Assignment3
Student Name: Yi-Ting, Hsieh(a1691807)

Objective:

To gain an understanding of the Message Passing Interface C implementation.

Solving the Salesman Problem(SP) using Branch-and-Bound algorithm in the C programming language in sequentially and in parallel using MPI.

Analyze the Salesman Problem(SP):

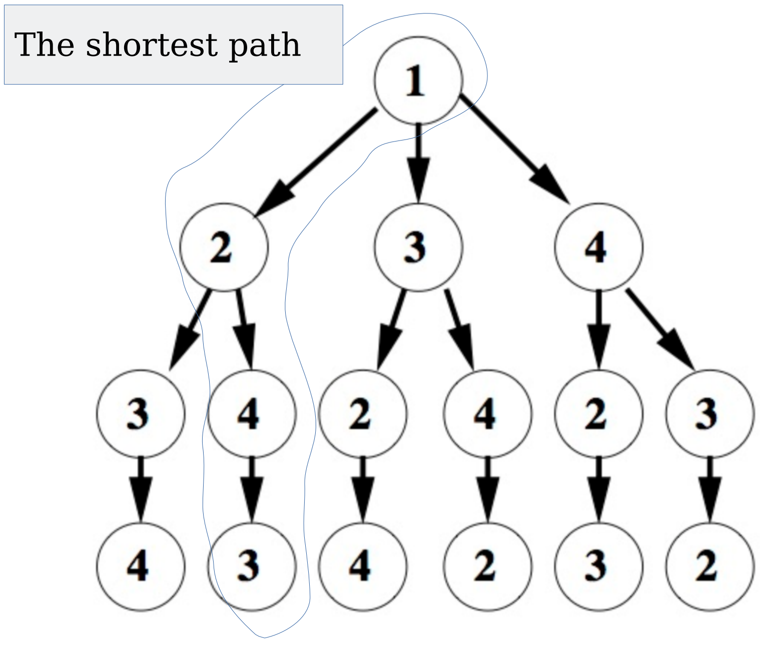
A SP is about finding the shortest route for a traveling salesman so that the salesman visits every city exactly once. Since for N cities there are at most N! possible routes. However, we are using branch and bound algorithm, when we have a distance greater than the bound, we can just prune the branch tree from our searching tree and therefore we do not need to explore the whole tree.

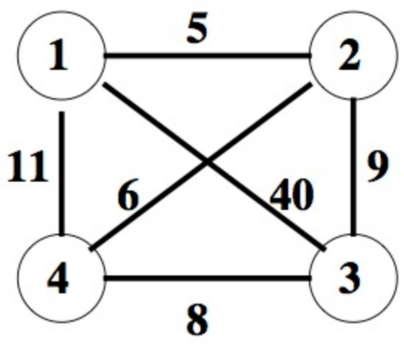
Design and Implementation(in Sequential):

1.) Intialize some variables that could be accessed for further use.

```
Int path;          // To store the shortest path
int** matrix;   // Store the distances between cities
int* order;      // Store the final order of the path
int* stack;      // Stack to store the current path we are looking
int* queue;     // A circular queue
int* alt;          // To store the next branch
```

2.) Read from our input file and stores the value of distances between cities in Matrix[][]
3.) Using Branch-and-Bound algorithm:
By using two data structures: *a stack and a circular queue to implement the tree structure*. Because the salesman always starts at the city 1 and then visits different cities in different orders, we can simply draws this behaviour to a tree diagram. First the salesman starts from the root, which is city 1, and goes down to level 2 to explore branch of the factorial trees. And then we keep counting the path as we go through and move on to the next level(Notice: in any path a city can only be visited only once). Again, we enumerate all possible vertices and go through all of it to find the shortest path.

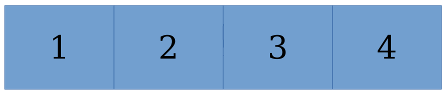




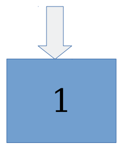
More detail about the implementation of branch and bound algorithm:
I will use the example above to show how my program works. As the diagram above it shows that we’ll look the route 1→2→3→4 at first:

: Head

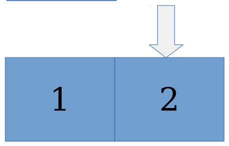
Queue:



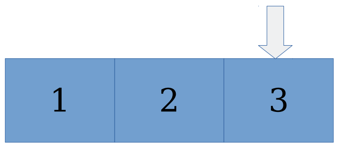
stack:



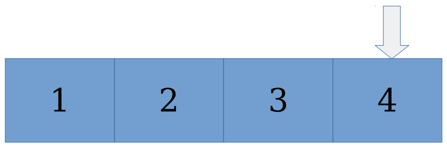
Path = Path(which is 0 now)



Path = Path + Matrix[0][1]
(the distance between city 1 to city 2)

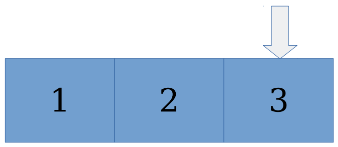


Path = Path + Matrix[1][2]
(the distance between city 2 to city 3)

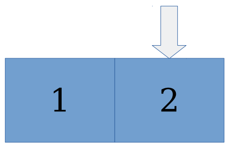


Path = Path + Matrix[2][3]
(the distance between city 3 to city 4)

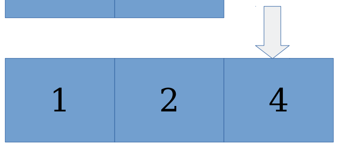
Once it reaches to the end, which is the leaf, the head will goes to the upper layer and see whether there is another branch.



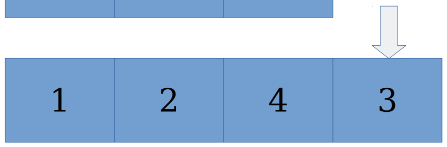
Path = Path - Matrix[2][3]
check branch: there is no branch



Path = Path - Matrix[1][2]
check branch: there is a branch with 4→3

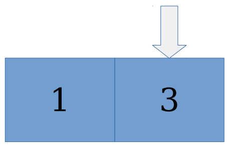


Path = Path + Matrix[1][3]
(the distance between city 2 to city 4)



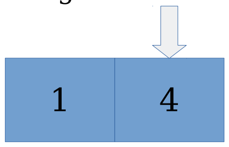
Path = Path + Matrix[3][2]
(the distance between city 4 to city 3)

To improve the performance, we also *check whether the cost of current path is bigger than the shortest path right now*. If the branch would take longer distance then we do not want to waste the time to explore all the paths inside that branch because the goal is finding the shortest one. For example, we have already explored the path 1→2→3→4 and 1→2→4→3. We know the shortest path so far is 1→2→4→3, which will cost 19 units of distance. As we goes back to the root(city 1) to explore another branch.



Path = Path + Matrix[0][2]
(the distance between city 1 to city 3)

We notice that the cost of traveling from city 1 to city 3 is bigger than the shortest path so far, so we stop looking this branch. Instead, we move to next branch.



Path = Path - Matrix[0][2]
Path = Path + Matrix[0][3]
(the distance between city 1 to city 4)

The whole behaviour is the same as pruning the tree so that the program does not need to explore all the trees.

Measurements of performance for the graph:

As we have already described the behaviour of B&B algorithm, we know the worst case complexity remains the same as Brute Force. That means we may have to go through (n-1)! to find out the shortest path where n stands for number of cities(included city 1).

I have already run through my program with different input sizes of cases and find out it works pretty well when the n is less than or equal to 12. But when the n is up to 15, it takes about 4 seconds to get the result. And when the n is 16, it takes more than 18 seconds to get the result. So we can imagine that it would take much longer to finish when the n is above 20. And that is why we are going to use parallelize our program to explore how multiprocessing is going to improve the performance.

Design and Implementation(in Parallel):

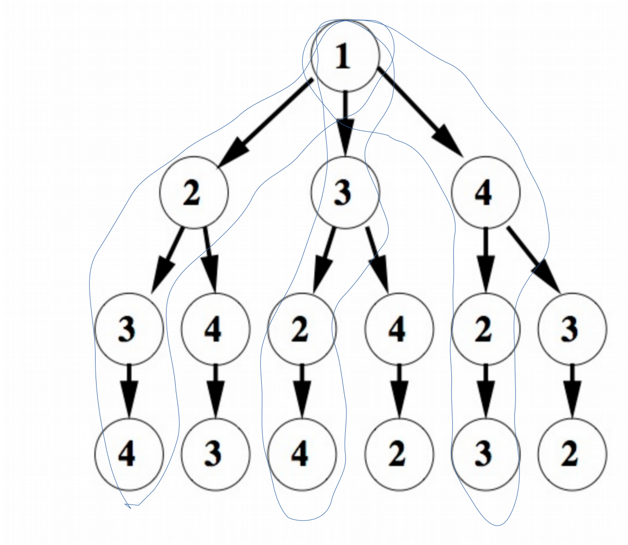
First, our *root process will read the input file and initialise some variables for our data structur*e, such as queue, stack, matrix, etc. And the root process will *broadcast all of these important variables to all other processes.* Before we go further, let’s split up the parallelization into four parts, decomposition, assignment, orchestration and mapping.

Decomposition:
First thing we have to do is breaking up the computation into tasks that could be executed in parallel. In this assignment, we are focusing on the *functional decomposition, which is breaking down the work and each task performs a portion of overall work.*

Assignment:
The goal is to balance workload for each process and reduce communication costs. There are two strategies, one is statically, and another is dynamically. I saw the additional explanation of Assignment 3 video, and she suggested us to use dynamically allocation for this assignment so that each process will have equal and balance workload in runtime, especially when the pruning happens. I did try this method, however, *the machine I am using only has dual cores. I notice it did not improve the performance but slow down a bit. The reason is that I kept my root process doing nothing but receiving messages and distributing the works, while another process doing all the works as well as synchronizing the messages*. And this is why *I use static allocation for this assignment.* By spliting out the queues equally to processes and synchronize the shortest path value before each process starts to explore a new branch tree. This will drop the performance in a few cases because it has to synchronize during the progress. Since we do not know what is the input data, it has a bigger chance to improve the performance as well for the pruning concerned especially when n is big enough. Because the layer2 has (n-1) subtrees, so we can use p(number of processes) processes work concurrently in different subtrees and go through all the subtress. To be more precise, *in layer2, we divide (n-1) by p and get how many subtrees for each process has to go through(Notice the process with the highest rank will have to go throuth the remainder branch if there is any).* And we have to assign each process with different subtrees so that there is no a subtree will be go through twice. In addition, we store the shortest path for each process in to it’s result buffer. The following diagram shows that how our program divide the queues to different processes.

When n is equal to 2:
Starting_Queue:

1	2	3	4
1	3	2	4
1	4	2	3



process 0
queue:

1	2	3	4

process 1
queue:

1	3	2	4
1	4	2	3

Orchestration:

Even we have already done the decomposition and assignment, we still have a little concerned about whether the processes need to communicate and synchronize the shortest path value during the runtime. Let's say, if we do not allow the synchronization during the runtime, we only gather the result when all the processes finish exploring routes, what if there is one process, which does not prun any branch tree, would take much longer time compared to other processes. In order to prevent this, we need a mechanism that allows interprocess communication and synchronization. In fact, I have already discussed this earlier that *"processes synchronize the shortest path value before each process starts to explore a new branch tree"*. The mechanism is simple, we *gather the result buffer from all processes to the root process, and then we compare the paths we have so far and find out what is the best route*. After that, we *broadcast the best route to all other processes before they explore another new branch.* The reason why I do not synchronize the result buffer every time when a process reaches a better shorter path is that, it may have race condition. What if a process_i just gets a better result but at the same time process_j is also getting a shorter path, and both of them sends the result buffer at the same time. It is hard to distinguish which one is the shortest one. Moreover, we want to reduce the communication as well. That is why we only synchronize the path buffer before each process is about to explore a new branch.

Mapping:

Mapping workers to hardware execution units to maximize locality, data sharing and minimize the costs of communication as well as synchronization. We do not do anything for this assignment, in fact, the operating system and mpicc compiler would do the job.

Example output of our logfile:

[Sample.txt.seq_log]

```
This path is pruned from the search: 1 2 3
This path is pruned from the search: 1 2
This path is pruned from the search: 1 3
This path is pruned from the search: 1 4 2


The Path is: 1 2 4 3
Distance: 19
```

I couldn't implement the log functionality for parallel one. I tried the MPI_File_Write(), but when I tried to write integer using MPI_INT type, I just kept getting some strange characters or symbols. I also tried to convert int to char and write the values, still no luck.

Analyze:

Local Machine(Dell E7250):
Specs:
i7-5600u dual cores @3.2GHz
(2 physical cores, 4 logical threads)
16 gb ram ddr-3l 1600mHz
Hard Drive: 512 Gb SSD(mSATA)

Measurements:
Right now I am just using <time.h> to analyze the sequential runtime and use MPI_Wtime() for counting the parallel program runtime. I start the timer when the program is about to run Branch and Bound codes, which means it excludes the reading file times and the time for setting up the environment.

Performance results:

Here is a table that shows the average of runtime for each sapmle file:

File Name	n	tsp_sequential	tsp_parallel	Shortest Path
sample.txt	4	0.000003	0.000005	1, 2, 4, 3
sample2.txt	5	0.000007	0.000009	1, 3, 4, 2, 5
sample3.txt	10	0.005951	0.002222	1, 2, 4, 9, 8, 10, 6, 3, 5, 7
sample4.txt	12	0.044443	0.027927	1, 3, 10, 5, 9, 4, 8, 7, 2, 6, 12, 11
sample4-1.txt	12	0.029142	0.011353	1, 8, 4, 9, 10, 2, 5, 3, 11, 6, 12, 7
sample5.txt	14	0.704373	0.462282	1, 9, 3, 6, 13, 14, 5, 2, 10, 4, 11, 7, 12, 8
sample6.txt	16	18.574791	17.100323	1, 7, 11, 16, 4, 13, 2, 6, 10, 5, 12, 14, 15, 9, 8, 3
sample7.txt	17	111.079867	77.891459	1, 15, 7, 17, 6, 16, 10, 4, 9, 12, 2, 3, 13, 14, 5, 11, 8
sample7-1.txt	17	84.838004	66.846040	1, 12, 8, 15, 3, 9, 7, 16, 13, 6, 2, 4, 10, 17, 14, 11, 5
sample7-2.txt	17	25.671497	30.671986	1, 2, 7, 5, 14, 9, 16, 17, 3, 12, 4, 15, 13, 10, 8, 6, 11
sample8.txt	18	91.942477	84.685368	1, 13, 14, 17, 8, 16, 7, 2, 18, 6, 9, 11, 15, 4, 3, 12, 10, 5

All the sample files are randomly generated by the python script provided from the forum. Obviously when the n is less than 10, the sequential program could run slightly faster than parallel one. The reason is that when the n is small, the processes in parallel program have to wait at some points to synchronize the result buffer while the sequential program just run straight without interrupt. In addition, modern chip can run in very fast speed, while when n is less than 10, it only has at best (9-1)! possibilies, which is 40320 possibilies. It can be easily solved by modern cpu.

When the n is becoming larger, the required time increases dramatically. We can easily see the performance difference between sequential one and parallel one. The parallel one has noticable performance speedup when n is above 10. We will discuss more speedup detail later. Right now, I want to discuss why tsp_parallel didn't speedup in sample6.txt and sample7-2.txt. We can see the sample6.txt has 16 cities, and the shortest path is when the second node is 7. I have already discussed the mechanism of our "assignment" process earlier, and we know that in this case, we assign the process 0 to search all the branches with the second node from 2 to 8, and the process

1 is searching the branches with the second node from 9 to 16. So the shortest path is almost the last few nodes being searched from process 0. We can deduce that maybe there was not much pruning occuring before it reached to this point, and the synchronization between processes also took some time. For sample7-2, we can see the shortest path is when the second node is 2. Which means, the sequential program may just prune most of the branches after it gets to this point, and that is why it can be faster than the parallel program. Aside from these cases, we can still see the performance has quite high chance be improved by the parallel program.

According to Amdahl's law, the performance of speedup equation:

$$S_p(n)=\frac{T^x(n)}{T_p(n)}=\frac{1}{f+\frac{1-f}{p}}$$

where f is the ratio of sequential code.

	tsp_sequential	tsp_parallel	speedup
sample3.txt	0.005951	0.002222	2.68
sample4.txt	0.044443	0.027927	1.59
sample4-1.txt	0.029142	0.011353	2.57
sample5.txt	0.704373	0.462282	1.52
sample6.txt	18.574791	17.100323	1.10
sample7.txt	111.079867	77.891459	1.43
sample7-1.txt	84.838004	66.846040	1.27
sample7-2.txt	25.671497	30.671986	0.84
sample8.txt	91.942477	84.685368	1.09

It is not easy to get the correct speedup from above table. But I do think if we can run more test cases and gather the results, I think we could still expect the average speedup would be above 1.30 under dual processors.

Result:

Unfortuantely, I only have a dual core machine, I can't run more tests on more processors. From above speedup table we can get the expect speedup value of our program would be 1.3. And we can utilize Amdahl's law to compute the ratio of sequential code and get 0.54. Which means about half of the code is just sequential. Moreover, remember I have mentioned that for our program, we use static assignment, this may encounter an issue, what if one of the process prune all the branch trees and another process is still running. To solve this issue, we can use dynamic assignment with implementing threads in our processes, which will allows the root process do the B&B algorithm, synchronize the buffer and distribute works equally to different process at the same time. Even there are still something we can improve our program, the above table still shows that our parallel program can have great performance and good speedup in many cases.