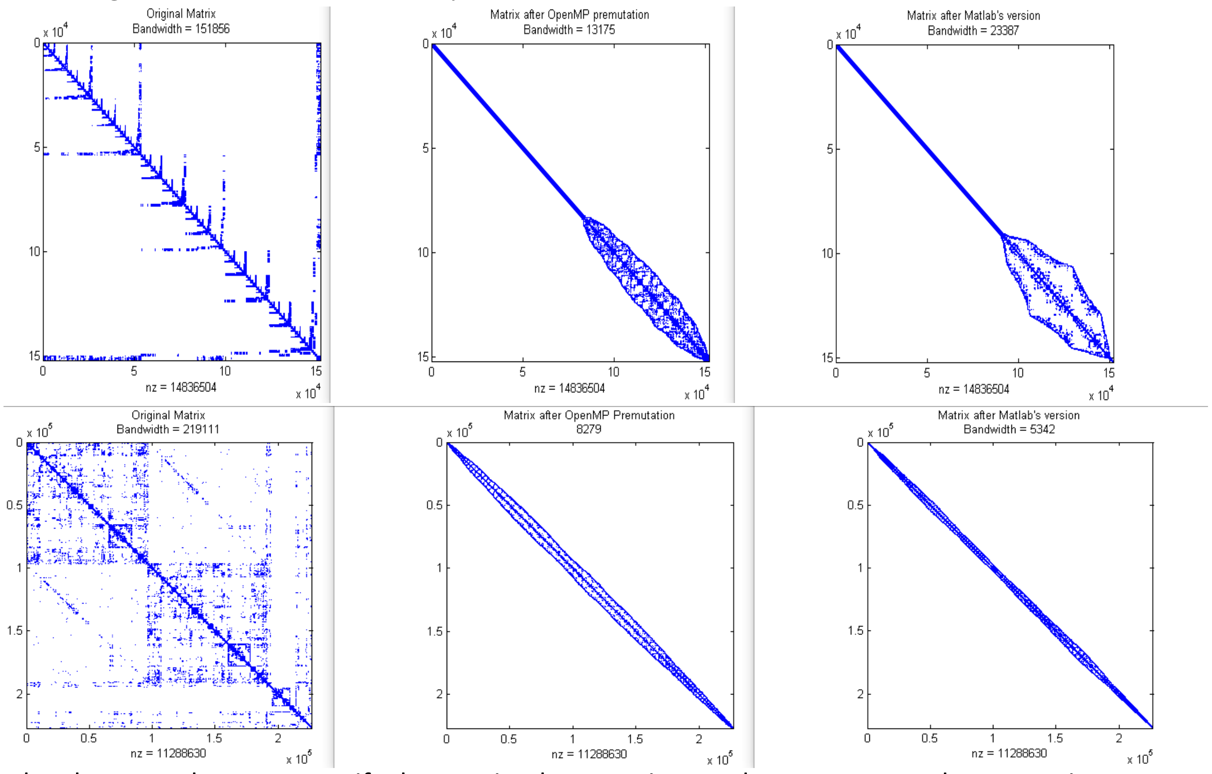
Below we can see our execution compared to that of the boost library and then the speed up we achieved.





We can see that when N is small the parallel algorithm is slower by a tiny amount, due to the overhead produced by the threads but as N gets larger the parallel algorithm becomes faster with a maximum speed up of 1,64. Also we used matrixes with size such that a parallel routine is always executed.

Finally bellow you can see some spy plots of the sparse matrixes we used and the bandwidth reduction we achieved. Note that the resulting bandwidth is highly dependant on the choice of the first node and there are other methods for choosing ,other than that of the minimum degree, and that ties in degree can be broken arbitrarily.



From the above results we can verify that our implementation produces a very good premutation. Also we see that the premutation is indeed very sensitive to the initial choice and each version works better on different matrixes.