Assignment 4
1) Binary Search

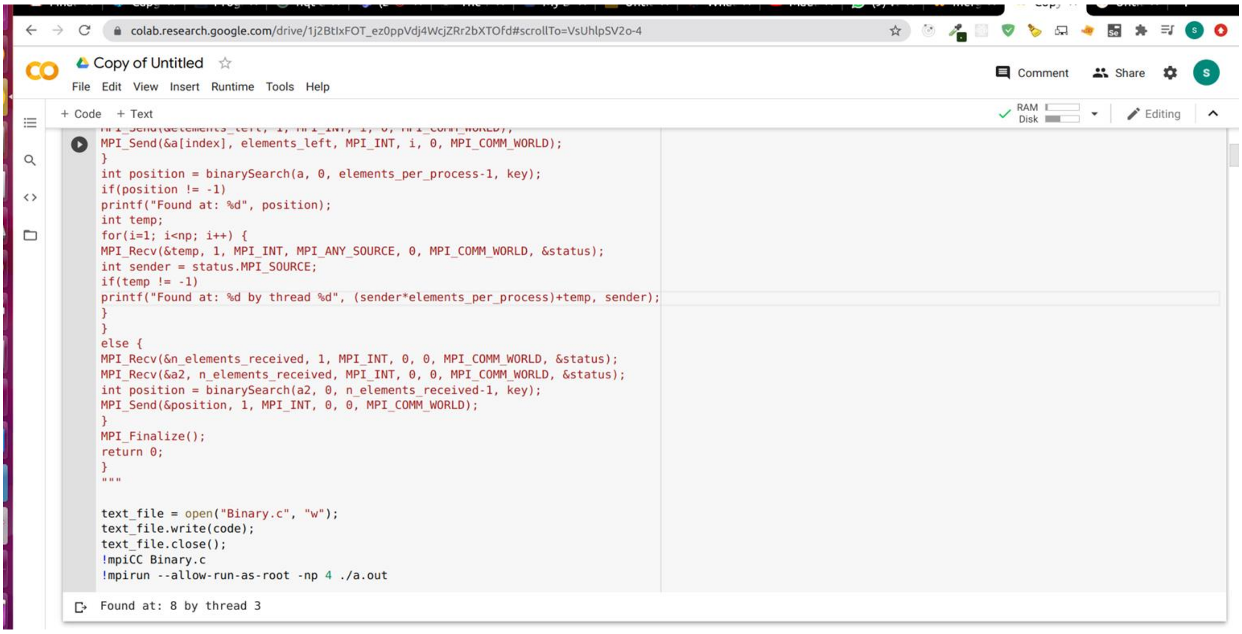
CODE:

```
code = """
#include<mpi.h>
#include<stdio.h>

#define n 12

#define key 55

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

int a2[20];

int binarySearch(int *array, int start, int end, int value) {
int mid;
while(start <= end) {
mid = (start + end) / 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;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);
if(pid == 0) {
int index, i;
if(np > 1) {
for(i=1; i<np-1; i++) {
index = i * elements_per_process;
//element count
MPI_Send(&elements_per_process, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0, MPI_COMM_WORLD);
}
index = i* elements_per_process;
int elements_left = n - index;
MPI_Send(&elements_left, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
MPI_Send(&a[index], elements_left, MPI_INT, i, 0, MPI_COMM_WORLD);
}
int position = binarySearch(a, 0, elements_per_process-1, key);
if(position != -1)
printf("Found at: %d", position);
int temp;
for(i=1; i<np; i++) {
MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
int sender = status.MPI_SOURCE;
if(temp != -1)
printf("Found at: %d by %d", (sender*elements_per_process)+temp, sender);
}
```

```
}
else {
MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&a2, n_elements_received, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
int position = binarySearch(a2, 0, n_elements_received-1, key);
MPI_Send(&position, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}
"""

text_file = open("Binary.c", "w");
text_file.write(code);
text_file.close();
!mpiCC Binary.c
!mpirun --allow-run-as-root -np 4 ./a.out
```

OUTPUT:

KEY TO FIND = 55

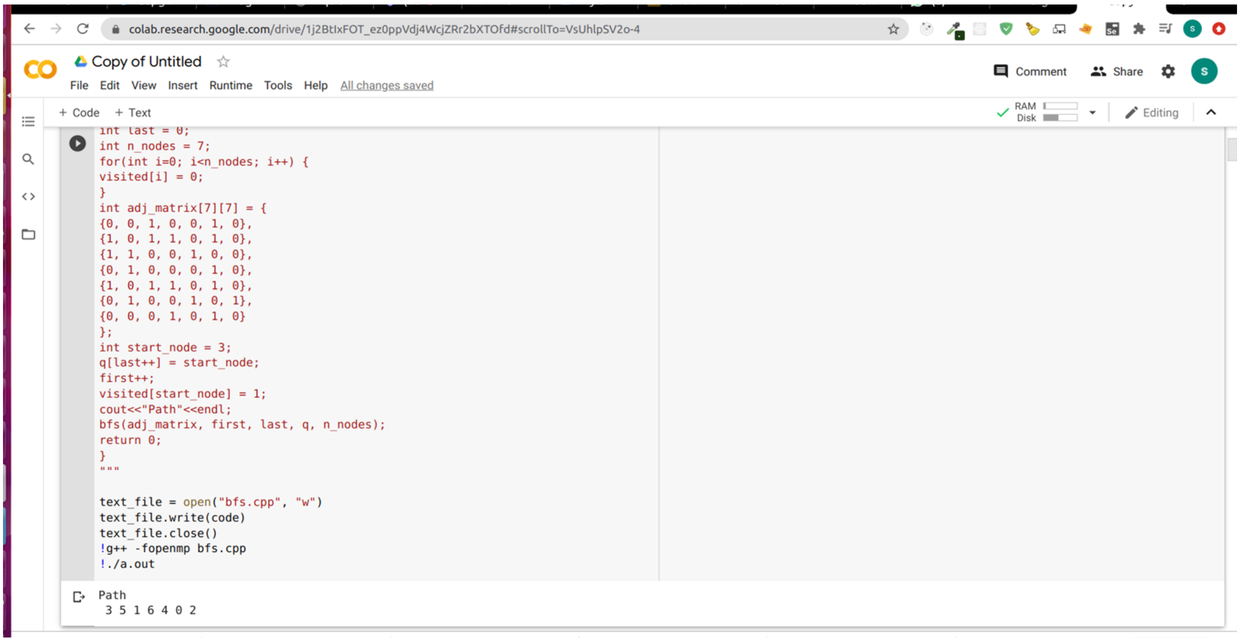


2)Best-First Search

CODE:

```
code = """
#include<iostream>
#include<omp.h>

using namespace std;
int q[100];
int visited[7];
int local_q;

void bfs(int adj_matrix[7][7], int first, int last, int q[], int n_nodes) {
if(first==last)
return;
int cur_node = q[first++];
cout<<" "<<cur_node;
omp_set_num_threads(3);
#pragma omp parallel for shared(visited)
for(int i=0; i<n_nodes; i++) {
if(adj_matrix[cur_node][i] == 1 && visited[i] == 0){
q[last++] = i;
visited[i] = 1;
}
}
bfs(adj_matrix, first, last, q, n_nodes);
}

int main() {
int first = -1;
int last = 0;
int n_nodes = 7;
for(int i=0; i<n_nodes; i++) {
visited[i] = 0;
}
int adj_matrix[7][7] = {
{0, 0, 1, 0, 0, 1, 0},
{1, 0, 1, 1, 0, 1, 0},
{1, 1, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 1, 0},
{1, 0, 1, 1, 0, 1, 0},
{0, 1, 0, 0, 1, 0, 1},
{0, 0, 0, 1, 0, 1, 0}
};
int start_node = 3;
q[last++] = start_node;
first++;
visited[start_node] = 1;
bfs(adj_matrix, first, last, q, n_nodes);
return 0;
}
"""

text_file = open("bfs.cpp", "w")
text_file.write(code)
text_file.close()
!g++ -fopenmp bfs.cpp
!./a.out
```

OUTPUT:



```
int last = 0;
int n_nodes = 7;
for(int i=0; i<n_nodes; i++) {
visited[i] = 0;
}
int adj_matrix[7][7] = {
{0, 0, 1, 0, 0, 1, 0},
{1, 0, 1, 1, 0, 1, 0},
{1, 1, 0, 0, 1, 0, 0},
{0, 1, 0, 0, 0, 1, 0},
{1, 0, 1, 1, 0, 1, 0},
{0, 1, 0, 0, 1, 0, 1},
{0, 0, 0, 1, 0, 1, 0}
};
int start_node = 3;
q[last++] = start_node;
first++;
visited[start_node] = 1;
cout<<"Path"<<endl;
bfs(adj_matrix, first, last, q, n_nodes);
return 0;
}
"""

text_file = open("bfs.cpp", "w")
text_file.write(code)
text_file.close()
!g++ -fopenmp bfs.cpp
!./a.out
```

```
Path
 3 5 1 6 4 0 2
```

LPI

High Performance Computing

Assignment 4

Date of Completion :- 11.9.2020 Roll No :- 41258

Title :- Parallel Search Algorithm

Problem Statement :- Design & implement parallel algorithm utilizing all resources available for ~~one~~
1) Binary search for sorted Array
2) Best - first search that (traversal of graph to reach a target in the shortest possible path)

Objectives :- To learn parallel implementation of searching algorithms
To learn about MPI.

Outcomes :- Students will be able to implement parallel searching techniques.
learn about MPI.

Software / Hardware Requirements :- Ubuntu OS, editor, Open MPI.

Theory :-

Binary search :-

1. Also known as logarithmic search is an algorithm that finds the position of the target value with a sorted array.
2. worst case → logarithmic time O(logn) where n is size of array.

3. Breadth First Search

1. Most common graph traversal algorithm.
2. Starts traversing from the source and leaves, the graph lengthwise thus exploring the neighbor nodes first.

Open MPI :-

1. It is a message ~~being~~ passing interface library which provides extremely high and competitive performance.
2. The OPEN MPI has 3 major ~~schedules~~ modules :-
a) OMPI = MPI node
b) ORTE = Open Runtime Environment
3) OPAL = Open Portable Access layer.
mpi cc compiler is used to compile C/C++ codes.

Algorithm

Parallel Binary Search :-
(Sorted array)

1) Divide the array into M blocks of size N/M
2) Apply ~~one~~ step1 of comparison to the middle element of each block
3) If found return index & terminate.
4) Otherwise identify the adjacent block and form a new block starting from the element following the one the signalled (>) and ending at the element preceeding the one that signalled (<).
5) If they are same element, return index.
6) Otherwise parallel binary search (new block)

Breadth first Search

Graph root G, source S.

1) enque (s)
2) Mark S as visited.

3. While (Q is not empty)
   //reverse the vector from Q
   // whose neighbor
   will be visited now
   1) v = deque (Q)   //processing all the neighbor of v
   2) w = neighbor of v
      if (w is not visited)
         enque (w)
      endif
   4) end while

Test Cases :-

for N = 12 Key 55
found at 8 by 3rd thread
Key 500
Not found

Bfs
Path :-
3 5 1 6 4 8 2

Conclusion :- Thus I completed the implementation of binary search and BFS using parallel reduction (MPI)