Parallel & Distributed Systems
Exercise 4: Reverse Cuthill McKee

Gonidelis Ioannis[1] and Zikopis Evangelos[2]

[1]Department of Electrical Engineering, AUTH, AEM: 8794
[2]Department of Electrical Engineering, AUTH, AEM: 8808

September 2020

Code Repository : https://github.com/vagzikopis/ReverseCuthillMcKee

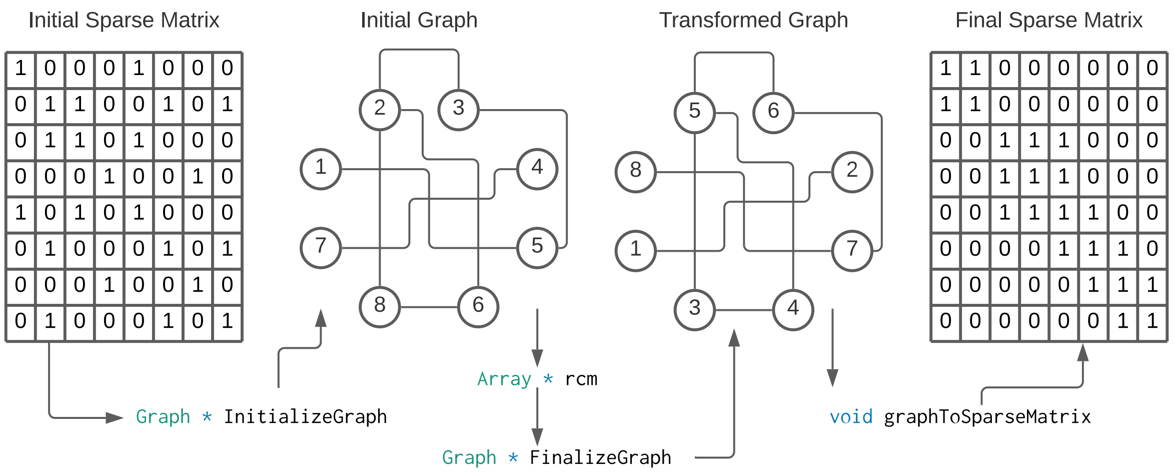
1 Introduction

The Reverse Cuthill McKee (RCM) is an algorithm to permute a sparse matrix that has a symmetric sparsity pattern into a band matrix form with a small bandwidth. RCM is an alternated form of the Cuthill–McKee algorithm (CM). Both algorithms generate an R set of vertices, that represents the new order of the final graph's vertices. Their difference is that RCM reverses the resulting indexes in R. As we can see in Figure 1, the final constructed graph is identical with the initial and their only difference is the node labelling.

2 Sequential Implementation

Our sequential implementation was designed based upon the algorithm's steps. The first step was to construct the initial graph from the input sparse matrix. This process is implemented by the function `initializeGraph()`, that generates a `Graph *` structure which represents the initial graph. After the graph construction, `rcm()` function performs the Reverse Cuthill McKee algorithm. At first, the nodes of the graph are sorted according to their degree. Also, each node's neighbours are sorted according to their degrees. After the sorting step, the node picking process begins were nodes are picked either from the graph or the Q set. For the selected node it is examined whether it should be added to the R set or not. Also, the neighbours of the selected node are added to Q set, if they are not already in. The algorithm ends when all nodes of the graph are placed in the R set. The final step consists of reversing the indexes in R set.

The next steps of our program are optional, since R set is computed and the algorithm has been completed. The final graph, that is associated with the output matrix, is constructed by the function `finalizeGraph()` and based upon the R set. Finally, the output matrix is constructed by the `graphToSparseMatrix()` function. Please note that the last two functions do not belong to the main computation algorithm, rather that they facilitate the final representation of the matrix. They don't have anything to do with the main algorithmic steps and that's why we won't demonstrate any optimizations on them in this report.



1	0	0	0	1	0	0	0
0	1	1	0	0	1	0	1
0	1	1	0	1	0	0	0
0	0	0	1	0	0	1	0
1	0	1	0	1	0	0	0
0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	0
0	1	0	0	0	1	0	1

1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	1	1	1	1	0	0
0	0	0	0	1	1	1	0
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1

Figure 1: Workflow Diagram

3 Parallel Implementation

Our parallel implementation utilizes the parallelization directives offered by `openMP`. There are certain points that need to be addressed first, though. We will try to break the theoretical algorithm into "computationally heavy" core parts and disambiguate which ones could boost performance when parallelized.

3.1 Graph Initialization

The reader should notice that this is where the main computation takes place. The two functions that accomplish the Graph Initialization are `initializeGraph()` and `parallelInitGraph()` for the sequential and the parallel part correspondingly.The reason why this is a heavy process in our program, is because **we provide the graph along with the degree information for every node**. That is to say, along with the initialization, there is being implemented the core job of counting the number of neighbours for every node. This double loop is

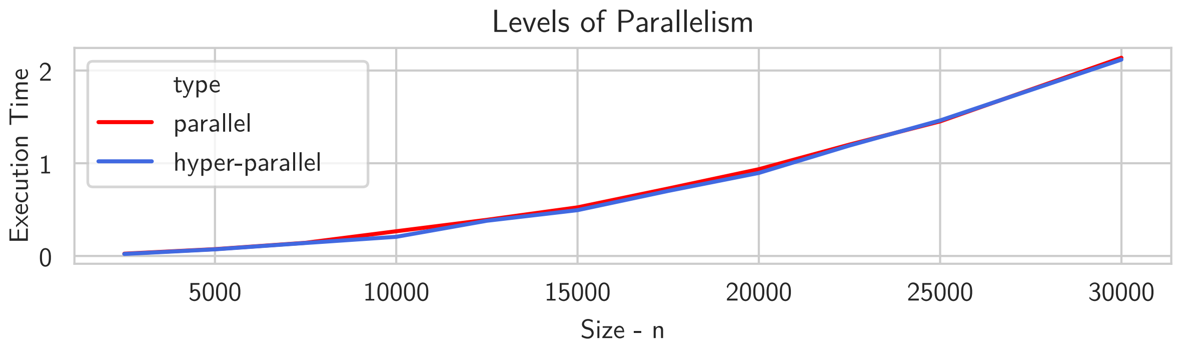


Figure 2: Levels of Parallelism Benchmark

distributed in a “one thread per node” form. Now this is the basic double loop that is parallelizable in our program and that is why it boosts performance the most, when handled properly.

3.2 Quick-Sorting and Picking Nodes

The main algorithm involves picking nodes from a sorted list. So we have the ‘sorting’ part and the ‘picking’ part.

Sorting is implemented using `quicksort` simply because “it’s quick”. There are two important arguments that need to be addressed here. First of all, `qsort()` adds a serious amount of overhead when parallelized. We have tried parallelizing it but the performance boost takes place for a minimum of a very high number of nodes. Why not use some more parallelizable sorting algorithm, then? We have tried switching to `mergesort`, as it is obvious that is by far more easy to be parallelized. Indeed the speed-up was greater, when `mergesort` was used. But that wasn’t our final choice, simply because our ultimate goal was the total performance overall. Creating a fast sequential code in the first place beats creating a slow sequential algorithm just to demonstrate some acceleration skills. To conclude, our code uses `quicksort` just because it is faster in every case while our final goal was to make a fast algorithm even faster, by utilizing `openMP`.

On the ‘picking’ part, this is where the nodes are being picked one by one from the queue - Q while the very same queue is being filled with each node’s neighbours at every iteration. This is just another double loop but there is a catch. The outer loop works in a conditional `while()` form and thus waits for a flag variable to reach a proper value, not knowing the number of iterations. This stems from the nature of the algorithm and that is why it can’t be parallelized in a performance boost way.

Another important part of this double loop is that we write in the main array R and the intermediate queue Q. Trying to do this job asynchronously imposes serious restrictions on critical parts of the code. This is because every node inserted does not have a predefined spot (in a form of chunk), rather than its place is decided dynamically according to the previous node (this is true for both nodes and neighbours positioning).*Race conditions* that add significant overhead, appear upon parallelization of the process described above.

3.3 Overall

From all the above, we could safely conclude, that according to our experiments, parallelizing a single `for-loop` of size `n` does not offer significant improvements and that is why we avoided it. This is depicted in Figure 2. There are only two double (and thus time consuming) loops in the (main) program, from which only one is offered for optimization. As one could observe from the test results though, the outcome is pretty encouraging. By applying the optimizations mentioned above, the execution time decreases more than twice, besides the fact that *the algorithm is pretty strict in sharing variables and race conditions or mutual exclusion cases.*

4 Experimental Results

4.1 Validation Test

In order to check the validity of our implementation, we tested the output of our sequential and parallel programs with the output from MATLAB. The input was a 5000×5000 sparse matrix. The results are depicted in Figure 3. As we can see, the bandwidth has been significantly reduced in both sequential and parallel outputs. Also, bandwidth values are very close to the MATLAB bandwidth value.

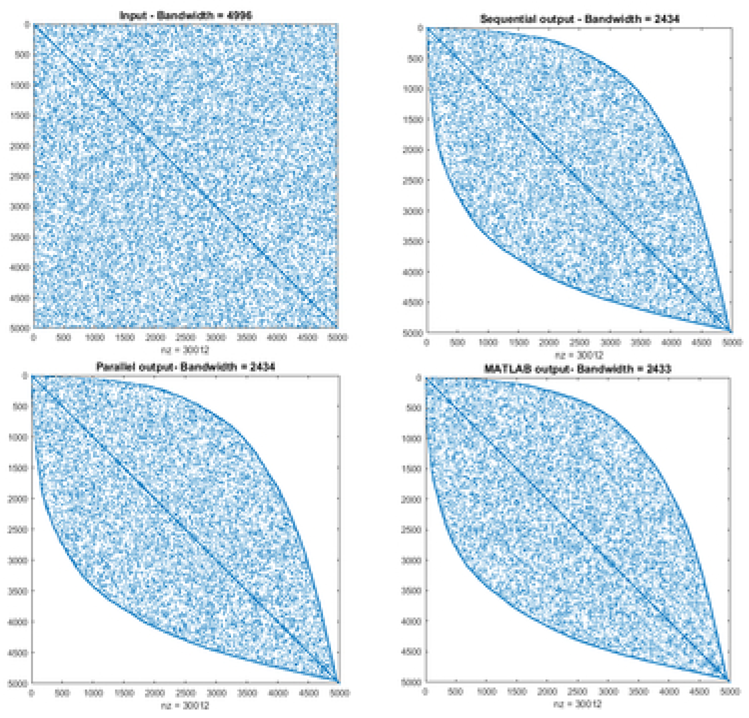


Figure 3: Validation Benchmark

4.2 Execution Time Test

Execution time benchmarking was performed on a DELL laptop with an Intel® Core™ i7-6500U CPU @ 2.50GHz × 4 processor and 8Gbytes RAM. Also, to test the scalability of our implementation we tested our parallel version on a more powerfull CPU AMD Ryzen7 4700 @ 2.0GHz x 8. Results are depicted in Figure 4. The inputs were sparse matrices with variable size and 1% density. We observed that as density was increasing the speedup among the sequential and parallel implementation was also increasing. This is a reasonable observation, as a more dense matrix means that a node has more neighbours and as a result there are more iterations in the for loops that can be parallelized.

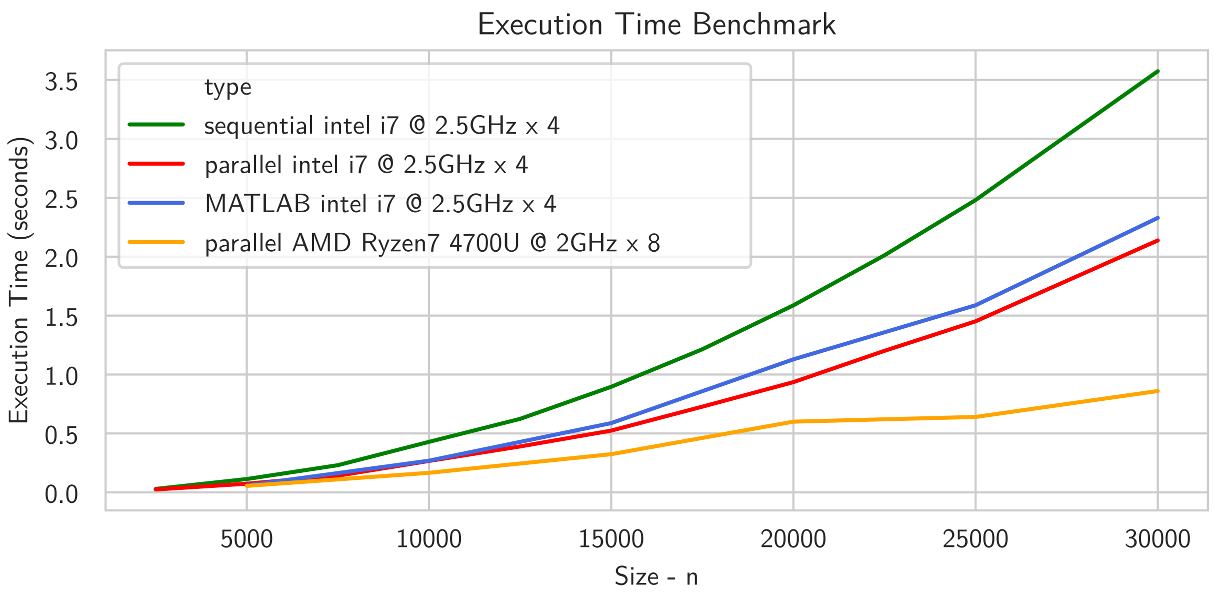


Figure 4: Execution Time Benchmark