Dimitra Karatza 8828
Aristotle University of Thessaloniki
Parallel and distributed systems
2019-2020
Github code

Reverse Cuthill McKee

30th September 2020

OVERVIEW

Reverse Cuthill McKee is an algorithm for generating a permutation for sparse matrices. By an appropriate renumbering of the nodes, it is often possible to produce a matrix with a much smaller bandwidth. Due to its multiple and time-consuming calculations for matrices with large dimensions, parallelization of the RCM enhances the code's performance. OpenMP can achieve a significant reduction in execution time as shown later on.

GOALS

1. Implement a sequential version of the RCM algorithm in C.
2. Parallelize the sequential version using OpenMP.
3. Notice the bandwidth reduction of the matrix when RCM is applied.
4. Achieve speedup of execution time with parallelization of the RCM algorithm.

CODE SPECIFICATIONS

Code consists of the following files:

- *main.c*: Creates a random sparse matrix, calls the *rcm()* function to find the proper permutations and calculates the matrix's bandwidth before and after the rcm algorithm. Moreover, it is responsible for measuring the execution time of the rcm algorithm and printing a summary of the results that come up from the code's execution.
- *rcm.h:* Header file of the code
- *queue.c:* Implements the data structure queue and operations like enqueue and dequeue.
- *quicksort.c:* Implements the quicksort algorithm.
- *Bandwidth.c:* Calculates the bandwidth of a given matrix.
- *sequential.c: Implements the sequential version of the RCM algorithm*
- *openmp.c: Implements the parallel version of the RCM algorithm.*

It is worth examining in detail the code of the sequential and parallel implementations:

Sequential

First of all, the final array *R* will keep the permutations of the node, while a queue named *Q* will keep each node's neighbors, sorted by their degree, pending to be inserted in R or not. While having the random sparse matrix, it would be useful to compute each node's degree and also keep track of the nodes that are inserted in the resulting array *R*. This is the use of the arrays *degrees[]* and *is_inserted[].*

So, while *R* is not full, the function *find_min_not_inserted()* searches for the element of the graph with the minimum degree that has not been inserted yet into the *R* and names it *parent.* Parent is added in *R* and then function *fill_Q()* is responsible for adding its neighbors to the queue *Q* in the increasing order of their degree.

While *Q* is not full, the first element from *Q* is extracted; named *child.* If *child* hasn't previously been inserted in *R*, it is added in the first free position. Then *fill_Q()* is used again to add to *Q* all the neighbours of *child* that are not in *R* in the increasing order of their degree.

After the whole process is finished and all elements have a position in R, these elements are swapped to give the final reversed array of the nodes' permutations.

Parallel

After multiple testing, it was noticed that there are two main points in the sequential algorithm that are worth parallelizing, due to the execution delay that they cause. These are:

- The calculation of the array *degrees[]* and the initialization of array *is_inserted[]*. Array *degrees[]* contains the degree of each node of the graph. The degree of a node is defined as the number of nodes adjacent to it. Array *is_inserted[]* contains 1 if an element is inserted in the final array *R,* or 0 if it's not.
- The calculation of the array *neighbors[],* which contains all neighbors of a node, in the function *fill_Q()*.

MILESTONES

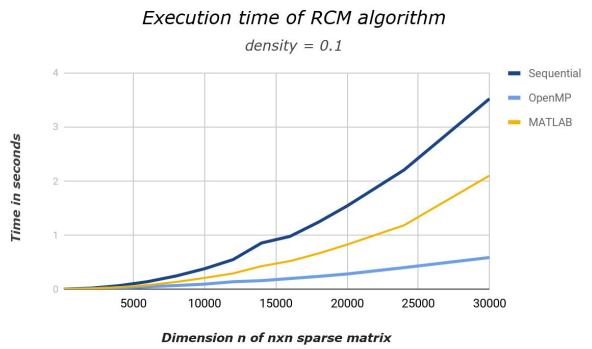
Critical sections for large input arrays

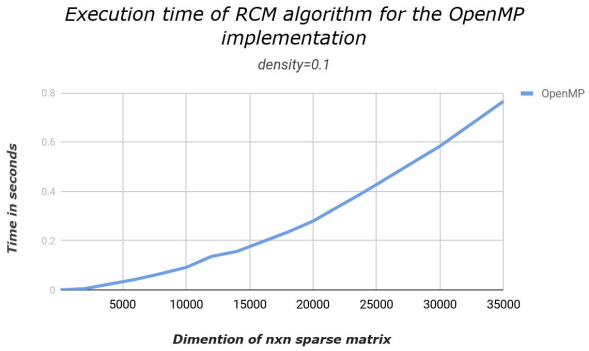
While parallelizing the finding of each node's neighbors for large input arrays, it was noticed that critical section was a problem for the performance of OpenMP, due to shared variables in between threads. This problem was solved with the use of private variables for the threads, which were concatenated to a global one in a critical section after the calculations. As a result, critical section now includes only the concatenation and does not lead to bad performance anymore.

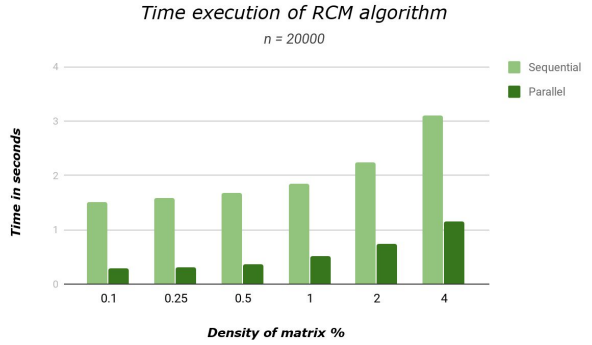
Parallelization threshold

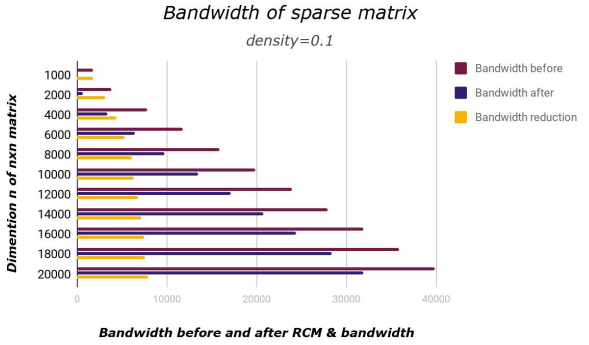
It was noticed that parallel code does not enhance the performance for very small arrays, i.e. for nxn arrays where n is smaller than 2000. As a result, if n<2000 for the input array, the RCM algorithm is implemented sequential.

CODE PERFORMANCE & BANDWIDTH REDUCTION

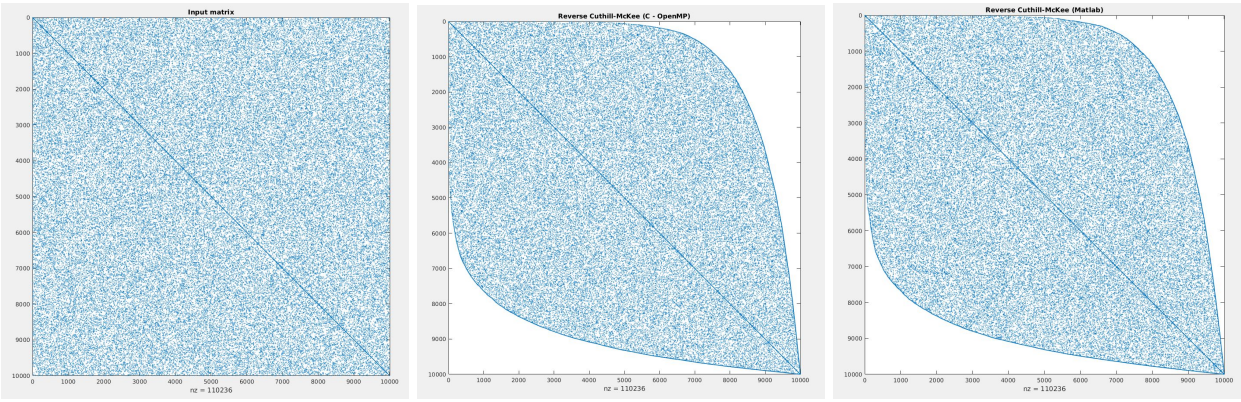








As shown above, the parallel implementation with OpenMP achieves a great speedup. Parallel is 80% faster than the sequential implementation and 65% faster than MATLAB. Advantages of parallel computing arise from these results, since we can see that parallel programming saves time, allowing the execution of code in a shorter wall-clock time. As a consequence of executing code efficiently, parallel programming scales with the problem size, and thus can solve larger problems, like the sparse graph matrix reordering.



Apart from time acceleration with parallel code, it is worth noticing that RCM algorithm reduces bandwidth. Above diagrams show bandwidth reduction for matrices with n=10000. Given the input matrix at the left, OpenMP implementation of RCM and MATLAB create output arrays with much smaller bandwidth. As long as OpenMP and MATLAB give the same output array, we are sure about the correctness of RCM implementation (both sequential and parallel).