UiO: IN4200 - Home Exam 1 - Spring 2021

Fábio Rodrigues Pereira - fabior@uio.no
Candidate Number - 15841

GitHub repository: https://github.com/fabiorodp/IN4200_High_Performance_Computing_and_Numerical_Projects

March 30th, 2021

Contents

1 Introduction

This home exam considers connectivity graphs with pairs of nodes directly connected, building nearest neighbors. Each edge of each connection is unweighted and "two-way," and this data representation technique is relevant in the field of data storage. Moreover, we will study a measure of "similarity" between two directly connected nodes, that is, the Shared Nearest Neighbors (SNNs). Finally, a typical unsupervised machine learning task, so-called "clustering based on SNN", will be analyzed.

Our job here is to develop algorithms to read, create SNN, and efficiently identify clusters. The functions to fulfill these tasks were developed using C programming, using serialized, paralleled, and OpenMP techniques, where the files are stored as follows:

- `data/facebook_combined.txt` - Example of connectivity graphs from Facebook.
- `data/graph_example.txt` - Example of connectivity graphs.
- `src/check_node.c` - C file containing the algorithm for the check_node function.
- `src/create_SNN_graph1.c` - C file containing the algorithm for the create_SNN_graph1 function.
- `src/create_SNN_graph1_omp.c` - C file containing the algorithm for the create_SNN_graph1_omp function.
- `src/create_SNN_graph2.c` - C file containing the algorithm for the create_SNN_graph2 function.
- `src/create_SNN_graph2_omp.c` - C file containing the algorithm for the create_SNN_graph2_omp function.
- `src/read_graph_from_file1.c` - C file containing the algorithm for the read_graph_from_file1 function.
- `src/read_graph_from_file2.c` - C file containing the algorithm for the read_graph_from_file2 function.
- `report/exam_tasks.pdf` - PDF file containing the exam tasks.
- `report/final_report.pdf` - PDF file containing the final report.
- `main.c` - C file containing the test functions.
- `main.h` - C header file containing the test functions.
- `README.md` - README with compile and run explanations.

2 Discussion

In the following subsections, we will go through the algorithm functions and their details.

2.1 Read Graph from file 1

In this function, we are writing an algorithm for reading a connectivity graph file and outputting it in a 2D array as well as their number of nodes. The order of the implementations is:

1. reads a text file prescribed as input containing a connectivity graph;
2. outputs the number of nodes through N;
3. allocates a 2D table of correct dimension inside the function;
4. fills its content and passes it out (thus triple pointer as type for table2D).

2.1.1 Specifies:

1. N := Number of nodes in the connectivity graph;
2. Binary table2D of size N x N of char values;
3. table2D[i][j] = table2D[j][i];
4. table2D[i][i] = 0 := a node cannot be a nearest neighbor of itself;
5. The of values=1 in table2D is twice the of edges.

2.1.2 Input and Output:

char *filename: File containing a connectivity graph;
int *N : The number of nodes.
char ***table2D := 2D table.

2.1.3 Example:

An example of table2D for the Connectivity Graphs file in data/graph_example.txt is:

For the input file:

```
# Connectivity graph example from
the exam1: graph_example.txt
# Author: Fabio Rodrigues Pereira
# Nodes: 5 Edges: 8
# FromNodeId ToNodeId
0 1
0 3
0 2
2 4
2 1
2 3
1 3
3 4
```

The output is:

```
0 1 1 1 0
1 0 1 1 0
1 1 0 1 1
1 1 1 0 1
0 0 1 1 0
```

2.1.4 Algorithm and performance-wise explanations:

```
1  void read_graph_from_file1(char *filename, int *N, char ***table2D)
2  {
3      // declaring variable pointer file
4      FILE *file;
5
6      // opening file
7      file = fopen(filename, "r");
8
9      // returning error if file not found
10     if ( file == NULL ){
```

```
11          perror("Error: File not found.");
12          exit(1);
13      }
14
15      // getting the number of nodes
16      char ln[64];
17      while ( fgets(ln, sizeof ln, file) )
18          if (sscanf(ln, "# Nodes: %d", N) == 1) break;  // efficiency
19
20      // allocating 2D pointer arrays
21      (*table2D) = calloc(*N, sizeof **table2D);  // *A = A[0], **A = A[0][0]
22      for ( int i = 0; i < *N; i++ )
23          (*table2D)[i] = calloc(*N, sizeof ***table2D);  // ***A = A[0][0][0] = int
24
25      // reading and assigning specific values to pointer arrays
26      while ( fgets(ln, sizeof ln, file) ) {
27          if (ln[0] == '#') continue;  // efficiency
28          unsigned int temp1, temp2;  // to not incur in memory storage traffic
29          sscanf(ln, "%u %u\n", &temp1, &temp2);
30          *(*(*table2D+temp1)+temp2) = *(*(*table2D+temp2)+temp1) = 1;
31      }
32
33      // closing file
34      fclose(file);
35  }
```

In lines 3-13, we declare a FILE global variable, open it in reading mode with fopen function, and test if the file was found.

In lines 16-18, we declare a vector of characters of 64 bites, long enough to read line by line from the text file, and then the fgets and sscanf functions do the job of looking for the number of nodes stated inside the text file. It is essential to mention that a if statement with break clause is used in this code block for efficiency purposes, that is, the while loop breaks as soon as the number of nodes is found, avoiding unnecessary work.

In lines 21-23, a 2D array is allocated using the function calloc, that is, all elements of the array are assigned with zeros. Each element of the array has 8 bytes (pointer) for the outer array element and 1 byte (char) for the inner array element.

In lines 25-31, we look for the graph's connections in the text file, using fgets and sscanf. For efficiency purposes, an if statement with a continue clause is applied, where the while loop ignores the lines that contain the character in the beginning. This character is observed in the lines that we are not interested in, then skipping these lines will save processor work. At the end of this code block, line 30, the algorithm assigns 1 in the 2D array (Nodes X Nodes) when the node connection is found, everything according to the rules in 2.1.1.

In line 34 we close the opened file and the function terminates.

2.2 Read Graph from file 2

In this function, we are writing an algorithm for reading a connectivity graph file and outputting it in Compressed Row Storage (CRS) format, containing col_idx and row_ptr 1D arrays and their number of nodes N. The order of the implementations is:

1. Reads a text file that contains a connectivity graph;
2. Outputs the number of nodes through N;
3. Allocate and assign values for the row_ptr, col_idx 1D arrays inside the function;
4. Passes the arrays out (thus double pointer as type).

2.2.1 Justification of using CRS:

1st. More memory efficient than version 1 for the case where the number of edges (N_edges) are smaller than N*N by orders of magnitude, because the numerical values of the table2D (sparse matrix) from version 1 are mostly zeros. It is a waste of float-point operations when the sparse matrix is used in the

way presented in version 1. Besides, it is a waste of memory storage when a sparse matrix is allocated as a 2D array.

2nd. CRS only stores the nonzero values and avoids multiplication with zeros.

2.2.2 Specifies:

```
- Uses two array1D:
    -> int **col\_idx:
        |-> (*col\_idx) = malloc(2*N\_edges * sizeof col\_idx)
        |-> consecutively stores for all nodes the indices of their
            nearest neighbors.

    -> int **row\_ptr:
        |-> (*row\_ptr) = malloc(N+1 * sizeof row\_ptr)
        |-> "dissecting" the array col\_idx with respect to the diff nodes.
```

2.2.3 Input and Output:

char *filename: File containing a connectivity graph;
int *N: The number of nodes.
int **row_ptr: 1D arrays with the row pointers;
int **col_idx: 1D array with the columns indices.

2.2.4 Example:

Examples of col_idx and row_ptr for the Connectivity Graphs file in data/graph_example.txt, check 2.1.3:

The output is:
col_idx: 1,2,3, 0,2,3, 0,1,3,4, 0,1,2,4, 2,3
row_ptr: 0,3,6,10,14,16

2.2.5 Algorithm and performance-wise explanations:

```c
void read_graph_from_file2(char *filename, int *N, int **row_ptr,
                           int **col_idx)
{
    FILE *file;

    // opening file
    file = fopen(filename, "r");

    // returning error if file not found
    if ( file == NULL ){
        perror("Error: File not found.");
        exit(1);
    }

    // getting the number of nodes:= N and edges:= N_edges
    char ln[64];
    unsigned int N_edges;
    while ( fgets(ln, sizeof ln, file) )
        if (sscanf(ln, "#%*s%d%*s%u\n", N, &N_edges)) break;  // efficiency

    // allocating arrays
    (*col_idx) = calloc( 2 * N_edges, sizeof **col_idx );
    (*row_ptr) = calloc( (*N + 1), sizeof **row_ptr );

    // declaring variables
    size_t node;
    unsigned int temp1, temp2, count = 0;

    // assigning values to the arrays
    for ( node = 0; node < (*N)+1; node++ )  // to keep arrays' order
    {
```

```
32          while ( fgets(ln, sizeof ln, file) )
33          {
34              if (ln[0] == '#') continue;  // efficiency
35
36              // not incur in memory storage traffic
37              sscanf(ln, "%u %u\n", &temp1, &temp2);
38
39              if ( temp1 == node )
40              {
41                  (*col_idx)[count] = temp2;
42                  count++;
43              }
44              else if ( temp2 == node )
45              {
46                  (*col_idx)[count] = temp1;
47                  count++;
48              }
49          }
50
51          // assigning the count
52          (*row_ptr)[node+1] = count;
53
54          // moving the cursor from the end to the beginning of the file
55          rewind(file);
56      }
57
58      // sorting col_idx array
59      for (size_t i = 0; i < *N; i++)  // looping over row_ptr
60          for (size_t j = (*row_ptr)[i]; j < (*row_ptr)[i+1]; j++)
61              for (size_t k = j + 1; k <  (*row_ptr)[i+1]; k++)
62                  if ( (*col_idx)[j] > (*col_idx)[k] )
63                  {
64                      int temp =  (*col_idx)[j];
65                      (*col_idx)[j] = (*col_idx)[k];
66                      (*col_idx)[k] = temp;
67                  }
68  }
```

In lines 4-13, we declare a FILE global variable, open it in reading mode with fopen function, and test if the file was found.

In lines 15-19, we declare a vector of characters of 64 bites, long enough to read line by line from the text file, and then the fgets and sscanf functions do the job of looking for the number of nodes and number of edges stated inside the text file. It is essential to mention that a if statement with break clause is used in this code block for efficiency purposes, that is, the while loop breaks as soon as the number of nodes and edges are found, avoiding unnecessary work.

In lines 21-23, 2 different 1D arrays, col_idx and row_ptr, are allocated using the function calloc, that is, all elements of the arrays are assigned with zeros. Each element of the array has 4 bytes (integer). The size of col_idx is 2 times the number of nodes times 4 bytes (integer). The size of row_ptr is the number of nodes plus 1, everything times 4 bytes (integer).

In lines 25-56, we are interested in specifying values to col_idx and row_ptr 1D arrays from a text file. For that, the algorithm reads line by line of the text file, recognizes the pair of connected nodes, assign temp1 and temp2 variables temporally (to avoid memory storage traffic) with the found connected nodes, checks the order of the correct nodes, and designate the node value to col_idx array. Finally, it saves the count in row_ptr, and the rewind function moves the buffer to the beginning of the text file to be used in the next node/step. It is important to notice the if statement with continue clause (line 34) which is used for avoiding the loop to work in unnecessary situations.

In lines 58-68, we sort col_idx according to its row_ptr. This step may increase efficiency for this project's following tasks, and then it was decided to be implemented here. On the other hand, it may compromise the performance of the function read_graph_from_file2, however, in performance-speaking, it would be worth it in the future.

2.3 Create SNN Graph 1

In this function, we are interested in finding the Shared Nearest Neighbors graph (SNN_table) from a table2D 2.1.

2.3.1 Input and Output

int N: Number of Nodes;
char **table2D: Array containing the 2D table;
int ***SNN_table: 2D array containing the SNNs, allocated inside the function.

2.3.2 Example

For example, an inputs N=5 and table2D equals:

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

generates a SNN_table equals:

0 2 2 2 0
2 0 2 2 0
2 2 0 3 1
2 2 3 0 1
0 0 1 1 0

2.3.3 Serialized algorithm:

```
1  void create_SNN_graph1(int N, char **table2D, int ***SNN_table)
2  {
3      // allocating 2D array for SNN_table
4      (*SNN_table) = calloc(N, sizeof **SNN_table);
5      for (size_t i = 0; i < N; i++)
6          (*SNN_table)[i] = calloc(N, sizeof ***SNN_table);  // ***A = A[0][0][0]
7
8      for (size_t i = 0; i < N; i++)  // not dependent for loop construct
9          for (size_t j = 0; j < N; j++)  // depend on the outer loop
10         {
11             // checking if the node connection does not repeat i.e. 0-1 or 1-0
12             // continue clause used for avoiding unnecessary work for the loop
13             if (j <= i) continue;
14
15             // checking if both nodes are connected
16             // continue clause used for avoiding unnecessary work for the loop
17             if ((table2D[i][j] == 0) && (table2D[j][i] == 0)) continue;
18
19             // assigning values in SNN_table
20             for (size_t k = 0; k < N; k++)
21                 if ((table2D[i][k] == 1) && (table2D[j][k] == 1))
22                     (*SNN_table)[i][j] = (*SNN_table)[j][i] += 1;
23         }
24 }
```

In lines 3-6, we allocate a dynamic 2D calloc array with 8 bytes (pointer) for each outer array elements and 4 bytes (int) for each inner array elements.

In lines 8-23, we create a nested for loop construct to assign values in SNN_table. Highlight to the lines 8 and 9 that we could have written in another way, so-called for loop fusion, as follows:

```
1  for( size_t x = 0; x < N*N; n++ ) {
2      size_t i = x/N; size_t j = x%N; ...
```

However, performance tests showed that it did not give any benefit. Therefore we kept the former architecture.

In lines 13 and 17 were implemented if statements with continue clauses for avoiding unnecessary work for the for loop construct.

In lines 20-22, the nested for loop is indexing the table2D array to find equal nodes in its row and columns, then increments 1 in its specific SNN_table element.

2.3.4 Parallel algorithm with OpenMP:

```
1  void create_SNN_graph1_omp(int N, char **table2D, int ***SNN_table)
2  {
3      // global variables that are shared in the parallel region by default
4      size_t z, i, j, k;
5
6      // allocating 2D array for SNNs
7      (*SNN_table) = calloc(N, sizeof **SNN_table);
8
9  #pragma omp parallel
10     {
11 #pragma omp for private(z)
12         for (z = 0; z < N; z++)  // not dependent on anything
13             (*SNN_table)[z] = calloc(N, sizeof ***SNN_table);
14
15 #pragma omp for private(i, j, k) schedule(dynamic)
16         for (i = 0; i < N; i++)  // not dependent on anything
17             for (j = 0; j < N; j++)  // depends on i (the outer loop)
18             {
19                 // checking if the node connection does not repeat i.e. 0-1 or 1-0
20                 if (j <= i) continue;
21
22                 // checking if both nodes are connected
23                 if ((table2D[i][j] == 0) && (table2D[j][i] == 0)) continue;
24
25                 // assigning values in SNN_table
26                 for (k = 0; k < N; k++) // depends on i and j
27                     if ((table2D[i][k] == 1) && (table2D[j][k] == 1))
28                         (*SNN_table)[i][j] = (*SNN_table)[j][i] += 1;
29             }
30     }
31 }
```

As one can see from the above code, since there are two for loop constructs that are not dependent on anything (line 12 and 16), they can be parallelized. The API chosen in this exam to perform the parallelization is the OpenMP.

First, we create a parallel region at line 9.

Then, two 'omp for' directives are created before lines 12 and 16 because they are not dependent on anything and can be handled by different threads (workers).

Private clauses are implemented (lines 11, 15) to the variable (z, i, j, k) to have private instances in each thread, avoiding race conditions and preventing different results on the final results. This clause is essential because the variables (z, i, j, k) were declared before the parallel region. By default, in OpenMP, they are shared between the threads in the parallel region, but we want to make it private.

It is essential to say that Collapse(2) clause is not implemented here, even though having 2 nested for loops in lines 16 and 17, because it did not improve the performance of our algorithm.

Finally, the schedule(dynamic) clause is implemented in line 15 because it should help us with load balance problems, where each for loop can take a different amount of time to complete its work. Indeed, in our algorithm architecture, some threads will finish the work immediately after the 'if' statement with the 'continue' clause is activated (lines 20 and 23), skipping the next nested for loop (lines 26-28). On the other hand, some threads will take more time to complete the work. Work imbalance leads to spending

more than necessary CPU time as the threads that finish earlier wait for the others to finish. Dynamic scheduling overcomes that by spreading loop chunks on a first-come, first-served principle.

2.4 Create SNN Graph 2

In this function, we are interested in creating the corresponding Shared Nearest Neighbors SNN graph, using Compressed Row Storage (CRS) techniques.

2.4.1 Input and Output

int N: Number of Nodes;
int *row_ptr: 1D Array with the rows pointers;
int *col_idx: 1D Array with the columns indices;
int **SNN_val: 1D array containing the SNN's values, allocated inside the function, in a CRS format.

2.4.2 Specifies:

```
- An 2D array for SNN:
    |-> unsigned int **SNN_val:
        |-> (*SNN_val) = malloc(2*N_edges * sizeof SNN_val)
```

2.4.3 Example

For example, a col_idx equals: 1,2,3, 0,2,3, 0,1,3,4, 0,1,2,4, 2,3
and a row_ptr equals: 0,3,6,10,14,16
return a SNN_val equals: 2,2,2, 2,2,2, 2,2,3,1, 2,2,3,1, 1,1.

2.4.4 Serialized algorithm:

```
1  void create_SNN_graph2(int N, int *row_ptr, int *col_idx, int **SNN_val)
2  {
3      // allocating SNN_val that has the same length of col_idx
4      (*SNN_val) = calloc(row_ptr[N + 1], sizeof **SNN_val);
5
6      // global variables
7      size_t z, x, i, j, row_nr;
8      unsigned long count;
9
10     // loop construct to assign values for (*SNN_val)
11     for ( z = 0; z < row_ptr[N + 1]; z++ )
12     {
13         count = 0;
14
15         for ( x = 0; x < N+1; x++ )  // getting row
16         {
17             if ( z < row_ptr[x] )
18             {
19                 row_nr=x-1;
20                 break;
21             }
22         }
23
24         for ( i = row_ptr[row_nr]; i < row_ptr[row_nr + 1]; i++ ) // in element/row
25         {
26             for ( j = row_ptr[col_idx[z]]; j < row_ptr[col_idx[z] + 1]; j++ )  // in
       element/col
27                 if ( col_idx[i] == col_idx[j] )
28                     count += 1;
29         }
30
31         (*SNN_val)[z] = count;
32     }
33 }
```

In lines 3-4, we allocate a 1D calloc SNN_val array of length equals to col_idx, and size equals 2 times the number of edges times 4 bytes (int).

In lines 6-23, we design the architecture for assigning values in SNN_val array. First, global variables are declared. Then, we loop through the elements in col_idx accordingly to row_ptr, checking node connections and counting the number of nearest neighbors. Finally, we assign the count of SNNs in SNN_val array. Attention to the break clause in if statement (line 20), bringing efficiency to the algorithm when it breaks a nested for loop as soon as it gets the needed information and avoids unnecessary work.

2.4.5 Parallel algorithm with OpenMP:

```
1  void create_SNN_graph2_omp(int N, int *row_ptr, int *col_idx, int **SNN_val)
2  {
3      // allocating SNN_val that has the same length of col_idx
4      (*SNN_val) = calloc(row_ptr[N + 1], sizeof **SNN_val);
5
6      // global variables are shared by default in the parallel region
7      size_t z, x, i, j, row_nr;
8      unsigned long count;
9
10 #pragma omp parallel for private(z, x, i, j, row_nr) reduction(+:count)
11     for ( z = 0; z < row_ptr[N + 1]; z++ )
12     {
13         count = 0;
14
15         for ( x = 0; x < N+1; x++ )  // getting row
16         {
17             if ( z < row_ptr[x] )
18             {
19                 row_nr=x-1;
20                 break;
21             }
22         }
23
24         for ( i = row_ptr[row_nr]; i < row_ptr[row_nr + 1]; i++ ) // in element/row
25         {
26             for ( j = row_ptr[col_idx[z]]; j < row_ptr[col_idx[z] + 1]; j++ )  // in
       element/col
27                 if ( col_idx[i] == col_idx[j] )
28                     count += 1;
29         }
30
31         (*SNN_val)[z] = count;
32     }
33 }
```

The code block above shows only a for loop construct (line 11) not dependent on anything, then parallelizable.

First, we create a parallel region with for directive and the necessary clauses already all together.

The directive for indicates that we want to distribute the threads among the loop steps (z).

The clause private, likewise explained in 2.3.4, transforms the variables (z, x, i, j, row_nr) from shared to private for each threads in the parallel region, avoiding race conditions and preventing different results on the final results.

In the end, the reduction clause designates that the variable count that is private to each thread is the subject of a reduction operation (+) at the end of the parallel region, avoiding race conditions and preventing different results on the final results.

2.5 Check Node

This function checks whether a node can be in a cluster, where each node in the cluster is directly connected with at least another node in the same cluster, with the number of SNNs between them being equal or larger than the threshold tau.

For a given node (node_idx) and a threshold value (tau), whether it can be inside a cluster based on the input SNN graph.

If yes, the function prints out the other nodes inside the cluster.

2.5.1 Input and Output

int node_id: Given node_id by user.
int tau: Given tau by user.
int N: Number of nodes.
int *row_ptr: Row pointers.
int *col_idx: Column indices.
int *SNN_val: SNN values.

2.5.2 Example

For example, an col_idx equals 1,2,3, 0,2,3, 0,1,3,4, 0,1,2,4, 2,3 and row_ptr equals 0,3,6,10,14,16 and SNN_val equals 2,2,2, 2,2,2, 2,2,3,1, 2,2,3,1, 1,1, returns:

node_id equals 4
tau equals 1 or 2
0,1,2,3

node_id equals 2
tau equals 3
2, 3

2.5.3 Algorithm and performance-wise explanations:

```
void check_node(int node_id, int tau, int N, int *row_ptr, int *col_idx,
                int *SNN_val)
{
    // error if node_id >= N
    if ( node_id >= N )
    {
        printf("\nError: The node_id can not be higher than the number of "
               "nodes N.\n");
        exit(1);
    }

    // store nodes that are in the same cluster
    // maximum possible are N minus 1 that is represented by the given node_id
    int *cluster = malloc(row_ptr[N+1] * sizeof *cluster);

    // global variables
    size_t i, j, x;

    // assigning -1 for all elements in cluster array
    // to not have any value equals node_id in cluster array
    for ( i = 0; i < row_ptr[N+1]; i++ ) cluster[i] = -1;

    // finding the clustered nodes and storing in the cluster array
    int count = 0;
    for ( i = row_ptr[node_id]; i < row_ptr[node_id + 1]; i++ ) // in element/row
    {
        for ( j = row_ptr[col_idx[i]]; j < row_ptr[col_idx[i] + 1]; j++ )  // in element
    /col
            if ( SNN_val[j] >= tau )
            {
                int is_in = 0;

                // checking if the node is already stored
                for ( x = 0; x < row_ptr[N+1]; x++ )
                {
                    if ( (cluster[x] == col_idx[j]) || (node_id == col_idx[j]) )
                        is_in = 1;
```

```
37                  }
38
39                  if ( is_in == 0 )
40                  {
41                      cluster[count] = col_idx[j];
42                      count++;
43                  }
44              }
45      }
46
47      // reducing memory usage
48      cluster = realloc(cluster, count * sizeof *cluster);
49
50      // printing clustered nodes
51      printf(">> The Nodes that form a cluster with node_id %d and tau %d: ",
52             node_id, tau);
53      for ( size_t z = 0; z < count; z++)
54      {
55          if ( z != 0 ) printf(",");
56          printf("%d", cluster[z]);
57      }
58
59      // freeing memory
60      free(cluster);
61  }
```

In lines 4-10, we implement an if statement to verify if the user gives a proper input, otherwise it returns an error.

In lines 14-21, we allocate a 1D malloc array (cluster) and assign values equal to -1 for all elements. This assignment is necessary because, further in our algorithm architecture, we will store the node indices in the cluster without repetition (lines 30-43). Since the elements inside the cluster array can not match any node index initially, hence -1 was chosen to be assigned.

The lines 34-45 contain our algorithm architecture for finding the node indices in the cluster set, without repetition, and according to exercise requirements.

In line 48, we reallocate the cluster array, because now it is smaller than the previous one, and this reallocation saves memory usage of the program.

In lines 50-57, the algorithm prints the node indices in the cluster set.

In line 60, the memory usage is freed.

2.6 Conclusion - Main Test

This section will test our implementations on data/facebook_combined.txt file, a Connectivity Graph representing social circles from Facebook, containing 4039 nodes and 88234 edges.

First, the files main.c and main.h have to be compiled. Linux users, with OpenMP 4.5+ installed, compile the algorithm with 'gcc -fopenmp main.c main.h'. For Mac Os users, the way to compile is using 'clang -Xpreprocessor -fopenmp main.c main.h -lomp'. For both systems, './a.out [connectivity_graphy_data_path] [number_of_threads]' is the way to run the tests.

The output below is an example of the results we will discuss afterwards:

```
% clang -Xpreprocessor -fopenmp main.c main.h -lomp
% ./a.out data/facebook_combined.txt 4

Testing read_graph_from_file1.c and create_SNN_graph1.c.....
Reading Connectivity Graph 1...
>> CPU time: 0.042819 s.
Please, type 'y' if you want to print table2D here:n
Creating SNN graph 1 without OMP...
>> CPU time: 1.283154 s && ELAPSED Time: 1.302531 s.
Creating SNN graph 1 with OMP...
>> CPU time: 2.273064 s && ELAPSED Time: 0.683618 s.
```

```
Speedup between SNN_table and SNN_table_omp:
>> SNN_table/SNN_table_omp = 1.905349
Comparing if SNN_table and SNN_table_omp are equals:
>> True
Please, type 'y' if you want to print SNN_table here:n
Please, type 'y' if you want to print SNN_table_omp here:n

Testing read_graph_from_file2.c and create_SNN_graph2.c.....
Reading Connectivity Graph 2...
>> CPU time: 101.214561 s.
Please, type 'y' if you want to print col_idx and row_ptr here:n

Creating SNN graph 2 without OMP...
>> CPU time: 8.024934 s && ELAPSED Time: 8.044511 s.
Creating SNN graph 2 with OMP...
>> CPU time: 16.249126 s && ELAPSED Time: 5.504587 s.
Speedup between SNN_val and SNN_val_omp:
>> SNN_val/SNN_val_omp = 1.461420
Comparing if SNN_val and SNN_val_omp are equals:
>> True
Please, type 'y' if you want to print SNN_val here:n

Please, type 'y' if you want to print SNN_val_omp here:n


Testing check_node.c.....................................
Please, give the node_id number:2
Now, the tau number:2
>> The Nodes that form a cluster with node_id 2 and tau 2:
    1,3,4,5,6,7,8,9,10,13,14,16,17,19,20,21,22,23,24,25,26,27,28,29,30,31,32,36,38,39,
40,41,44,45,46,48,49,50,51,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,71,
72,73,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,91,92,93,94,95,96,97,98,99,100,101,
102,103,104,105,106,107,108,109,110,111,112,113,115,116,117,118,119,120,121,122,123,
124,125,126,127,128,129,130,131,132,133,134,135,136,137,139,140,141,142,143,144,146,147,
148,149,150,151,152,155,156,157,158,159,161,162,163,164,165,166,167,168,169,170,171,
172,173,174,175,176,177,178,179,180,181,182,184,185,186,187,188,189,190,191,192,193,
194,195,196,197,199,200,201,202,203,204,206,207,208,211,212,213,214,217,218,219,220,
221,222,223,224,225,226,227,228,229,230,231,232,235,236,237,238,239,240,242,243,245,
246,247,248,249,250,251,252,253,254,257,258,259,260,261,262,263,264,265,266,268,269,
270,271,272,273,274,275,276,277,278,280,281,283,284,285,288,289,290,291,293,294,295,
296,297,298,299,300,301,302,303,304,306,307,308,309,310,311,312,313,314,315,317,318,
319,320,321,322,323,324,325,326,327,328,329,330,331,332,333,334,336,337,338,339,340,
341,342,343,344,345,346,347,0%
```

First, in lines 5-7, we test `read_graph_from_file1.c` that had CPU time of 0.042s and then the algorithm requests from the user if the printing representation is necessary. We did not print here due to its considerable size, but it was tested with smaller Connectivity Graphs, and all of them worked accordingly.

After, in lines 8-17, we test `create_SNN_graph1.c` (without OMP) that had elapsed time of 1.30s (line 9) and `create_SNN_graph1_omp.c` (with OMP) that had elapsed time of 0.68s for 4 threads (line 11). The speedup between them was 1.90 (line 13), which shows that the parallelization worked satisfying, almost doubling the serialized code's performance for this task. The serialized and parallelized arrays were compared and were precisely the same (line 15). In lines 16-17, the algorithm asked if the printings were necessary. We skipped this part because of the rouge size of the printings.

After, we tested `read_graph_from_file2.c` that had CPU time of 101.21s (line 21). This CPU time was greater than `read_graph_from_file1.c` because inside the former function, we sorted the values of the col_idx array for future performance gains. Also, in line 22, the algorithm requested from the user if the printing representations were necessary. We did not print here due to its considerable size, but it was tested with smaller Connectivity Graphs, and all of them worked accordingly.

In lines 24-34, we tested `create_SNN_graph2.c` (without OMP) that had elapsed time of 8.04s (line 25) and `create_SNN_graph2_omp.c` (with OMP) that had elapsed time of 5.50s for 4 threads (line 27). The speedup between them was 1.46 (line 29), which showed that the parallelization worked successfully, almost increasing in 50 percent the serialized code's performance for this task. The serialized and parallelized arrays were compared and were precisely the same (line 31). In lines 32-34, the algorithm asked if the printings were necessary. We skipped this part because of the rouge size of the printings.

The lines 37-53 tested `check_node.c` for given node_id of 2 and tau of 2. The cluster was represented in lines 40-53.

The conclusion we can evoke at the end of this project is that the parallelization using OpenMP brings a significant increase in the computations' performance, immediately after increasing the number of threads. The following plots illustrate this improvement according to the increasing of the number of the threads:

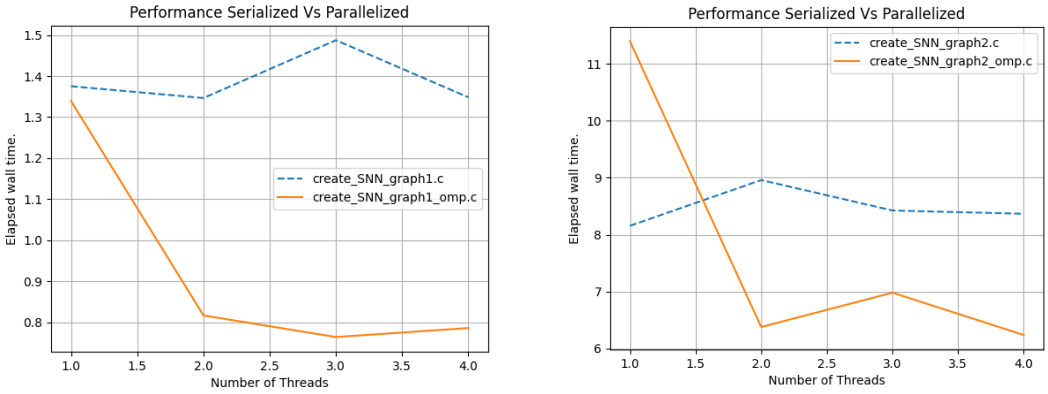

Figure 1: Performance of Serialized Vs Parallelized algorithms. Remember that Serialized algorithms only uses 1 thread for any test.

Figure 1 shows that the parallelized algorithm with OpenMP overcomes serialized code's performance, and as long as the computer power for more threads is available and higher, as the performance increases even more.

3 Bibliography

[1] Hager, Georg. Wellein, Gerhard. *Introduction to High Performance Computing for Scientists and Engeneers*; CRC Press. Boca Raton, Florida. 2011.

[2] Introduction to OpenMP - Tim Mattson (Intel). (Mar 30, 2021)

[3] Microsoft - OpenMP in Visual C++. (Mar 30, 2021)