Assignment A4.

Title - Parallel Searching Algorithm

Problem statement -
Design & implement parallel algorithm utilizing all available resources for
- Binary search for sorted array
- Depth first search or Breadth first search or Best first search

Objective -
To study & learn about parallel implementation of Searching algorithms.
To learn about MPI API in C/C++.

Outcomes - We will be able to
learn about parallel Searching techniques.
learn about MPI.

Theory -

A) Binary Search -
- Binary Search also known as logarithmic search is an algorithm that finds the position of the target value within a sorted array.
- It compares the target value with the middle element of an array. If they are not equal, the half in which target element cannot lie is eliminated and the search continues on the remaining half.
- If the search ends with remaining half being empty, the target is not in the array.

- Binary Search runs in logarithmic time in the worst case, making O(log n) comparisons where n = size of array.

B) Breadth first Search -
- BFS is the most common graph traversal algorithm.
- It starts travelling from the source and traverse the graph layerwise, thus exploring the neighbor nodes first.
- In sequential implementation, a queue is maintained of the neighbor nodes in each layer.

- OpenMPI -
- It is a message passing Interface library which provides extremely high & competitive performance.
- The OpenMPI code has 3 major code modules -
  1. OMPI - MPI code
  2. ORTE - Open Runtime Environment
  3. OPAL - Open Portable Access Layer.
- mpicc compiler is used to compile the c/c++ codes embedded with OpenMPI.

Algorithms -

A) Parellel Binary Search :-
Parellel binary search (sorted array)
1. Divide the array into M blocks, of size n/M.
2. Apply one step of comparison to the middle element of each block.
3. If equality obtained, return address & terminate.
4. Otherwise, identify the adjacent blocks & form a new block starting from the element following the one that

signalled (>) & ending at the element proceeding the one that signalled (<)

5. If they are same element, return index.

6. otherwise, parallel_binary_search (new_block)

B) Breadth first search

BFS (Graph root G, Source s)

1. enque (s)
2. Mark s as visited.
3. while (q is not empty)

// remove the vertex from q where neighbors will be visited now

3.1 v = deque (q)

// processing all the neighbors of v.

// w = neighbors of v ∀ neighbors

3.2 if (w is not visited)

3.2.1 enque (w)

3.3 end if.

4. end while.

Conclusion -

Thus, we successfully implemented parallel binary search & breadth first search using open MPI.

Test cases & analysis

Searching	Input size	Sequential time	Parallel time	Efficiency
Binary Search (key = 54)	n = 1024	1.153	1.542377	0.7477
	n = 2048	1.673	1.236108	1.353
	n = 4096	1.075	0.933585	1.150
Depth first Search traversal	n = 1024	0.011	0.007	1.57
	n = 2048	0.05	0.019	2.63
	n = 4096	0.109	0.026	4.19

$$\text{Efficiency} = \frac{WCSA}{WCPA}$$

We observe that for increasing values of n, the parallel algorithm performs better than sequential algorithm.

the above calculation are done with keeping the number of processors constant & equal to 5.

Input :- key to be searched = 54.

Output :- Original size = 4096
New array size = 3576
Block size per processor = 3576 / 5 = 715
key found at index = 15 by processor No. 0.

Binary Search

In [6]:

```
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_WORL
D);

                MPI_Send(&a[index], elements_per_process, MPI_INT, i, 0, MPI_COM
M_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, &stat
us);
            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, &statu
s);

        MPI_Recv(&a2, n_elements_received, MPI_INT, 0, 0, MPI_COMM_WORLD, &statu
s);

        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;
}
"""
```

In [7]:

```
text_file = open("mpiBinary.c", "w");
text_file.write(code);
text_file.close();
```

In [8]:

```
!mpiCC mpiBinary.c
```

In [9]:

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

```
Found at: 8 by 3
```

Breadth First Search

In [10]:

```
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, 1, 1, 0, 0, 0, 0},
      {1, 0, 1, 1, 0, 0, 0},
      {1, 1, 0, 0, 1, 0, 0},
      {0, 1, 0, 0, 1, 0, 0},
      {0, 0, 1, 1, 0, 1, 0},
      {0, 0, 0, 0, 1, 0, 1},
      {0, 0, 0, 0, 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;

}
"""
```

In [11]:

```
text_file = open("code.cpp", "w")
text_file.write(code)
text_file.close()
```

In [12]:

```
!g++ -fopenmp code.cpp
```

In [13]:

```
!./a.out
```

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

In [ ]: