IN4200: Home Exam 1

Candidate number: 15804

Abstract

In this note, different strategies are suggested for how to read and analyze an ASCII-file containing connectivity data, with emphasis on code performance. First, information is stored as a 2D table, and a shared nearest neighbours (SNN) graph is made with similar dimensions. The second method uses the compressed row storage (CRS) method, both to store information and to create an SNN graph. In the latter case, cluster analysis is performed from a search key, with the breadth-first search (BFS) algorithm as inspiration. The note is to be read as a comment of the C++ provided.

I Introduction

A connectivity graph is a data set consisting of individual data objects. In this context, these objects will be referred to as *nodes*, with a unique node ID. In a connectivity graph, nodes are directly connected to each other by *edges*. In a general, the edges can be unidirectional, directional and/or weighted. Here, an edge is unidirectional, connecting both nearest neighbours (NNs) to each other.

A similarity measurement between NN pairs can be how many unique NNs are shared between them - so called shared nearest neighbours (SNNs). On the basis of SNNs, connected nodes can be placed in a subset, where all NN pairs within the subset has SNNs above or, equal to, a certain treshold τ. For simplicity, all the above quantities are assumed to be positive integer values. An example of a connectivity graph is displayed in fig. 1.

The text-file to be read is assumed to have the ASCII format

```
Undirected graph:  facebook combined.txt
# A connectivity graph representing social circles from Facebook
# Nodes:  5 Edges:  8
# FromNodeId ToNodeId
0 1
```

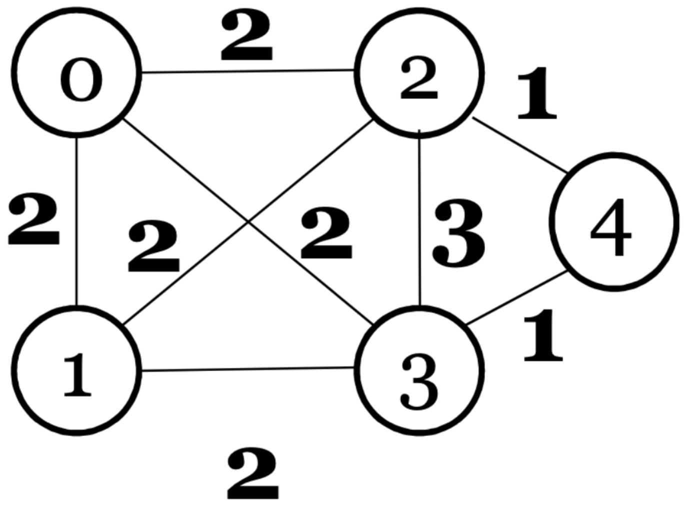

Figure 1: **Connectivity Graph**: Nodes with ID 0-4 are connected through unidirectional edges denoted by lines. Each pair has shared nearest neighbours (SNNs) as shown by the uncircled integers associated with a specific edge/node pair.

```
0 2
[...]
```

The following sections contain explanations to the C++ code provided for how to create connectivity graphs and clusters from the textfile format given above. That is, either through 2 dimensional constructs, or by CRS format [1]. Performance considerations are given special emphasis in each case. The code should be sufficiently commented to understand intuitively, thus only the breadth first algorithm [2] is provided.

II 2D Format

II.I Read ASCII formatted file

First the text-file is read by the function `read_graph_from_file1(char *filename, int *N, char ***table2D)`, with the number of nodes `*N` and the connectivity graph `***table2D`, as pointers to be assigned values. Since the table has dimension $\mathbb{R}^{NxN}$ and only contains 0's and 1's, using built-in type *char* (1 byte) is advantageous. *chars* takes up less storage than other data types. `sscanf` and `fgets` from `stdio.h` are used to read off the information from the text file with exception handling.

Before reading off the text file, a double for loop, with loop unrolling of columns (inner loop), fills in 0's. The unrolling decreases loop overhead, and is done over columns to avoid strided access, due to C++'s row-major storage [1]. With unrolling 4 indices, 4*x* the work is done per iteration, compared to w/o unrolling. This serial code optimization is done frequently in following tasks, for the purpose of optimizing serial code. After allocating and assigning 1s, the resulting matrix should be symmetric, with 0's on its diagonal (no self-edges).

II.II 2D SNN graph

A SNN graph is made from `table2D` with `create_SNN_graph1(int N, char **table2D, int ***SNN_table)`, with parallel programming for utilizing the multi-core hardware system (cache coherent non-uniform memory access (CC-NUMA) structure) [1]. Here one could use a matrix product, together with the symmetry properties of `table2D`, resulting in floating point operations (FLOPs) of the order $O((N/2)^3)$.

However, with a large connectivity graph, the probability for many 0's grows. That is, under the assumption of connectivity graphs being of the purpose of investigating heterogeneity in a data set or population data. Thus, a strategy of temporary arrays and more loop overhead is chosen, avoiding looping over many non-edged nodes. This might not be the best overall strategy, given the symmetry properties of `table2D`, and since more memory traffic is needed in each iteration in the former method.

III CRS Format

III.I Read ASCII formatted file

The function `read_graph_from_file2 (char *filename, int *N, int **row_ptr,int **col_idx)` reads the text-file provided and stores the information in CRS format. `row_ptr` is the number of accumulative connections (length nodes + 1), whereas `col_idx` lists nodes connected to node i (length 2 x edges). Data elements are stored as integers (4 bytes) (note: Type unsigned long int may be more suited to data sets with only positive integers). Yet, there is large advantage with a compressed storage for a potentially dense matrix.

Two temporary arrays (`from` and `to`) are used while reading the text file, and two more arrays (`sum` and `indx`) are used to place edged nodes correctly in `row_ptr` and `col_idx` respectively. The serial code relies heavily on loop unrolling. At last the quicksort algorithm sorts the nodes from lowest to highest, since the data might be unordered. The idea is that the cost of sorting, will be later rewarded by more contiguous memory access.

III.II 1D SNN graph

As before, parallel programming with OpenMP is used to find SNNs, with the function `create_SNN_graph2(int N, int *row_ptr, int *col_idx, int **SNN_val)`. Here each element in `SNN_val` contains information of the SNNs between the nodes accessed in `col_idx[i]`, where $i \in$ [`row_ptr[node]`,`row_ptr[node+1]`). Since the mechanics of the for loops are not that different from the alternative chosen for the 2D table, a short comment on parallel programming follows.

A consideration to be made when doing work in parallel, is how to initialize memory storage. Using dynamic memory allocation, a first-touch policy is in place [1]. Thus to miminize NUMA traffic and avoid cache incoherence, the candidate has tried to initiate each chunk of memory with the thread doing the actual work on it in parallel regions of the code. Thus looping over the outer for loop seemed the best option, and applying loop unrolling to optimize serial regions within each thread. 4 unrolls were specified to avoid long compilation time and OpenMP overhead.

III.III Clusters

To find clusters can be a demanding task, and is suited for work being spread over several threads, when clustering entire graphs. However, here, one needs to find a cluster associated with a search key (a given node id). It can be solved with a modified breadth-first search (BFS) algorithm (algo. 1).

Algorithm 1 Breadth-First Search (BFS): The basic outline of an altered BFS [2] algorithm. It accesses the search key, and put discovered nodes in a queue to explore each layer until no newly discovered nodes fulfill the threshold criteria SNN's $\geq \tau$.

```
while nodes in queue true do                          ▷ Run as long as new nodes enter queue
    reset queue indices $i, k = 0$;
    while nodes in current queue do                   ▷ nodes added to queue in prev. depth
        q = queue[i];                                 ▷ Store as variable
        Delete q from queue[i];                       ▷ Make room for future discovered nodes
        for j in range(row_ptr[q],row_ptr[q]) do      ▷ Go through nodes edged to node $q$
            If SNN_val[j] $\leq \tau$ and cluster[col_idx[j]] == 0 do;      ▷ cluster criteria
            cluster[col_idx[j]] = 1;                  ▷ col_idx[j] is the discovered node in cluster
            queue[k] = col_idx[j];                    ▷ Add to queue
            k++;
        end for
        i++;
    end while
    If No new nodes from previous depth do
    nodes in queue false                              ▷ exit while $\rightarrow$ finished clustering
end while
```

IV Time performance

After three runs for each function, the time performance is shown in tab 1. As expected, the CRS format performed best in regards to time. Hence it proves to be a more effective storage, especially for dense matrices, which is often the case for heterogeneous data and connectivity graphs.

Table 1: **Time Benchmarks**: CPU time (ms) average after three runs for connectivity graph functions. 4 cores were active in creating SNN-graphs for both the 2D SNN table and CRS storage. Search key for clustering: 0, $\tau = 1$. Optimization flag: -Ofast.

	Read [ms]	Create SNN [ms]	Clustering [ms]
Case 1: 2D			
	50.20	1410.0	–
Case 2: CRS			
	26.50	1399.0	7.0

———————————— A NOTE FOR THE EXAMINER ———————————-

More specific considerations could have been made regarding FLOPs/second and Words/second. However, the candidate chose to focus more on broad concepts of memory allocation and access, effective serial and parallel code, and to give a discussion of optimal choices for the code. The candidate asks kindly for more specific evaluation criteria in the future. Thanks.

References

[1] Georg Hager; Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Press, 2011.

[2] Edward F. Moore. The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*, page 285–292, 1959.