IN4200 : HIGH-PERFORMANCE COMPUTING AND NUMERICAL PROJECTS

CANDIDATE NUMBER: 15807
HOME EXAM 1

1. Introduction

In this report we will be studying connectivity graphs. A connectivity graph can be used to show how the data objects are directly connected with each other. In such a graph, each data object is represented by a node, and an edge connecting a pair of nodes means that the two nodes are *nearest neighbours*, i.e, directly connected. We consider all the edges as unweighted, and undirected. This means that if node v is a nearest neighbor of node u, then node u is a nearest neighbor of node v.

2. Storage formats for connectivity graph

In order to represent the connectivity graph, we will have to store the information in some data structure. Here we will consider two ways to store the data represented by the graph, firstly as a 2D table, and secondly using compressed row storage(CRS) format.

2.1. Representation of graph as a 2D table.

We can represent the connectivity graph as a 2D table, i.e, a matrix. The dimension of the 2D table is $N \times N$, where N is the number of nodes in the graph. The values in the table are either 0, or 1. Specifically if row i, and column j has value 1, then node i and j form a nearest neighbor pair, i.e they are directly connected. If they are not connected, they will have a value of 0. Since this is an undirected graph, the 2D table will be symmetric, this means $table2D[i][j] = table2D[j][i]$. By convention we have that $table2D[i][i] = 0$, so all the diagonal elements will be zero.

2.2. Representation of graph in compressed row storage(CRS) format.

We can represent the connectivity graph as compressed rows. The compressed row storage format uses two 1D arrays of integer values: row_ptr, and col_idx. The array row_ptr, is of length $N + 1$, where N is the number of nodes in the graph. The array col_idx is of length $2N_{edges}$, it stores the indices of the nearest neighbors for all the nodes. The row_ptr is used for "dissecting" the col_idx array. In this way, we only store the non-zero values of the graph.

Date: 25th March 2021.

3. Implementation

3.1. **Reading a connectivity graph from file.**

3.1.1. *2D table format.*

In this task, the function will read the input text file and store the connectivity graph in 2D table format. This method is straight forward to implement, firstly the function will open the input file, read the number of nodes, N, and then allocate the 2D table of size $N \times N$. Then I read the rest of the file into the 2D table, the code is shown below.

```
int FromNodeId, ToNodeId;
  while (fscanf(datafile, "%d %d\n", &FromNodeId, &ToNodeId) != EOF) {
      if(ToNodeId != FromNodeId) { //make sure only the legal values are stored from file
        if( (ToNodeId < *N) && (FromNodeId < *N) && (ToNodeId > -1) && (FromNodeId > -1)) {
          (*table2D)[FromNodeId][ToNodeId] =
          (*table2D)[ToNodeId][FromNodeId] = (char)1;
        }
      }
  }
```

3.1.2. *CRS format.*

It is a bit tricky to store the data in CRS format. After reading the number of nodes N, and edges, and allocating *row_ptr*, and *col_idx* arrays, I create two 1D arrrays *to_nodes* and $from\_nodes$, of length $2N_{edges}$. Then I read the rest of the file, I store the *FromNodeId*, in the array $from\_nodes$, and the *ToNodeId*, in the array *to_nodes*. Each time I also increment the *row_ptr* at the index of the node ID, plus one. Since the graph is undirected, each node ID is a FromNodeId, and a ToNodeId. Thus I do this again, this time by switching the FromNodeId to ToNodeId, and ToNodeId to FromNodeId. In the end the *row_ptr* array would contain the total number of elements in each row. I now have all the information stored to create the *col_idx* array, this is done as shown below

```
  //going through the to and from arrays to make the col_idx
  int col_counter = 0;
  for (int i = 0; i < (*N); i++) {
      for (int j = 0; j < (edges*2); j++){
          if (to_nodes[j] == i) {
              (*col_idx)[col_counter] = from_nodes[j];
              col_counter++;
          }
      }
  }
```

Next I create the *row_ptr* array, by using $row\_ptr[k] = row\_ptr[k] + row\_ptr[k-1]$, for $k = 1, .., N$. This is shown in the code below

```
 //Making the values of the row_ptr correct
  for (int k = 1; k < (*N+1); k++) {
      (*row_ptr)[k] += (*row_ptr)[k - 1];
  }
```

Finally I sort the *col_idx* row by row, using the Merge-sort sorting algorithm, I chose this algorithm because it runs in $O(n \log n)$, it is efficient for larger arrays, as the rows of *col_idx* can be quite big.

3.2. **Creating an SNN graph.**

One measure of "similarity" between two directly connected nodes, u and v, is the number of their shared nearest neighbors (SNNs). That is, how many other nodes are directly connected to both, u and v ? We can represent this by a so-called SNN graph. The SNN graph can stored as either a 2D table, or in the CRS format.

3.2.1. *2D table format.*

In this task, the function will take the connectivity graph in 2D table format, and create the corresponding SNN graph. The SNN graph is the same size as the 2D table, except the elements are now non negative integers, which represent the SNN's between node i, and node j.

In order to find the SNNs, I iterate through the 2D table. Here I take advantage of the fact that the 2D table is symmetric, and all the diagonal elements are zero. Thus I only need to iterate through the upper triangular part, minus the diagonal elements. I iterate through the matrix, and only count the SNNs if $table2D[i][j] == 1$, as a value of 0 indicates there are no SNNs for node i and j. If $table2D[i][j] == 1$, I iterate through the neighbors of node i and node j, and count the number of neighbors i and j have in common. This is the number of SNNs for $SNN\_table[i][j] = SNN\_table[j][i]$. The code is shown below

```
    for (int i=0; i<N-1 ; i++){
      for (int j=i+1; j<N ; j++){
        if (table2D[i][j]==1){
          for (int k=0; k<N ; k++){
            if (i!=k && j!=k && table2D[i][k]==1 && table2D[j][k]==1){
              (*SNN_table)[i][j] ++;
              (*SNN_table)[j][i] ++;
            }
          }
        }
      }
    }
```

3.2.2. *CRS format.*

Similar to the previous task, this task will create the SNN graph in CRS format, given the input connectivity graph in CRS format. The CRS format for an SNN graph uses a 1D array SNN_val, which is of the same length as *col_idx*, which stores the SNNs. The way I solved this can be best explained by the code below,

```
for (int i = 0; i < N ; i++){
    for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
      int node = col_idx[j];
      for (int k = row_ptr[i]; k < row_ptr[i+1] ; k++){
        for(int m = row_ptr[node]; m < row_ptr[node+1] ; m++){
          if (col_idx[k] == col_idx[m]){
```

```
            (*SNN_val)[j]++;
          }
        }
      }
    }
  }
```

Firstly, the outermost loop i iterates from 0 to $N-1$, it iterates through all the nodeID's. Then I iterate through row i, and the variable *node* will store the *node_id* of the node at index j in *col_idx*. Then I iterate through row i, and row *node*, that is, I iterate through the neighbors of node i, and node *node*.

This is because the neighbors of node n are stored in row n, in CRS format. To iterate through row n in CRS format, the for loop goes like,

```
 for (int j = row_ptr[n] ; j < row_ptr[n+1] ; n++){
    nodeID = col_idx[j]
}
```

that is, the iterator j will go through the indices of row n, and value corresponding to that index will be obtained by checking the *col_idx* array and index j.

I iterate through the neighbors of node i, and node *node*, and count how many neighbours they both have in common. This is the number of SNNs, it will be stored in $SNN\_val[j]$

3.3. **Clustering based on SNN.**

Clustering based on SNN, with a threshold value τ, aims to produce one or several clusters. Each cluster is a subset of the nodes, where each node in the cluster is directly connected with at least another node in the same cluster, with the number of SNNs between then being greater than or equal to the threshold value, τ. This means that node i is in a cluster with node j, as long as there is some path from node i to node j, where the edge weights(these are the SNNs) are greater than or equal to τ. To solve this task, my implementation ca be best described in the code snippet below

```
int *bool_arr = (int*)calloc(N, sizeof(int*)); //marking all nodes as unvisited

printf("{");
//perform a depth first search to find the clusters
DFS(N, bool_arr, node_id, node_id, tau, row_ptr, col_idx, SNN_val);
printf("}");
```

Firstly, when I call the function *check_node*, (with all the input arguments as described in the exam text), a 1D array called *bool_arr* of size N is allocated. This is because I want to do a depth first search(DFS) on the *node_id*, we want to find clusters of. The idea is that if I want to check if *node_id*, with some threshold value τ, can be in a cluster, then I do a DFS of the SNN graph, in CRS format, starting at the node *node_id*, I continue the traversal of the graph as long as the number of SNNs are greater than or equal to τ. This way I will find all the nodes that can be reached from

node_id, having $SNNs \geq \tau$, thus all the nodes forming a cluster will be found. Here is a code snippet from the DFS function,

```
if (node != main_node) printf(" %d, ", node);
visited[node] = 1; //mark node as visited

for (int i=row_ptr[node] ; i < row_ptr[node+1] ; i++){
  if(visited[col_idx[i]]==0 && SNN_val[i] >= tau){
    DFS(N, visited, col_idx[i], main_node, tau, row_ptr, col_idx, SNN_val);
  }
}
```

In order to not print out the value of *node_id* twice, I added the if condition, so it will only print the nodes that are different from *node_id*. In terms of efficiency , depth first search is an efficient method for traversing a graph. Note that DFS is called exactly once on each node. If the connectivity graph has N nodes, and E edges, then the DFS traversal of the graph can be done in $O(N + E)$. Thus it is an efficient method to use.

4. Parallelized implementation

We are asked to parallelize two function with OpenMP. The two functions to be parallelized are *creat_SNN_graph*1, and *create_SNN_graph*2. My method for solving this task is described below

4.1. **Parallelization of** *create_SNN_graph*1**.**

Parallelization of this function was straight forward. I used openMP's #pragma omp parallel for, to parallelize the outermost for loop. This is shown in the code below

```
#pragma omp parallel for
   for (int i=0; i<N-1 ; i++){
     for (int j=i+1; j<N ; j++){
       ....
       ....
```

This is the same code that was shown in section 3.2.1, which was the serial implementation. Since the foor loops are independent of each other, I am able to parallelize in this way.

4.2. **Parallelization of** *create_SNN_graph*2**.**

Parallelization of this function is done similarly to the previous function. The parallelized implementation is shown in the code below

```
#pragma omp parallel for
   for (int i = 0; i < N ; i++){
     for (int j = row_ptr[i]; j < row_ptr[i+1]; j++){
       ...
       ...
```

This is the same code snippet that was shown in section 3.2.2. I use the #pragma omp parallel for, to parallelize the outermost for loop, the for loops are independent of each other, so there will not be any race conditions.

Function	**Time taken** (seconds)
read_graph_from_file1	0.089087
create_SNN_graph1	1.325817
read_graph_from_file2	2.066268
create_SNN_graph2	6.164284
check_node	0.000032

TABLE 1. Time taken to run the serial implementation of all the functions. The first two functions are used for the graph in 2D table format, while the last three are for the CRS format.

	Time taken (seconds)	
Function	**Serial**	**Parallelized**
create_SNN_graph1	1.325817	0.348335
create_SNN_graph2	6.164284	0.872270

TABLE 2. Time taken for the functions with serial implementation, and then with parallelized implementation.

5. Looking at the timings of the functions

To see if my parallelized implementation is actually faster than the serial implementation, I have timed the functions to give me an idea of the efficiency of my code. All the timings shown in table 1, and 2 are measured by using the input file as $facebook.txt$, this graph consists of 4039 nodes, and 176468 edges. Table 1 just shows the timings for the serial implementation for all the functions. The only functions parallelized are create_SNN_graph1 and create_SNN_graph2, the timings with and without parallelization are shown in table 2.

6. Conclusion

When it comes to efficiency considerations, I can certainly see that the openMP parallelized version is faster than the serial implementation, this can be seen in table 2. The create_SNN_graph1 function creates the SNN graph in 2D matrix form, it takes 1.3258sec without parallelization, and reduces to 0.3483sec with parallelization. However the create_SNN_graph2 is quite slow, taking 6.1643sec with the serial implementation, but it does get faster with the parallelized version. This function creates the SNN graph in CRS format. In my implementation, I do iterate through the neighbors of each node many times, in order to find the number of SNNs. Perhaps it is possible to find a different way to count the SNNs, to make the serial implementation faster. I notice that read_graph_from_file2 is slow compared to read_graph_from_file1. This could be because in read_graph_from_file2, I read the file once, but then to make the col_idx array, the computational complexity is $O(N \cdot E)$, which can get quite large for large graphs. I then iterate N times to make the row_ptr, and N times again to sort col_idx. I

could come up with a different method, one which does not require iterating through the graph so many times.