CMSC 502: Assignment 1
Due Date: Mon Oct 9th, 2018; Total points: 25

Parallel TSP using MPI

Your assignment is to compare the serial TSP solution (5 points), threaded TSP solution (10 points) and the MPI-based TSP Solution (10 points). In your report, compare the performances of these different implementations.

You MUST use a dynamic programming (DP) solution to solve the individual TSP subproblems.

You will be given a set of points (x,y pairs); come up with the distances between the points. Use a distance threshold (user input) to decide whether the edge will be present or not. This generates the adjacency matrix for your undirected graph.

Results and Report

I expect that you will execute timing runs. From these you can prepare plots showing speedup and accuracy (in terms of length of the tour) for parallel versions for several sizes.

Deliverables

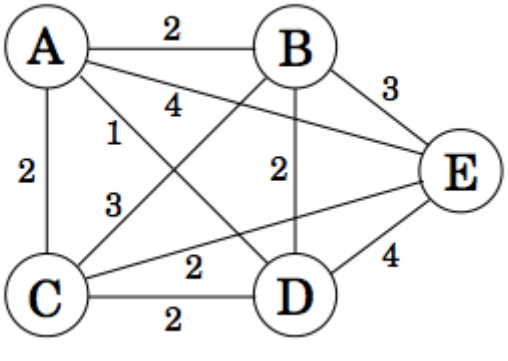
1. Source code(s)
2. Written report describing results. I expect a few pages with graphs or tables along with paragraphs describing your results and conclusions.

Serial TSP

The traveling salesman problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Denote the cities by *1*,...,*n*, the salesman's hometown being *1*, and let $D = (d_{ij})$ be the matrix of intercity distances. The goal is to design a tour that starts and ends at *1*, includes all other cities exactly once, and has minimum total length. The figure below shows an example involving five cities.



Let's dive right into the DP. So what is the appropriate sub-problem for the TSP? In this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1 as

required, have visited a few cities, and are now in city *j*. What information do we need in order to extend this partial tour? We certainly need to know *j*, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them. Here, then, is an appropriate sub-problem.

For a subset of cities $S \subseteq \{1,2,...,n\}$ that includes *1*, and $j \in S$, let $C(S,j)$ be the length of the shortest path visiting each node in S exactly once, starting at *1* and ending at *j*.

When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot both start and end at *1*.

Now, let's express C(S,j) in terms of smaller sub-problems. We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}. We must pick the best such i:

$$C(S,j) = \min_{i \in S, i \neq j} C(S-\{j\},i) + d_{ij}$$

The sub-problems are ordered by |S|. Here's the code.

```
C({1},1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
        C(S,1) = ∞
        for all j∈S, j≠1:
            C(S, j) = min{C(S−{j},i)+d_ij: i∈S, i≠j}
return min_j C({1,...,n},j)+d_j1
```

There are at most $2^n.n$ sub-problems, and each one takes linear time to solve. The total running time is therefore $O(n^2.2^n)$.

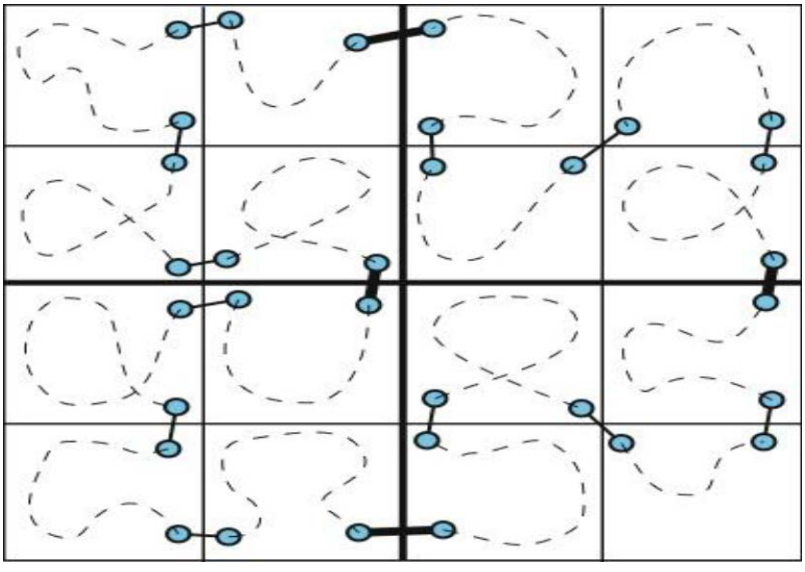
Threaded-TSP

Use the same set-up as MPI-based TSP shown below; instead of sending each block to a processor, send them to different threads.

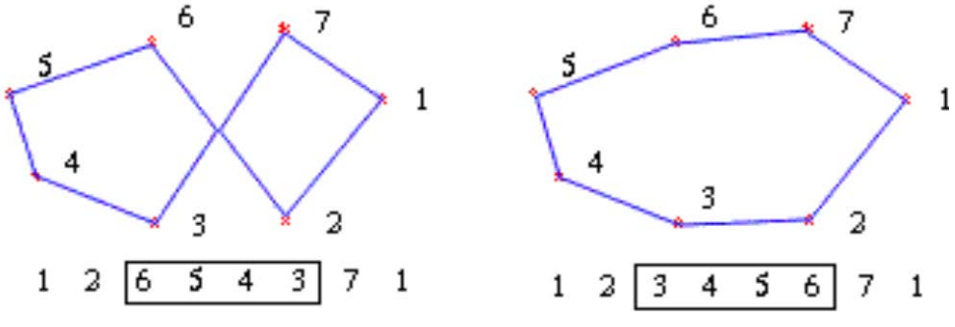
MPI-based TSP

Create 1 master process and n-1 slave processes. The master process divides the whole region into a square number of blocks and sends the city information in the block to the corresponding slave process. Each slave process takes care of one block and, in this way, the TSP algorithm is executed in parallel. After all the work is done, every slave process sends the result back to the master process.

The main idea of this algorithm is to exploit the fact that nodes are randomly distributed in Euclidean space: allocate geometric regions to different processes, and then stitch the regions together. The stitching needs to be accomplished using the nearest neighbor concept.



Handling inversions: For each process, due to the fact that the inversion takes place where the tour happens to have a loop rather than an intersection, then the loop is opened and the salesman is guaranteed a shorter tour. Therefore, if the process find there is an intersection (cross) in the tour (left figure), it will invert the tour to make an open loop (right figure).



Finding start-end points for each block:
The all-pair-shortest path algorithm gives you n x n matrix at each block (considering each block has n points to consider). So potentially, we have *b!* cases to consider and find the optimal stitching mechanism. Figure out a way of doing this in quicker time (this may yield non-optimal results).

Discussion of MPI in C
Before proceeding, a description of MPI's basic operation is in order. A C MPI program progresses through several stages before completion. In short, the stages can be summarized as follows:

Include mpi.h
Initialize the mpi environment (Parallel Code starts here)
Do the work and pass information between nodes by using MPI calls
Terminate the mpi environment (Parallel Code ends here)

MPI accomplishes this by sorting the available nodes into an object called a group. The group is referenced by another object which is tied to it - called a communicator. Although it is possible to use more than one communicator and set up multiple groups, for most purposes it is acceptable to use just one communicator and one group. For this assignment, you can use MPI_COMM_WORLD (the predefined communicator) and it is associated with a group which contains all the MPI processes.

Any given process can be further identified by its rank. Every process within a particular group/under a particular communicator has a unique integer rank. These ranks are numbered starting at 0 and are contiguous. For example, if we are dealing with eight processes, they will be ranked from 0 to 7.

An overview of some of the common MPI commands as they appear in the example program follows.

```
#include <mpi.h>
.
.
.
int main( int argc, char* argv[])
{
.
.
.
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nthreads);

if (rank == 0)
{
.
.
.
//read cities
MPI_Bcast(&ncities, 1, MPI_INT, 0, MPI_COMM_WORLD);
// assign sub cities for each of the blocks.
// Merge individual TSPs into the final route; consider inversions
MPI_Finalize();
}
.
.
.
```

Let us take a look at how the MPI mechanism operates. After the variables are defined, the MPI environment is initialized with MPI_Init. From this point onward, we now have multiple versions of the code running - this is where the parallelism has begun. And so each of those versions (processes) needs to know what its rank is and also how many other processes are out there. This is assigned by use of MPI_Comm_rank and MPI_Comm_size.

From this point on, the rank 0 process serves as a sort of master process. For example, the rank 0 process sends information to the other processes by use of MPI_Bcast and also sends information to other processes by way of the MPI_Send command. When those processes have completed work on that information, they send the information back to the rank 0 process which combines it into the final result (using Merge-TSP).

The lines of code marked off in red in the above example are MPI specific. Further commentary on their functions follow.

#include <mpi.h> - This include statement needs to go into every module that makes an MPI call.

int MPI_Init(int *argc, char ***argv) - Initializes the MPI execution environment. MPI_Init needs to be called prior to any other MPI functions, and therefore is required in every C program that uses MPI. MPI_Init must be called only once. (Note - MPI_Init obtains its arguments from the mpirun command. The mpirun command provides the number of nodes and a list of hosts to the MPI_Init command).

```
int main(int argc, char *argv[])
{
... ...
MPI_Init(&argc,&argv);
... ...
}
```

int MPI_Comm_rank (MPI_Comm comm, int *rank) - Determines the rank of the calling process (and places it at the address *rank). (A rank can be anywhere from 0 up to the number of processors). The rank will also vary based upon the communicator used (in our program, the only communicator used is MPI_COMM_WORLD). A process can have one rank under one communicator, but a different rank under another communicator.

int MPI_Comm_size (MPI_Comm comm, int *size) - Determines the number of processes in the current group of comm and places that integer at the address *size. In our example, the only communicator used is called MPI_COMM_WORLD.

int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm) – You have to transmit a message from the process with rank 0 (as associated with the MPI_COMM_WORLD communicator) to all the other processes of that group. MPI_Bcast defines its variables in this way: *buffer is the starting address of the buffer, count represents the number of items in the buffer, datatype represents the kind of datatype in the buffer, root is the rank of the broadcasting root, and comm is the communicator.

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) - A common command for sending data via MPI. buf is the initial address of the send buffer, count provides the number of elements (nonzero integer), the datatype specifies exactly what datatype is in use in the buffer, dest is the rank of the destination process, tag is the message tag, and the final parameter is the communicator.

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status) - A common command for receiving data via MPI. buf is the initial address of the receive buffer, count provides the number of elements (nonzero integer), the datatype specifies exactly what datatype is in use in the buffer, source is the rank of the destination process, tag is the message tag, and the final parameter is the communicator.

MPI_Finalize(); - This command terminates the MPI execution environment. Therefore it needs to be the last MPI command called. This command needs to be included in every MPI program to ensure that all resources are properly released.

A great deal of additional information is available on the net on the specifics of MPI, e.g., https://computing.llnl.gov/tutorials/mpi/#Getting_Started