LP1 Assignment HPC H4

Parallel Search Algorithm

Date - 24th September, 2020.

Assignment Number - HPC H4

Title

Parallel Search Algorithms

Problem Definition

Design and implement parallel algorithm utilizing all resources available for

- Binary Search for Sorted Array
- Best-First Search(traversal of graph to reach a target in the shortest possible path)

Learning Objectives

- Learn parallel decomposition of searching algorithms.
- Learn parallel computing using OpenMP and MPI

Learning Outcomes

I will be able to decompose searching algorithms into subproblems, to solve sub problems using threads in OpenMP and ranks in MPI

Software Packages and Hardware Apparatus Used

- Operating System : 64-bit Ubuntu 18.04
- Browser : Google Chrome
- Programming Language : C++ (OpenMP and MPI header file included), Python 3
- Jupyter Notebook Environment : Google Colaboratory

Programmers' Perspective

Let S be the system set:

S = {s; e;X; Y; Fme; Ff; DD; NDD; Fc}

s=start state

- Weighted Graph
- Sorted Array

e=end state

- Traversed Path in the Graph using Best First Search
- Index of Searched Element from Sorted Array

X=set of inputs

X = {X1, X2}

- where X1 = Weighted Graph (Adjacency Matrix)
    - Adjacency Matrix
    - Number of Nodes
- where X2 = Sorted Array

Y=set of outputs

Y = {Y1, Y2}

- Y1 = Traversed Path in Graph
- Y2 = Index of Searched Element from Sorted Array

Fme is the set of main functions

Fm = {fm1,fm2}

- fm1 = Main Display Function for Best First Search
- fm2 = Main Display Function for Binary Search

Ff is the set of friend functions

Ff = {f1,f2} where

- f1 = Parallel Best First Search

        - f2 = Parallel Binary Search

DD = Deterministic Data

- Weighted Graph
- Sorted Array

NDD = Non-deterministic data (Eg - Null Values in Dataset)

- No Non-Deterministic data detected

Fc = failure case

- Invalid Traversal Path in BFS
- Incorrect Search Index in Binary Search

Concepts related Theory

OpenMP

OpenMP is an implementation of multithreading, a method of parallelizing whereby a *primary* thread (a series of instructions executed consecutively) *forks* a specified number of *sub*-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function (called `omp_get_thread_num()`). The thread id is an integer, and the primary thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the primary thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `omp.h` in C/C++.

MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

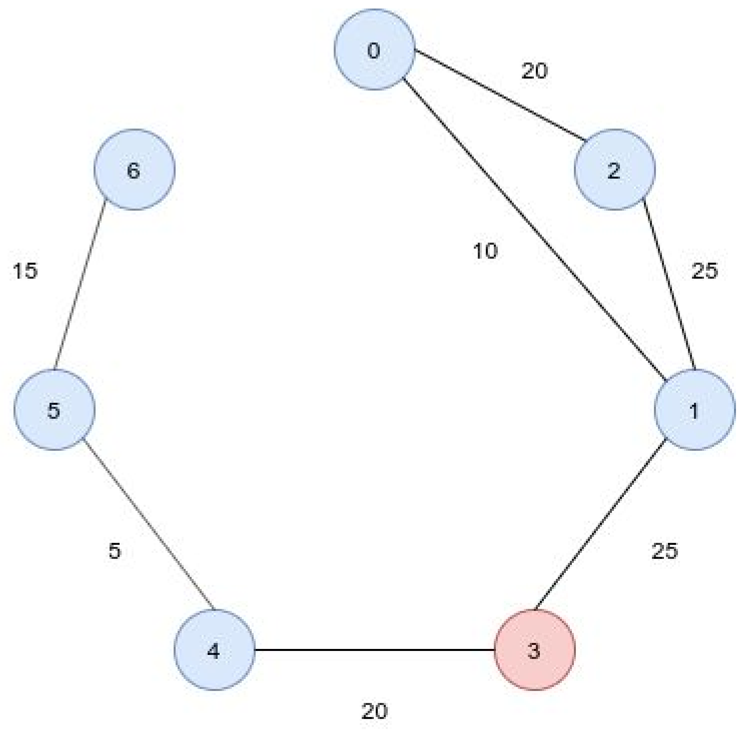
Best First Search

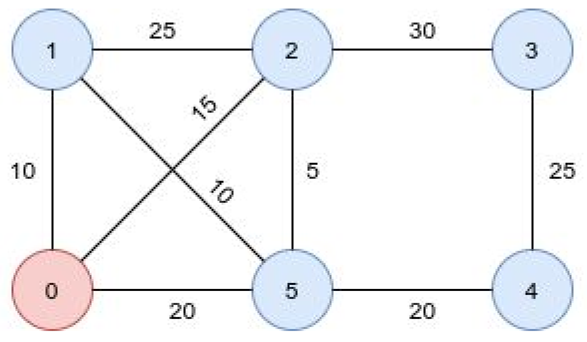
Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.
Best-first search works by estimating the promise of node n by a "heuristic evaluation function f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."
Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first. This specific type of search is called greedy best-first search or pure heuristic search.
Efficient selection of the current best candidate for extension is typically implemented using a priority queue.

Clique and Clique Problem

In the mathematical area of graph theory, a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. Cliques are one of the basic concepts of graph theory and are used in many other mathematical problems and constructions on graphs.
Cliques have also been studied in computer science: the task of finding whether there is a clique of a given size in a graph (the clique problem) is NP-complete, but despite this hardness result, many algorithms for finding cliques have been studied.
In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph. It has several different formulations depending on which cliques, and what information about the cliques, should be found.

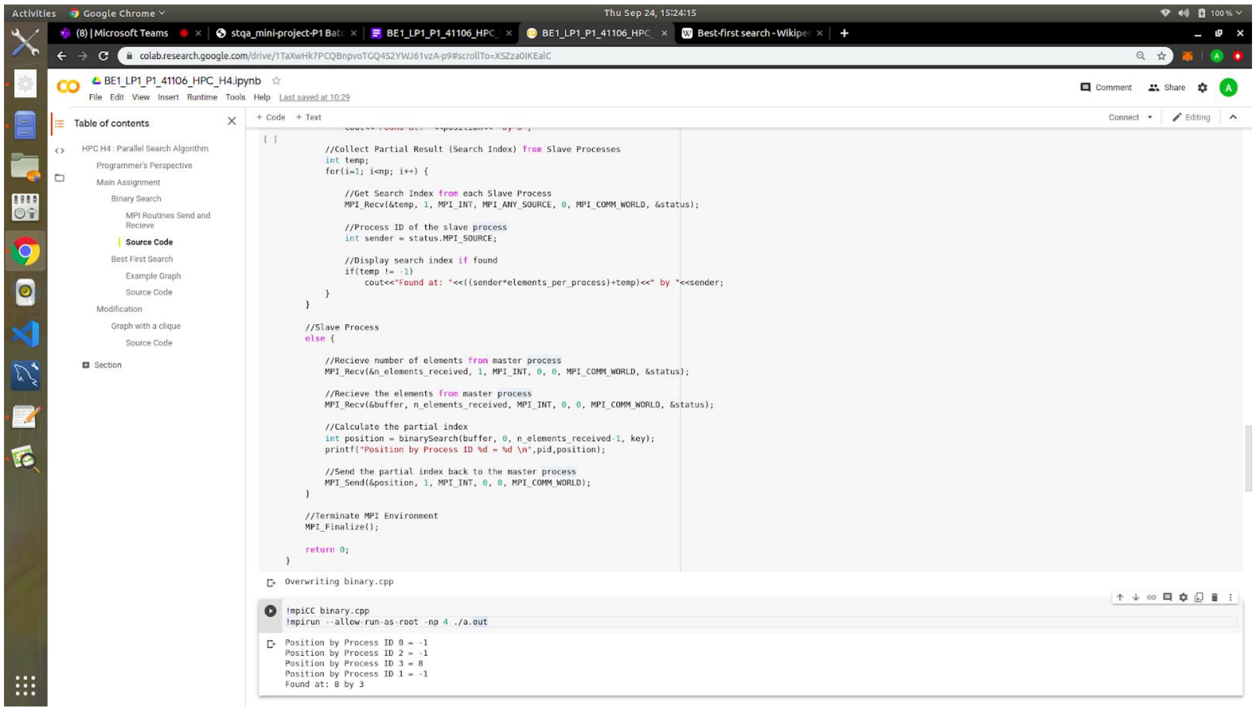
Diagram of graphs used in Best First Search





Output Screenshots

Binary Search



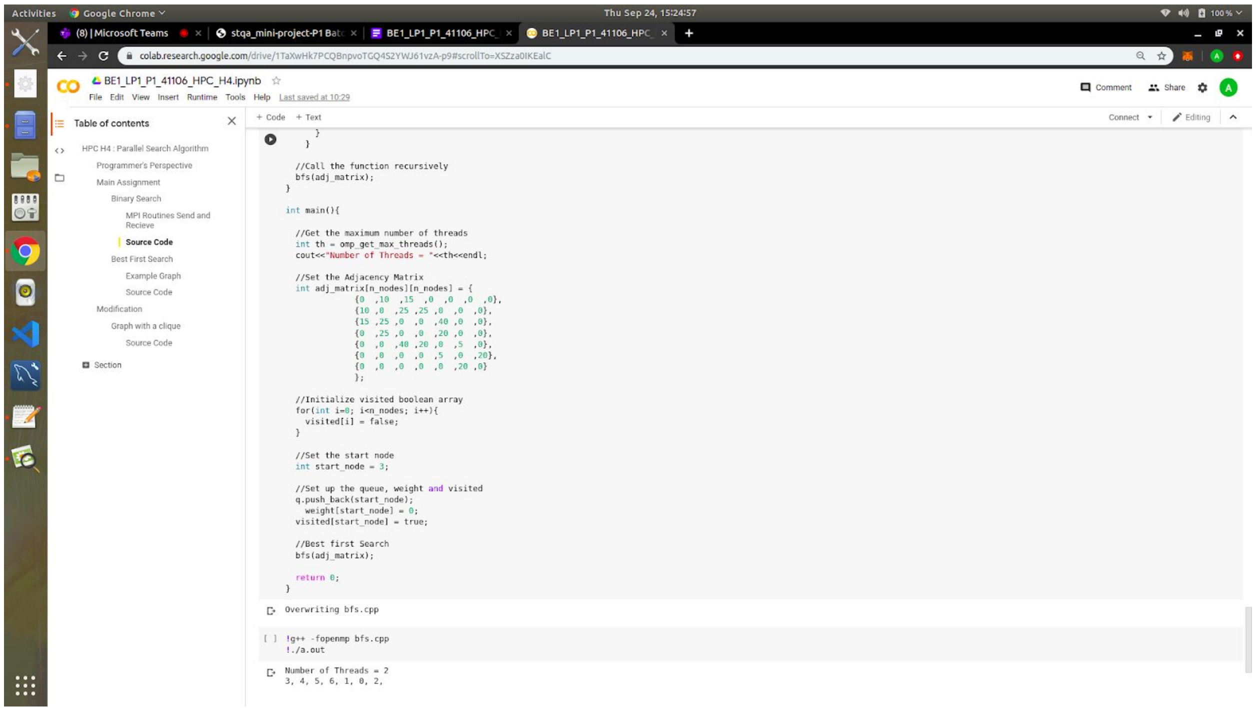
```
        //Collect Partial Result (Search Index) from Slave Processes
        int temp;
        for(i=1; i<np; i++) {

            //Get Search Index from each Slave Process
            MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);

            //Process ID of the slave process
            int sender = status.MPI_SOURCE;

            //Display search index if found
            if(temp != -1)
                cout<<"Found at: "<<((sender*elements_per_process)+temp)<<" by "<<sender;
        }
    }

    //Slave Process
    else {

        //Recieve number of elements from master process
        MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        //Recieve the elements from master process
        MPI_Recv(&buffer, n_elements_received, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        //Calculate the partial index
        int position = binarySearch(buffer, 0, n_elements_received-1, key);
        printf("Position by Process ID %d = %d \n",pid,position);

        //Send the partial index back to the master process
        MPI_Send(&position, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    //Terminate MPI Environment
    MPI_Finalize();

    return 0;
}
```

```
Overwriting binary.cpp
```

```
!mpiCC binary.cpp
!mpirun --allow-run-as-root -np 4 ./a.out
```

```
Position by Process ID 0 = -1
Position by Process ID 2 = -1
Position by Process ID 3 = 8
Position by Process ID 1 = -1
Found at: 8 by 3
```

Best First Search



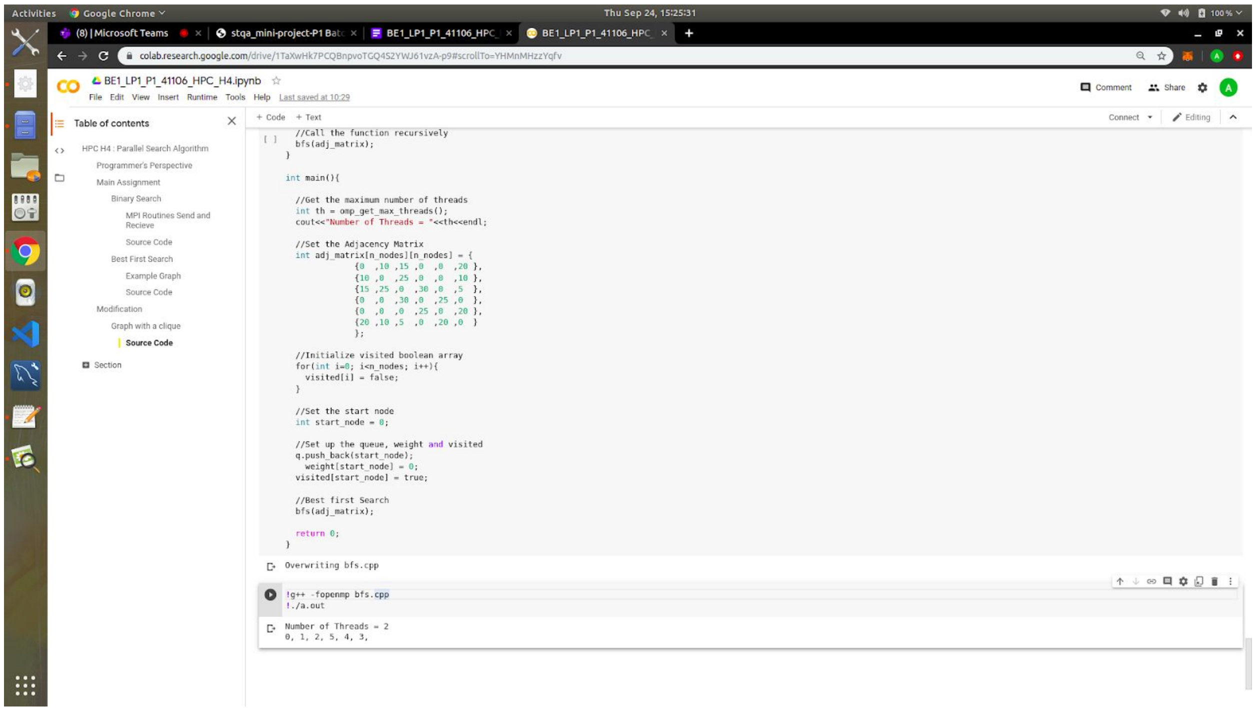
```
        }
      }

    //Call the function recursively
    bfs(adj_matrix);
  }

  int main(){

    //Get the maximum number of threads
    int th = omp_get_max_threads();
    cout<<"Number of Threads = "<<th<<endl;

    //Set the Adjacency Matrix
    int adj_matrix[n_nodes][n_nodes] = {
                {0  ,10  ,15  ,0  ,0  ,0  ,0},
                {10 ,0  ,25 ,25 ,0  ,0  ,0},
                {15 ,25 ,0  ,0  ,40 ,0  ,0},
                {0  ,25 ,0  ,0  ,20 ,0  ,0},
                {0  ,0  ,40 ,20 ,0  ,5  ,0},
                {0  ,0  ,0  ,0  ,5  ,0  ,20},
                {0  ,0  ,0  ,0  ,0  ,20 ,0}
                };

    //Initialize visited boolean array
    for(int i=0; i<n_nodes; i++){
      visited[i] = false;
    }

    //Set the start node
    int start_node = 3;

    //Set up the queue, weight and visited
    q.push_back(start_node);
      weight[start_node] = 0;
    visited[start_node] = true;

    //Best first Search
    bfs(adj_matrix);

    return 0;
  }
```

```
Overwriting bfs.cpp
```

```
!g++ -fopenmp bfs.cpp
!./a.out
```

```
Number of Threads = 2
3, 4, 5, 6, 1, 0, 2,
```

Best First Search - Use Graph with Clique (Modification)



```
    //Call the function recursively
    bfs(adj_matrix);
  }

  int main(){

    //Get the maximum number of threads
    int th = omp_get_max_threads();
    cout<<"Number of Threads = "<<th<<endl;

    //Set the Adjacency Matrix
    int adj_matrix[n_nodes][n_nodes] = {
                {0  ,10 ,15 ,0  ,0  ,20 },
                {10 ,0  ,25 ,0  ,0  ,10 },
                {15 ,25 ,0  ,30 ,0  ,5  },
                {0  ,0  ,30 ,0  ,25 ,0  },
                {0  ,0  ,0  ,25 ,0  ,20 },
                {20 ,10 ,5  ,0  ,20 ,0  }
                };

    //Initialize visited boolean array
    for(int i=0; i<n_nodes; i++){
      visited[i] = false;
    }

    //Set the start node
    int start_node = 0;

    //Set up the queue, weight and visited
    q.push_back(start_node);
      weight[start_node] = 0;
    visited[start_node] = true;

    //Best first Search
    bfs(adj_matrix);

    return 0;
  }
```

```
Overwriting bfs.cpp
```

```
!g++ -fopenmp bfs.cpp
!./a.out
```

```
Number of Threads = 2
0, 1, 2, 5, 4, 3,
```

Source Code

Binary Search using MPI

```
%%writefile binary.cpp
#include<mpi.h>
#include<iostream>
using namespace std;

int n = 12;
int a[] = {1,2,3,4,7,9,13,24,55,56,67,88};
int key = 55;

//Temporary Array for Slave Process
int buffer[20];

int binarySearch(int *array, int start, int end, int value) {
   int mid;
   while(start <= end) {
       mid = start + (end-start)/2;
       if(array[mid] == value)
           return mid;
       else if(array[mid] > value)
           end = mid - 1;
       else
           start = mid + 1;
   }
   return -1;
}

int main(int argc, char* argv[]) {

   int pid, np, elements_per_process, n_elements_received;

   MPI_Status status;
```

```
    //Initialize MPI Environment
    MPI_Init(&argc, &argv);

    //To get rank of a process
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);

    //To get number of processes which are communicating
    //MPI_COMM_WORLD is the default communicator
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    //Master Process
    if(pid == 0) {
        int index, i;

        //Check if more than one process is running
        if(np > 1) {
            for(i=1; i<np-1; i++) {

                index = i * elements_per_process;

                //Send the number of elements to the slave process
                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);

                //Send the actual element to the slave process
                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0,
MPI_COMM_WORLD);

            }

            //For the last process

            index = i* elements_per_process;
            int elements_left = n - index;

            //Send the number of elements to the slave process
            MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            //Send the actual element to the slave process
            MPI_Send(&a[index], elements_left, MPI_INT, i, 0,
MPI_COMM_WORLD);
        }

        //Master itself performs binary search
        int position = binarySearch(a, 0, elements_per_process-1, key);
        printf("Position by Process ID %d = %d \n",pid,position);
        if(position != -1)
            cout<<"Found at: "<<position<<" by 0";

        //Collect Partial Result (Search Index) from Slave Processes
        int temp;
        for(i=1; i<np; i++) {

            //Get Search Index from each Slave Process
            MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
&status);

            //Process ID of the slave process
            int sender = status.MPI_SOURCE;

            //Display search index if found
            if(temp != -1)
                cout<<"Found at: "<<((sender*elements_per_process)+temp)<<"
by "<<sender;
        }
    }

    //Slave Process
    else {

        //Recieve number of elements from master process
        MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
```

```
        //Recieve the elements from master process
        MPI_Recv(&buffer, n_elements_received, MPI_INT, 0, 0,
MPI_COMM_WORLD, &status);

        //Calculate the partial index
        int position = binarySearch(buffer, 0, n_elements_received-1, key);
        printf("Position by Process ID %d = %d \n",pid,position);

        //Send the partial index back to the master process
        MPI_Send(&position, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    //Terminate MPI Environment
    MPI_Finalize();

    return 0;
}
```

Best First Search using OpenMP

```
%%writefile bfs.cpp
#include<bits/stdc++.h>
#include "omp.h"

#define n_nodes 7

using namespace std;
list<int>q;
vector<int>weight(n_nodes,1000);
bool visited[n_nodes];

struct Comparator {
    // Compare 2 Edges objects using weight
    bool operator ()(const int &e1, const int &e2){
        return weight[e1]<weight[e2];
    }
};
```

```
//Display the list/priority queue - Debugging Function
void showlist(list <int> g) {
   list <int> :: iterator it;
   for(it = g.begin(); it != g.end(); ++it)
       cout<<*it<<" ";
   cout<<endl;
}

//Display the vector/weights - Debugging Function
void showvec(vector <int> g) {
   vector <int> :: iterator it;
   for(it = g.begin(); it != g.end(); ++it)
       cout<<*it<<" ";
   cout<<endl;
}

void bfs(int adj_matrix[n_nodes][n_nodes])
{
   if(q.empty())
     return;
   q.sort(Comparator());

   //pop first element and display it
   int cur_node = q.front();
   q.pop_front();
   printf("%d, ", cur_node);
    //For every element in the row of the adjacency matrix
   #pragma omp parallel for shared(visited,q,weight)
   for(int i=0; i<n_nodes; i++)
   {
     //If an unvisited Edge exists
     if(adj_matrix[cur_node][i]>0 && visited[i]==false)
     {

       //Replace the weight if it is larger
       if(weight[i] > adj_matrix[cur_node][i]){
            weight[i] = adj_matrix[cur_node][i];
        }

        //Push the destination of the smallest edge onto the queue
        q.push_back(i);
        visited[i]=true;
      }
    }
  //Call the function recursively
 bfs(adj_matrix);
}

int main(){
  //Get the maximum number of threads
 int th = omp_get_max_threads();
 cout<<"Number of Threads = "<<th<<endl;

 //Set the Adjacency Matrix
 int adj_matrix[n_nodes][n_nodes] = {
             {0  ,10  ,15  ,0  ,0  ,0  ,0},
             {10 ,0  ,25 ,25 ,0  ,0  ,0},
             {15 ,25 ,0  ,0  ,40 ,0  ,0},
             {0  ,25 ,0  ,0  ,20 ,0  ,0},
             {0  ,0  ,40 ,20 ,0  ,5  ,0},
             {0  ,0  ,0  ,0  ,5  ,0  ,20},
             {0  ,0  ,0  ,0  ,0  ,20 ,0}
             };
  //Initialize visited boolean array
 for(int i=0; i<n_nodes; i++){
   visited[i] = false;
 }

 //Set the start node
 int start_node = 3;

 //Set up the queue, weight and visited
 q.push_back(start_node);
```

```
   weight[start_node] = 0;
 visited[start_node] = true;
  //Best first Search
 bfs(adj_matrix);

 return 0;
}
```

Google Colab Notebook Link

https://colab.research.google.com/drive/1TaXwHk7PCQBnpvoTGQ4S2YWJ61vzA-p9?usp=sharing

Conclusion

I have successfully parallelized searching algorithms like binary search and best first search using OpenMP and MPI in C++.