Parallel Distributed Systems

Athanasios Manolis (8856)

September 30, 2020

https://github.com/thanosmanolis/rcm_openmp

Abstract

Report regarding an assignment for the course **Parallel Distributed Systems**. The goal is to create an implementation of Reverse Cuthill-McKee (RCM) algorithm for sparse matrix reordering in C language, and then create a parallel one using OpenMP.

1 Introduction

The RCM ordering is frequently used when a matrix is to be generated whose rows and columns are numbered according to the numbering of the nodes. By an appropriate renumbering of the nodes, it is often possible to produce a matrix with a much smaller bandwidth.

The bandwidth of a matrix is computed as the maximum bandwidth of each row of the matrix. The bandwidth of a row of the matrix is essentially the number of matrix entries between the first and last nonzero entries in the row, with the proviso that the diagonal entry is always treated as though it were nonzero.

2 Code Description

2.1 Sequential implementation

Create a queue array **Q** and a result array **R**. Those two are implemented with the function of a queue (FIFO), so that an element can be enqueued or dequeued at any time. Also create a **degrees** array to store the degree of each node and a **inserted** array to show for each node if it is inserted or not to **R** or **Q**. In order for this to work, each time an element is enqueued, change its corresponding value in **inserted** array to 1. In addition, store the degree of each node (sum of non-diagonal elements at each corresponding row).

*Until all nodes are explored (**R** is full), do the following*:

- Find the node with the lowest degree (**min_degree_idx**), and insert it to . Then insert all of its neighbors (not already inserted to **R** or **Q**) to **Q**, sorted in increasing order of degree through the function **add_neighbors_to_queue()**, which uses QuickSort, (a Divide and Conquer algorithm, implemented in another function, slightly modified, in order to sort the elements according to their degrees which are stored in another array).
- *Until **Q** is empty, do the following*: Extract the first node from **Q**. Add it to **R** and then again, insert all of its neighbors (not already inserted to **R** or **Q**) to **Q**, sorted in increasing order of degree.

When **R** is full, reverse **R** (final step of RCM algorithm), free the allocated memory, and return to the main function.

2.2 Parallel implementation

After trial & error it was decided that there are three parts worth computing in parallel:

- Calculation of the degree of each node
- Searching of the neighbors of the node most recently inserted to **R** (through the function **add_neighbors_to_queue()**)
- Sorting of the neighbors (through the function **quicksort()**)

For the *first* parallel part, the parallelization is achieved through a classic *for loop* OpenMP implementation. For the *third* parallel part, the *sections* construct is used. In essence, the first two partitions created by quickSort, are computed in parallel.

For the *second* parallel part, some extra steps had to be done. Along with the same inputs as before, function **add_neighbors_to_queue()** had one extra input, which is the index of the last neighbor of the examined element, calculated previously in the program. This was needed to be done because the loop about to become parallel, was a *while loop* with a *break;* command inside, which is not supported by OpenMP. Therefore those extra features allow us to convert it to a *for loop* and easily parallelize it with OpenMP. Apart from those extras, a **#pragma omp critical** command was added for each iteration, since the same variable is accessed by many threads at the same time.

3 Execution times

All executions were done using Intel i7-7500U CPU, 8GB RAM, 4GB Swap Memory, with a 0.1% matrix density.

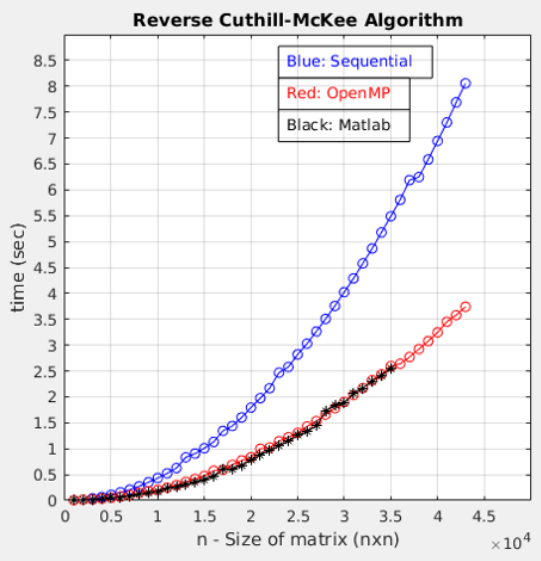

Figure 1: Comparison of different implementations

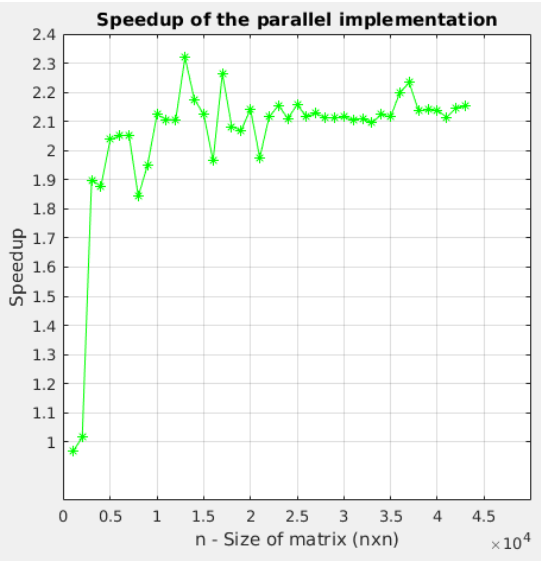

Figure 2: Speedup achieved using OpenMP

4 Observations

4.1 Result

From the images below, one can observe that this RCM implementation is an accurate one, since it's almost the same result as matlab, which for is valid. This specific test was made for n=10000 and density=0.1%.

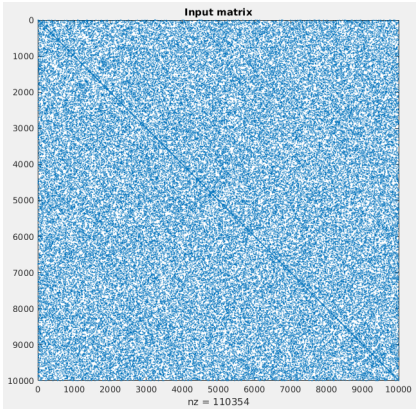

Figure 3: Input matrix (n=10000 , density = 0.1%)

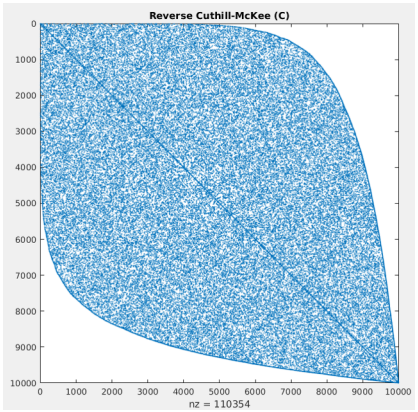

Figure 4: Output achieved with C

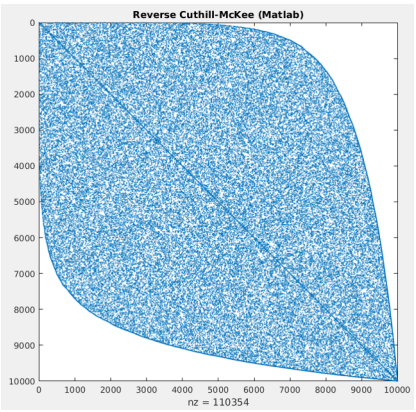

Figure 5: Output achieved with Matlab

4.2 Comments

After observations over the speedup for various parallelizations, it was concluded that the only parts worth parallelizing are three, which are actually responsible for almost 80% of the total execution time. Despite them only being three, one can clearly see that OpenMP gave a great speedup indeed (up to 2.3 times faster) which is the same execution time as Matlab's official implementation. For the first two cases, parallel computing is used only for **n** values higher than the thresholds (1000 and 2000 accordingly), because for small matrices, sequential implementation is faster than the parallel one. As for the third case, it will be computed in parallel for a number of neighbors higher than 100.

References

[1] *Tutorial: Bandwidth reduction - The CutHill-McKee Algorithm*, available at http://ciprian-zavoianu.blogspot.com/2009/01/project-bandwidth-reduction.html

[2] *Reverse Cuthill Mckee Algorithm*, available at https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm

[3] *Reverse Cuthill McKee Ordering*, available at https://people.sc.fsu.edu/~jburkardt/cpp_src/rcm/rcm.html