IN3200/IN4200: Dissecting Home Exam 1

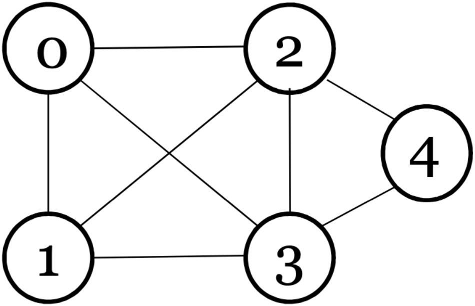
Spring 2021

Objectives of home exam 1

- Hands-on experience with two common data storage schemes
- Implementation (and testing) of serial functions
- OpenMP parallelization
- Reflections on performance related issues

Connectivity graph

- Nodes represent individual data objects
- Edges represent *direct-connections* between pairs of "nearest neighbors"
- Symmetry: if node u is a nearest neighbor of node v, then v is a nearest neighbor of u



2D table as storage format for connectivity graph

A 2D table of values 0 and 1 (implemented as a 2D array of type `char**`)

row 0	0	1	1	1	0
row 1	1	0	1	1	0
row 2	1	1	0	1	1
row 3	1	1	1	0	1
row 4	0	0	1	1	0

Note: The number of values of 1 in the 2D table is twice the number of edges in the connectivity graph

Note: `char` is also an integer type in C, one byte per value

Read a connectivity graph from file (version 1)

```
void read_graph_from_file1 (char *filename, int *N, char ***table2D)
{
  int num_nodes, num_edges, i, w1, w2; char line[100], str1[50], str2[50];
  char **table;

  FILE *fp = fopen(filename, "r");
  fgets (line, 100, fp);  // skip line 1
  fgets (line, 100, fp);  // skip line 2
  fgets (line, 100, fp);
  sscanf (line, "# %s %d %s %d",str1,&num_nodes,str2,&num_edges);
  fgets (line, 100, fp);  // skip line 4

  table = (char**)malloc(num_nodes*sizeof(char*));
  table[0] = (char*)calloc(num_nodes*num_nodes, sizeof(char));
  for (i=1; i<num_nodes; i++)
    table[i] = &(table[0][i*num_nodes]);

  for (i=0; i<num_edges; i++) {
    fgets (line, 100, fp); sscanf (line, "%d %d", &w1, &w2);
    if (w1!=w2 && w1>=0 && w1<num_nodes && w2>=0 && w2<num_nodes)
      table[w2][w1] = table[w1][w2] = 1;
  }
  fclose(fp);

  *N = num_nodes; *table2D = table;
}
```

Compressed row storage (CRS) format for connectivity graph

Implemented as two arrays of integer values, for example,

`row_ptr`: *0, 3, 6, 10, 14, 16*
`col_idx`: *1, 2, 3, 0, 2, 3, 0, 1, 3, 4, 0, 1, 2, 4, 2, 3*

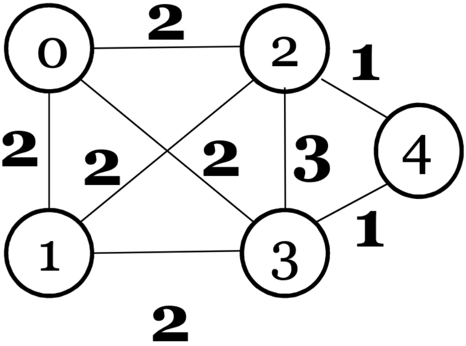
Length of `row_ptr`: number of nodes in the connectivity graph + 1
Length of `col_idx`: twice the number of edges

Read a connectivity graph from file (version 2)

- Get the info about the number of nodes
- Allocate array `row_ptr`
- Read through the input file **in two rounds**
- Round 1:
  - Count how many nearest neighbors each node has
  - Prepare the values inside array `row_ptr`
- Allocate array `col_idx`
- Round 2:
  - Parse the info of each edge, insert *two entries* suitably into `col_idx`

SNN graph

For each pair of nearest neighbors u and v, the number of their *shared nearest neighbors* (SNNs) is defined as the number of other nodes that are directly connected with both u and v.



Naive implementation of create_SNN_graph1

```
void create_SNN_graph1 (int N, char **table2D, int ***SNN_table)
{
  int i,j,k,snn;

  int **s_table = (int**)malloc(N*sizeof(int*));
  s_table[0] = (int*)malloc(N*N*sizeof(int));
  for (i=1; i<N; i++)
    s_table[i] = &(s_table[0][i*N]);

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      if (table2D[i][j]==1) {
        snn = 0;
        for (k=0; k<N; k++)
          if (table2D[i][k]==1 && table2D[j][k]==1)
            ++snn;
        s_table[i][j] = snn;
      }
      else
        s_table[i][j] = 0;

  *SNN_table = s_table;
}
```

Two improvements of `create_SNN_graph1`

```
// ....

for (i=0; i<N-1; i++) {
  s_table[i][i] = 0;

  for (j=i+1; j<N; j++)
    if (table2D[i][j]==1) {
      snn = 0;
      for (k=0; k<N; k++)
        snn += table2D[i][k]*table2D[j][k];
      s_table[i][j] = s:table[j][i] = snn;
    }
    else
      s_table[i][j] = s_table[j][i] = 0;
}
s_table[N-1][N-1] = 0;
```

- Use symmetry of SNN pairs (saving 50% computation)
- Use the integer values (0 & 1) in `table2D` directly in computation

Naive implementation of create_SNN_graph2

```
void create_SNN_graph2 (int N, int *row_ptr, int *col_idx, int **SNN_val)
{
  int u,v,j, len1,len2;
  int *s_val = (int*)malloc(row_ptr[N]*sizeof(int));

  for (u=0; u<N; u++) {
    len1 = row_ptr[u+1]-row_ptr[u];
    for (j=row_ptr[u]; j<row_ptr[u+1]; j++) {
      v = col_idx[j]; len2 = row_ptr[v+1]-row_ptr[v];
      s_val[j] = num_shared_indices(len1,&(col_idx[row_ptr[u]]),
                                    len2,&(col_idx[row_ptr[v]]));
    }
  }
  *SNN_val = s_val;
}

int num_shared_indices (int len1, int *indices1, int len2, int *indices2) {
  int k,m,idx,num=0;
  for (k=0; k<len1; k++) {
    idx = indices1[k];
    for (m=0; m<len2; m++)
      if (indices2[m]==idx)
        ++num;
  }
  return num;
}
```

Improvements of `create_SNN_graph2`

- Sort the `col_idx` segment for each row beforehand, using for example *quick-sort* ("extra work" , but it's worth it!)
- Counting shared indices between two sorted index lists can make use of *binary search* (which is much faster)
- Use the symmetry of SNN pairs (saving 50% computation)

```
for (u=0; u<N; u++) {
  len1 = row_ptr[u+1]-row_ptr[u];
  for (j=row_ptr[u]; j<row_ptr[u+1]; j++) {
    v = col_idx[j];
    if (v>u) { // using the symmetry of SNN pairs
      len2 = row_ptr[v+1]-row_ptr[v];
      s_val[j] = num_shared_indices_sorted(len1,&(col_idx[row_ptr[u]]),
                                           len2,&(col_idx[row_ptr[v]]));
      // find the position of u inside the col_idx segment of row v
      // u_pos = binary_search ....
      s_val[u_pos] = s_val[j];
    }
  }
}
```

OpenMP parallelization

```
#pragma omp parallel for private(j,k,snn)
for (i=0; i<N-1; i++) {
  s_table[i][i] = 0;

  for (j=i+1; j<N; j++)
    if (table2D[i][j]==1) {
      snn = 0;
      for (k=0; k<N; k++)
        snn += table2D[i][k]*table2D[j][k];
      s_table[i][j] = s:table[j][i] = snn;
    }
    else
      s_table[i][j] = s_table[j][i] = 0;
}
```

- Straightforward OpenMP parallelization of the outer-most loop (no danger for race conditions)
- Important to mark some of the variables as *thread-private*
- Load imbalance is a potential problem, should use small *chunksize* or *dynamic* scheduler
- Same parallelization strategy for `create_SNN_graph2`

"Clustering" (only relevant for IN4200 students)

```
void check_node (int node_id, int tau, int N,
                 int *row_ptr, int *col_idx, int *SNN_val)
{
  char *in_cluster = (char*)calloc(N, sizeof(char));
  in_cluster[node_id] = 1;  // mark the node

  traverse_one_row (tau, node_id, row_ptr, col_idx, SNN_val, in_cluster);

  // print nodes whose "in_cluster" values are non-zero
  // ....
}

// a recursive function for depth-first search
void traverse_one_row (int tau, int row_nr, int *row_ptr, int *col_idx,
                       int *SNN_val, char *in_cluster)
{
  for (int i=row_ptr[row_nr]; i<row_ptr[row_nr+1]; i++)
    if (in_cluster[col_idx[i]]==0 && SNN_val[i]>=tau) {
      in_cluster[col_idx[i]] = 1;   // mark the node
      traverse_one_row (tau, col_idx[i], row_ptr, col_idx,
                        SNN_val, in_cluster);
    }
}
```