Project Report

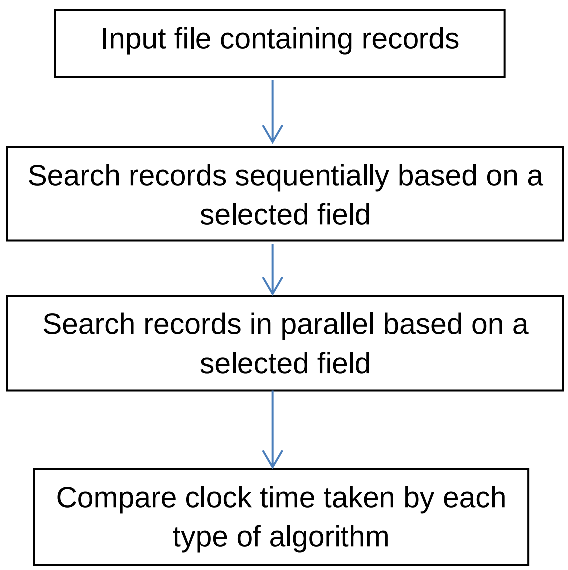
Project Title	Time comparison of sequential vs. parallel binary search
Course code	CS3006
Course title	Parallel Distributed and Computing
Department	Computer Science
Section	C
Language used	C++/shell script
Group Members	▪ Sara Jamal (19K-0207) ▪ Naz Panjwani (19K-1256)

Project report

❖ Problem

Binary Search is a divide-and-conquer searching algorithm which searches for an input value in a sorted array. The average-case time complexity for Binary Search is O(logn). Considering that it is a divide-and-conquer search where the input array is repeatedly divided into equal halves, can the search operation be made even faster using MPI parallelization?

❖ System Diagram



❖ Design

Typically, a serial approach to non-recursive Binary Search algorithm goes as follows:

```
while(first<=last){
        int middle = first + (last - first) / 2;

        // Check if search value is present at middle
        if (randomNums[middle] == searchVal)
            return middle;

        // If search value greater, ignore left half
        if (randomNums[middle] < searchVal) {
            first = middle + 1;
        }

        // If search value is smaller, ignore right half
        else
            last = middle - 1;
    }
```

❖ Source Data Description

Apps to be sorted by their respective app IDs are placed in ascending order into an input file which will be used by the operation code.
The input file consists of around 11,000 records of Google play store app records, from which the data is read by the program and inserted into a struct vector. Sequential and binary searches are then applied on the vector and their respective execution times compared. The number of processors to be used in parallel execution of Binary Search is to be specified by the user.

❖ Parallel Region Pseudocode

Each MPI process gets a chunk of file each containing n/k records where n is the total number of records in the file and k is the number of processors. Each process then performs binary search on its allocated chunk of file and returns the result from file if the given target value is found by a process along with the rank of the process.

```
if (rank == 0)
{
    ofstream temp("temp.txt");
    temp << 0;
    temp.close();
    cout << "Enter app ID to search: ";
    cin >> key;
    for (int i = 0; i < size1; i++)
        MPI_Send(&key, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}
else
    MPI_Recv(&key, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
begin = MPI_Wtime();
int start = rank * blockSize, found;
int end;
if (rank == size1 - 1) // last proc
    end = (rank + 1) * blockSize;
else
    end = (rank + 1) * blockSize - 1;

cout << " I AM PROCESS " << rank << " SEARCHING FROM " << start << " TO " << end << " LOOKING FOR " << key << endl;
while (start <= end && found == 0)
{
    ifstream temp("temp.txt");
    temp >> found;
    if (found == 1)
        break;
    temp.close();
    int mid = (start + end) / 2;
    cout << "\nRank " << rank << ": id = " << stoi(apps[mid].appID) << endl;
```

```cpp
        if (stoi(apps[mid].appID) == key){
            cout << "Element found by processor " << rank << " .\n";
            cout << apps[mid].appID << " " << apps[mid].appName << " " << apps[mid].rating << apps[mid].reviews << " "

            ofstream temp("temp.txt");
            temp << 1;
            temp.close();
            break;
        }
        else if (stoi(apps[mid].appID) < key)
        {
            start = mid + 1;
        }
        else
        {
            end = mid - 1;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD); /* IMPORTANT */
    stop = MPI_Wtime();
    if (rank == 0){ /* use time on master node */
        ifstream temp("temp.txt");
        temp >> found;
        if (found == 0)
            cout << "\nCouldn't find target value!";
        temp.close();
        printf("\nRuntime = %f\n", stop - begin);
    }
    ifstream temp("temp.txt");
    temp >> found;
    temp.close();
```

❖ Snapshot of Input Data (on each Thread)

Process rank along with chunk of vector allocated to them to find app id no. 8888.

```
I AM PROCESS 3 SEARCHING FROM 8130 TO 10840 LOOKING FOR 8888
I AM PROCESS 1 SEARCHING FROM 2710 TO 5419 LOOKING FOR 8888
I AM PROCESS 2 SEARCHING FROM 5420 TO 8129 LOOKING FOR 8888
I AM PROCESS 0 SEARCHING FROM 0 TO 2709 LOOKING FOR 8888
```

❖ Snapshot of Output Data (on each Thread)

Rank of process along with the index of vector searched by the process.

```
Rank 0: id = 2372
Rank 2: id = 6775
Rank 2: id = 7453
Rank 2: id = 7792
Rank 2: id = 7961
Rank 2: id = 8046
Rank 0: id = 2541
Rank 0: id = 2626
Rank 0: id = 2668
Rank 0: id = 2689
Rank 0: id = 2700
Rank 1: id = 4065
Rank 1: id = 4743
Rank 2: id = 8088
Rank 2: id = 8109
Rank 2: id = 8120
Rank 2: id = 8125
```

```
Rank 2: id = 8128
Rank 2: id = 8129
Rank 2: id = 8130
Rank 0: id = 2705
Rank 0: id = 2708
Rank 1: id = 5082
Rank 1: id = 5251
Rank 1: id = 5336
Rank 1: id = 5378
Rank 1: id = 5399
Rank 0: id = 2709
Rank 0: id = 2710
Rank 1: id = 5410
Rank 1: id = 5415
Rank 1: id = 5418
Rank 1: id = 5419
Rank 1: id = 5420
```

```
Rank 3: id = 9486
Rank 3: id = 8808
Rank 3: id = 9147
Rank 3: id = 8977
Rank 3: id = 8892
Rank 3: id = 8850
Rank 3: id = 8871
Rank 3: id = 8881
Rank 3: id = 8886
Rank 3: id = 8889
Rank 3: id = 8887
Rank 3: id = 8888
```

```
Element found by processor 3 .
8888 The Secret Daily Teachings 4.3206 32M 1,000+ Paid $4.99  Everyone Lifestyl  11-Feb-18
```

Rank 0 had to perform 10 iterations,

Rank 1 had to perform 12 iterations,

Rank 2 had to perform 12 iterations, and

Rank 3 had to perform 12 iterations

before app ID 8888 was found.

❖ Output for finding app ID 8888 on 4 processors:

Sequential:

```
k190207@k190207:~$ ./proj.sh
Enter 1 for sequential binary search, 2 for parallel binary search: 1

���r Choose a relevant option �����

 1.Sequential binary search using App ID

 2.Sequential binary search using App Name


Enter your choice: 1
Enter app ID to search: 8888

8888 The Secret Daily Teachings 4.3206 32M 1,000+ Paid $4.99  Everyone Lifestyle 11-Feb-1

Time measured: 0.000021535000000000000 seconds.

real    0m6.278s
user    0m0.088s
sys     0m0.000s
```

Parallel:

```
Rank 0: id = 2709

Rank 0: id = 2710
Runtime = 0.000069

real    0m3.638s
user    0m10.252s
sys     0m0.064s
```

❖ Conclusion

One crucial idea to note, when finding a parallel approach to a problem, is that the serial operation needs to be costly enough to compute resources to parallelize. Often times, complex calculations such as matrix multiplication or dot product are used to demonstrate parallel effectively, for a specific reason. That is, the operation needs to be costly enough to outweigh the performance repercussions of a parallel solution. Specific performance repercussions such as thread spawning, excessive function calls, voltage and thermal limitations across several threads, and cache hierarchy issues can all play a part in slowing down performance. In the case of Binary Search, the operation simply is not costly enough to make useful out of a parallel approach. Hence MPI is not an effective solution to all parallel problems.