IN4200 - High-Performance Computing and Numerical Projects
Home exam I

15806
(Dated: March 26, 2021)

I. ALGORITHMS

I will during the subsections below describe the idea behind each program implemented in the home exam project. There are six programs in total but the last one, being `test_programs.c`, does not have any special algorithm beside to test all previous programs. Note that both of the `read_graph_from_file#.c` files will read the input text file only once. The programs ending with 1 tend to use a more brute force fashioned approach by presenting a two-dimensional array which include possible connections between nodes. Programs ending with 2 implements a more memory storage friendly algorithm known as compressed row storage (CRS) format.

A. read_graph_from_file1.c

This program requires three inputs where two of them are to be allocated and defined within the program. These are the text file to be read `char *filename`, an integer representing the total number of nodes `int *N` in the text file and lastly a two-dimensional array `char ***table2D` storing all possible connections between nodes. Both `int *N` and `char ***table2D` is defined within this program.

We assume that all input text files are in equal format which means that the two first line may always be skipped. Hence, the program will start by reading the number of nodes and edges which is read from the third line. The number of nodes is stored in the input argument `N` so that it may be used later. The number of edges is stored locally within the program as `N_edges` only for detection of illegal edges in text file. The number of nodes `N` is also used for allocating the `table2D` array since it should be of size $\texttt{N} \times \texttt{N}$. The classification of row and column is given by the node id so that for instance node 10 can be linked to row 10 and column 10 separately.

When `table2D` is allocated, we will loop through all elements and initiate them as 0. The reason is that 0 means no connection between node i and j where i and j is the row and column index. Now that the allocation of the table is done, we are ready to read each line the text file representing an unique edge.

We have not assumed that illegal edges will not occur. By illegal, I mean edges which lies outside the scope of possible node id such as negative nodes or nodes that are greater than `N`. This is taken care of by the `if` block as shown as a pseudo code below:

```
if (node1<0 || node1>=N || node2<0 || node2>=N)
{
  printf("Illegal edge in file (line %d+5):
              (%d, %d)\n", i, int1, int2);
  printf("Excluded from table!");
  tot_edges = tot_edges - 1;
}
```

The block will print out the line from the text file that is illegal as well as both of the read node id. Note that we have an additional parameter `tot_edges` which is decreased by one if some illegal edge occurs. It is used for checking the difference between `N_edges` and `tot_edges` at the end of program. If there is a difference, the program will print it out in the terminal.

We have also not assumed that self-links (From node = to node) may not occur in the text file. This is done by implementing another `else if` block that checks if an edge contains two equal integers. A pseudo code is shown below:

```
else if (node1 == node2)
{
  printf("Illegal self-link in file (line %d+5):
                    (%d, %d)\n", i, int1, int2);
  tot_edges--;
}
```

Also, here the illegal line will be printed in the terminal as well as both of the read node id. We have in fact assumed that multiple edges does not have the same undirected edge so we do not need to consider this.

If both of the above `if` blocks are passed, we add the legal edge to the table. Note that the table should be diagonally symmetric. That is if node i is connected to node j, then node j is also connected to node i. Therefore we need to add two edges to the table for each legal edge. This is done as follows:

```
else
{
  (*table2D)[node1][node2] = 1;
  (*table2D)[node2][node1] = 1;
}
```

Note that the `else` block are for edges that surpass the previous `if` blocks. Both elements in the `table2D` that are linked to `node1` and `node2` are given the value 1.

B. read_graph_from_file2.c

Here we introduce the format of CRS. In contrast with the previous program, this requires four input in-

stead of three where the two-dimensional array is replaced by two one-dimensional arrays. Hence, the inputs are `char *filename`, `int *N`, `int **row_ptr` and `int **col_idx`. The two input arrays will be allocated an defined within this program. The `row_ptr` array will have a length of `N` + 1 and the `col_idx` array will have a length of 2`N_edges`.

Before we allocate these two arrays, we need to make sure that we do not include illegal edges. This approach is done equally to what we did in the program `read_graph_from_file1.c`. The difference now is that we are not adding the legal edges to a table, but rather storing the connections in two temporally arrays namely `int *count_nodes` and `char **count_connections`. The first array counts the occurrence of node i and j which is needed for defining the values in `row_ptr`. The second array is needed for adding all nodes connected to both node i and j and is used for reading later. Therefore both arrays should have length `N`. Note that `count_connections` needs to be allocated such that it can include all possible connected nodes. Therefore, each element is allocated with a threshold value of 10`N`. A pseudo code on how edges is added to these arrays is shown below:

```
else
{
  count_nodes[node1]++;
  count_nodes[node2]++;

  sprintf(num, "%d ", node2);
  strcat(count_connections[node1], num);

  sprintf(num, "%d ", node1);
  strcat(count_connections[node2], num);
}
```

The `else` block adds legal edges to the temporally arrays. The first thing we do in this block is to use both node id to add up the occurrence of `node1` and `node2`. Then, we use each node id as index to add the connected node as a string in the `count_connections` array. This is done by first convert the connected node id as a string and then add it to the array. This is used for reading the numbers later.

Now we have stored the number of occurrence of each node and stored all possible connections to each node as strings. We have also neglected illegal edges by reducing `N_edges` to `tot_edges`. Note that these two parameters can be equal in size which happens when no illegal edges occur. Then `row_ptr` and `col_idx` are allocated as size of `N` + 1 and 2`tot_edges` respectively. We then create a for loop iterating over all nodes and start by adding the values in `row_ptr` as shown below:

```
for (int node=0; node<*N; node++)
{
  (*row_ptr)[node+1] = (*row_ptr)[node] +
                       count_nodes[node];

  // more code reading the string array
}
```

The first index in `row_ptr` always start with 0 as there are no previous nodes occurring by definition. Then we add the next index element as the sum over the previous index element and the counted number of the previous node id occurrence. This will go on until we index over `N` + 1. The last index will then have the value corresponding the total value of edges. Now we need to add the values in `col_idx` looping over the same nodes. When reading each string element in the `count_connections` array, we convert each connection back to an int and then insert those values in the `col_idx` array.

The two temporal arrays are then deallocated as they are no longer needed before the end of the program.

C. create_SNN_graph1.c

This program requires three inputs which are `int N`, `char **table2D` and `int ***SNN_table`. The two first inputs has already been defined from `read_graph_from_file1.c`. The last input is to be allocated and defined within this program. We know that the table must be of size `N` × `N` and is therefore allocated accordingly. Each element is initialize with value 0.

We need to consider two assumption before running through every element in the `table2D` array. That is each node is not a shared nearest neighbor (SNN) with itself meaning that the diagonal contain zeros only, i.e. `SNN_table[i][j]` = 0 where $i = j$. Next we know that if node i is connected to node j, then node j must be connected to node i which means that the matrix is symmetric. So we only need to check either the upper or lower triangular of `table2D`. We check the upper triangular in our case.

A third loop is implemented as we need to check if node i and node j approves for a SNN, that is if both i and j is connected to node k. Within this third loop, we implement a number of statements in a `if` block before adding values to the elements in the `SNN_table` array. The first statement is that node i and node j is connected to each other. The second and third statements is that the third node k can not be equal to i or j. The final fourth and fifth statements make sure that both node i and node j is connected to node k, hence share a nearest neighbor. The triple for loop is then implemented as follows:

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

We note that the elements in the `SNN_table` array are only added when all statements are met.

Parallelization is added to this program as it is beneficial for this particular case. The parallelization is defined such that the outer for loop will split into the requested amount of threads. More of how do determine the number of threads are found in the README.md file. The reason for this placement is because every iteration through each node is independent of each other. A dynamic schedule is used as each iteration i in the outer loop will continuously decrease the length of the first inner for loop as i gets larger. A static schedule will result in a unbalance operation distribution between the threads. We also need to make sure that the indices i, j and k are private variables within each thread.

D. create_SNN_graph2.c

This program requires four inputs which are `int N`, `int *row_ptr`, `int *col_idx` and `int **SNN_val`. The three first inputs are predefined from the `read_graph_from_file2.c`. The last input `SNN_val` will be allocated and defined in this program. We know that `SNN_val` has the same length as `col_idx` which is `2N_edges` and can be found as the last element in the `row_ptr` array. Each element is then filled with 0.

Next we iterate over every node i and then iterate over its connected node two times j and k. The reason for this approach is that we want to check if node j is connected to any of the connected nodes to k. Therefore, we need another for loop that iterates over the connected nodes to k, namely h. What we want to check is that if $j = h$, we know that node i and node k is a SNN pair. Also, we need to make sure that $j \neq k$ as we should not check for SNN within the same node. The for loops is shown as below:

```
for (i=0; i<N; i++)
{
 for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
 {
  for (k=row_ptr[i]; k<row_ptr[i+1]; k++)
  {
   if (col_idx[j]!=col_idx[k])
   {
    for (h=row_ptr[col_idx[j]];
         h<row_ptr[col_idx[j]+1]; h++)
    {
     if (col_idx[k]==col_idx[h])
     {
      (*SNN_val)[j]++;
     }
    }
   }
  }
 }
}
```

Note that we add values in the `SNN_val` array for index j. This is because we want this array to be equally ordered as the `col_idx` array.

This program is parallelized as it is benifical here as well. Each iteration of index i is independent of each other implying that parallelization should be implemented. Since the length of all three inner for loops will vary in size, the schedule should be dynamically to prevent unbalanced operation sharing between threads. The parallelization demands the indices i, j, k and h to be private variable which is handled for when the parallelization is initiated.

E. check_node.c

The last program requires six inputs which are `int node_id`, `int tau`, `int N`, `int *row_ptr`, `int *col_idx` and `int *SNN_val`. The four last inputs are all defined and allocated through `read_graph_from_file2.c` and `create_SNN_graph2.c`. The two first inputs are chosen by the user. `node_id` is an integer representing a node to check for if it is possibly connected to a cluster. `tau` represents a threshold value of what defines a cluster. Since these two inputs are chosen by user, two arguments are required when executing this program. More information on this is found in the README.md file.

The program starts with allocating an array `cluster` of size `N` which all holds the value of -1 except for the input `node_id` which holds the value 0. The values in the array are coded in a way that -1 means that this particular node will not be considered as part of the cluster. Value of 0 means that this node has been approved as part of the cluster but need to check all its edges for additional nodes in the cluster. Value of 1 means that this node and its neighbors has been checked.

Now that we have this numeric coding implemented, we start by introducing a while loop that will run as long as `cluster` has at least one element given the value of 0. We then iterate over the array and check for nodes i containing the value 0. If true, this index is given value 1 and then we check if this node has an SNN value $\geq \tau$ with any connected nodes. If this statement is true, the connected node is given the value of 0 in the `cluster` array. To prevent the while loop to run for infinity, we need to check if the neighboring node contains the value -1 in the `cluster` array. A pseudo code is shown below:

```
while (check_nodes==1)
{
  check_nodes = 0;
  for (i=0; i<N; i++)
  {
    if (cluster[i]==0)
    {
      cluster[i] = 1;
      for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
      {
        if (SNN_val[j] >= tau &&
            cluster[col_idx[j]] == -1)
        {
          cluster[col_idx[j]] = 0;
            check_nodes = 1;
        }
      }
    }
  }
}
```

Note that for every iteration in the while loop, we set `check_nodes=0`. Only if the required conditions are met, will the while loop keep running as seen within the innermost loop as we set `check_nodes=1`.

F. test_programs.c

This final program is an auxiliary program only intended for testing all the previous programs. Therefore it does not contain any particular algorithm of interest. Note that the previous programs excludes the `main` function implying that `test_programs.c` is the only program that needs to compile. Because of this, we need to make sure that all programs is imported into `test_program.c`. We also need to import necessary header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

#include "read_graph_from_file1.c"
#include "read_graph_from_file2.c"
#include "create_SNN_graph1.c"
#include "create_SNN_graph2.c"
#include "check_node.c"
```

`stdio.h` is used for printing various outputs in the terminal, `stdlib.h` is used for allocating and deallocating arrays, `string.h` is used for string purposes (adding string to other string) and `omp.h` is used for parallelization.

How to compile `test_programs.c` and execute the other programs is further explained in the README.md file.

II. RESULTS

To make sure that the programs is executed correctly, we compared the outputs with the example given in the home exam description. This was done by creating a connectivity graph file that contains the number of nodes and edges in the header and including all edges that describes the connectivity shown in Figure 1.

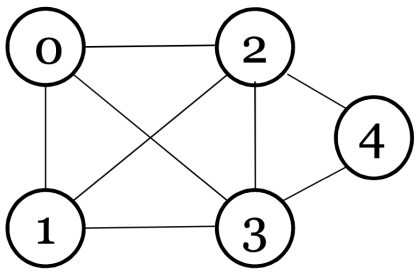

Figure 1. Connectivity graph with five nodes and eight edges.

We also compared our results with the SNN connections shown in Figure 2.

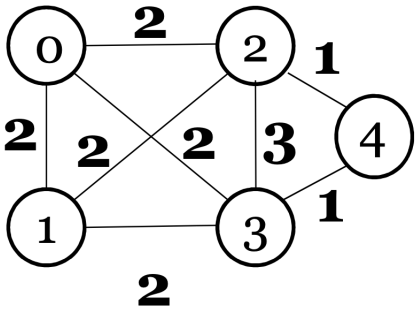

Figure 2. Shared nearest neighbor graph given the connections in Figure 1.

Table I. Machines used when executing programs.

Machine	CPU type
Private	Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8
UiO	2x AMD EPYC 7601, 2.2GHz, 32-core

When we get outputs from using both of the data storage format that match Figure 1 and Figure 2, we assume that all other connectivity graphs are correctly read and stored in the respective arrays.

I have used the 'facebook_combined.txt' file for comparing the efficiencies between the two format implementation with serial and parallelized coding. This file contains 4039 nodes and 88234 edges which states that $N_{edges} \ll N^2$. We should then consider CRS to be faster than the 2D table format. The file can be downloaded from https://www.uio.no/studier/emner/matnat/ifi/IN3200/v21/teaching-material/facebook_combined.txt. During the test runs, I have used my private laptop and one of the machines provided by UiO. The specs of the two machines are shown in Table I.

The speed performances are shown in Table II.

Table II. Performance in terms of time used of the different programs. Note that test 1 and test 2 corresponds to the two different memory storage format used. Storing a 2D table is test 1 and the CRS format is test 2. All cells are given in seconds. Maximum threads is 8 and 128 on private and UiO machines respectively.

Machine	Single thread	Maximum threads
Private (test 1)	74.867	18.581
Private (test 2)	5.903	1.592
UiO (test 1)	83.518	1.407
UiO (test 2)	6.472	0.395

We observe from Table II that using CRS is considerably faster than storing a 2D table. The factor of increased performance is over 11 in all cases except for when using the UiO machine with 128 threads. The relatively small factor of performance boost may be because of allocating tasks between threads kills some time. We also note that using a single thread performed better on my private computer.

Finally, we check if an arbitrary node in the 'facebook_combined.txt' may be part of a cluster with some threshold value τ. By opening the text file and looking at some of the first edges, we note that node 0 is connected with many other nodes. By testing the `check_node.c` program on node 0 with $\tau = 50$, we get that the node is in a cluster with additional 18 nodes, forming a cluster with a total of 19 nodes. The other nodes varies from 9 to 322 in id value. We do not have any further information about the nodes but we may assume that the low id of the nodes relative to the total number of nodes can be related to an unidentified factor somehow.