

CMSC 502: Assignment 1

Due Date: Tuesday Oct 5th, 2021; Total points: 100

Our goal is to find optimal routes for individuals during pandemics. The objective is to minimize exposure to infected individuals. In a simplistic set-up, this problem can be mapped to a traveling salesman problem, where an individual starts from a particular city, visits every other city exactly once and then comes back to the starting point (closed loop).

Parallel TSP using MPI and Multi-threading

Your assignment is to compare the serial TSP solution (20 points), corresponding threaded TSP solutions (40 points) and the MPI-based TSP solution (40 points) for the pandemic-route planning problem. In your report, compare the performances of these different implementations. **You MUST use an Euclidean minimum spanning tree (EMST) solution to solve the individual TSP subproblems.**

Generate a set of points (x,y pairs) randomly (you can use the shell script uploaded on Canvas or generate your own city coordinates). Use another random number generator to assign the infection probability at each city (total infected/total population). Next assign the edge weights to the graph of cities (complete graph) as the product of the infection probabilities of the source and destination nodes; thus, for an edge $s \rightarrow d$, the edge weight is $p_s \cdot p_d$. This is a simple way of creating an undirected graph with cities as nodes and infection probabilities as the edge weights. This will generate 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

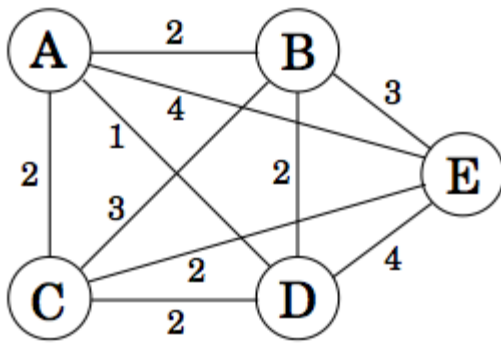
1. Source code(s)
2. Written report describing results. I expect a few pages with graphs or tables along with paragraphs describing your results and conclusions.

Serial TSP

The traveling salesman problem

A traveling salesman is getting ready for a big sales tour during the pandemic. 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 infection probabilities between cities, what is the best order in which to visit them, so as to minimize the overall infection probability?

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



EMST:

An approximation used to produce a fairly good tour is called the **Euclidean minimum spanning tree** or **EMST**. EMST in two dimensions, given n points, is to actually construct the complete graph on n vertices, which has $n(n-1)/2$ edges, compute each edge weight by finding the infection probability between each pair of points, and then run a standard minimum spanning tree algorithm (such as Prim's algorithm or Kruskal's algorithm) on it. Since this graph has $\Theta(n^2)$ edges for n distinct points, constructing it already requires $\Omega(n^2)$ time. This solution also requires $\Omega(n^2)$ space to store all the edges.

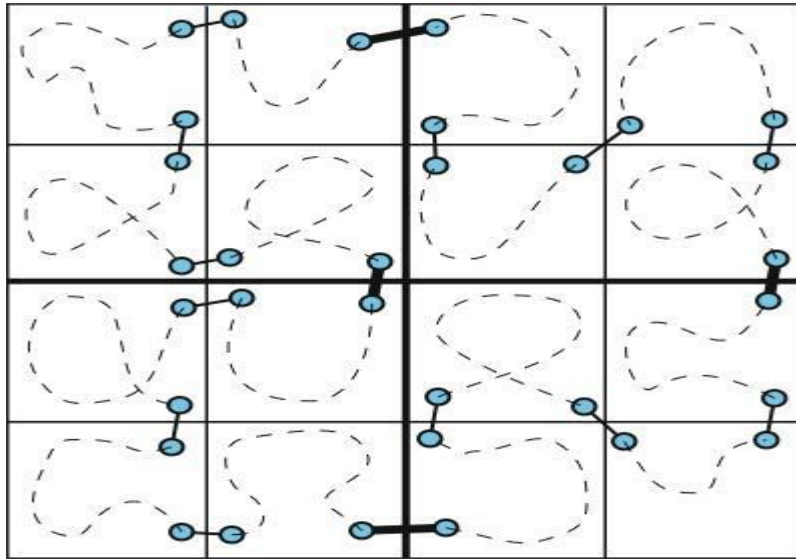
Threaded-TSP

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

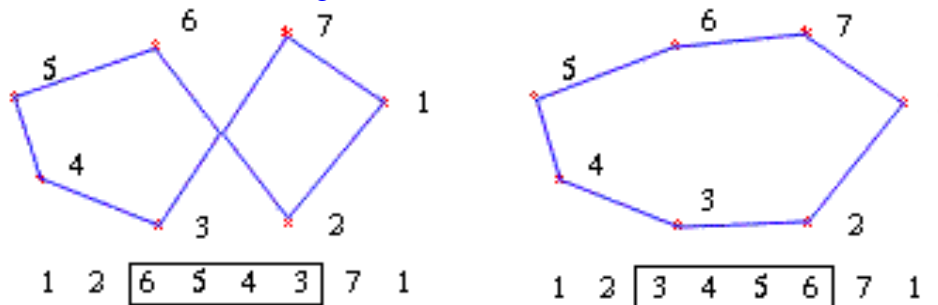
Create 1 master process and $n-1$ slave processes. The master process divides the whole region into a square number of blocks and sends the city information in the block to the corresponding slave process. Each slave process takes care of one block and, in this way, the TSP algorithm is executed in parallel. 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. Use any algorithm of your choice to implement the stitching of individual (open) TSPs.



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 less dangerous tour. Therefore, if the process finds there is an intersection (cross) in the tour (left figure), it will invert the tour to make an open loop (right figure).

See this for additional information: <https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect>



Finding start-end points for each block:

The all-pair-shortest path algorithm gives you $n \times n$ 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