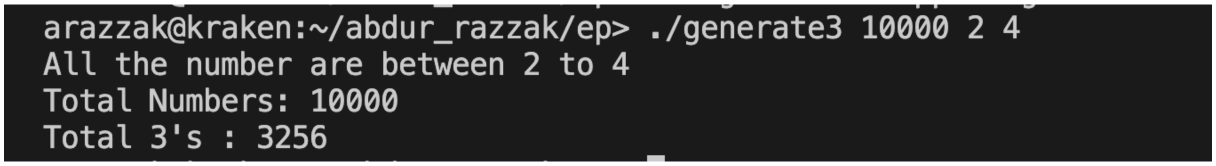
Evaluate Different Platforms

Abdur Razzak, Aggie ID: 800810091

Preparing the Input

1. For each size of input, one input file will be generated, and the exact file will be tested to 5 different algorithms to make them feel same completely in order to compare proper runtime.

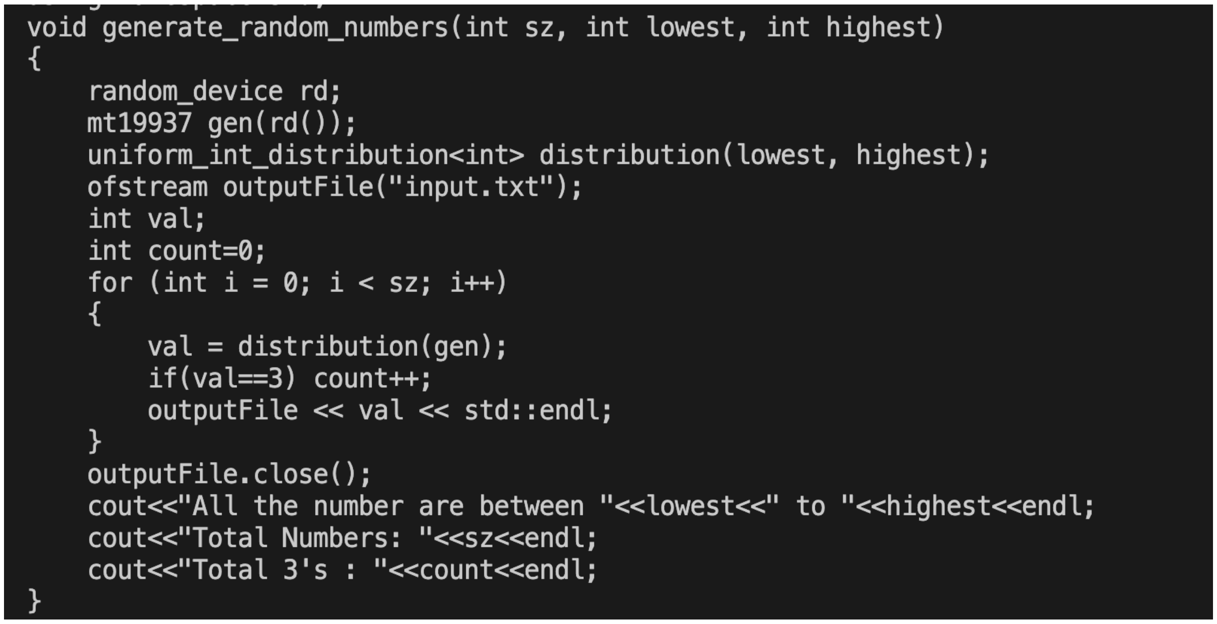
2. Therefore, before running every algorithm, generating3 file will be run once.



```
arazzak@kraken:~/abdur_razzak/ep> ./generate3 10000 2 4
All the number are between 2 to 4
Total Numbers: 10000
Total 3's : 3256
```

Here, the first argument is the size of array, and the next values are range of integers that the array will have. 2 is the lower bound and 4 is the upper bound. So, this array will have 10,000 integers and all of them are between 2 and 4. Also, the number of serial counts is taken to compare with parallel algorithms. Here, 3256 number of times the three is in the array.

3. Code: The below function is used to create the array and inserting into the file.



```
void generate_random_numbers(int sz, int lowest, int highest)
{
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(lowest, highest);
    ofstream outputFile("input.txt");
    int val;
    int count=0;
    for (int i = 0; i < sz; i++)
    {
        val = distribution(gen);
        if(val==3) count++;
        outputFile << val << std::endl;
    }
    outputFile.close();
    cout<<"All the number are between "<<lowest<<" to "<<highest<<endl;
    cout<<"Total Numbers: "<<sz<<endl;
    cout<<"Total 3's : "<<count<<endl;
}
```

4. MPI was every time run with 4 nodes connected in the host file.

Analysis of C-Threads

For Size = 10,000:   All the time are in milliseconds

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	4	6	5	5	5	5	0.6
8	6	5	5	6	6	5.6	0.5
16	7	7	7	6	8	7	0.6
32	9	7	7	7	8	7.6	0.8

For Size = 100,000: All the time are in milliseconds

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	29	29	30	29	31	29.6	1.4
8	28	29	29	30	31	29.4	0.8
16	31	30	28	28	31	29.6	1.6
32	32	35	31	31	32	32.5	1.9

For Size = 1,000,000: **All the time are in milliseconds**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	170	169	170	170	170	169.8	0.4
8	165	166	166	166	166	165.8	0.4
16	164	161	164	166	162	163.4	1.74
32	165	162	163	162	163	163	1.1

For Size = 10,000,000: **All the time are in milliseconds**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	1539	1514	1510	1513	1541	1523	13.6
8	1502	1498	1500	1522	1505	1505	8.17
16	1493	1491	1494	1490	1517	1497	10.1
32	1483	1483	1521	1485	1485	1491	14.8

Code:

```
#include <iostream>
#include <vector>
#include <thread>
#include <random>
#include <fstream>
#include <chrono>
#include <mutex>
using namespace std;

struct padded_int {
int value;
char padding[60];
};


int t, length, count;

mutex m;

void count3s_C_thread(int thread_number, vector<int>& array_3)
{
    // thread local counter with padding
    padded_int countLocal;
    countLocal.value = 0;

    int length_per_thread = ::length / ::t;
    int start = thread_number * length_per_thread;
    // loop through the local indexes and cound the 3's
    for (int i = start; i < start + length_per_thread; i++)
    {
        if (array_3[i] == 3)
        {
            countLocal.value ++;

        }
    }
    // updating the global values with local counts with lock to avaid racing
    m.lock();
    ::count+=countLocal.value;
    m.unlock();
}

// This method will initiate all the threads
void count3s_C_By_Thread_Func(vector<int>& array_3, int array_len, int threadNumber){
    // initializing the global vales
    vector<thread> threads;
    ::count = 0;
    ::t = threadNumber;
    ::length = array_len;
    // creating the threads
    for (int thread_number = 0; thread_number < t; thread_number++){
        threads.emplace_back(thread(count3s_C_thread, thread_number, ref(array_3)));
    }
    // waiting for child threads to finish and join back
    for (thread &thread : threads){
        thread.join();
    }
    // showing the output
    cout << "Total Count of 3 from C-Threads: " << ::count << endl;

}

int main(int argc, char *argv[])
{
    if (argc != 3) {
        cerr << "Usage: " << argv[0] << " <number of threads>" << endl;
        return 1;
    }

    //fetching the command line arguments
    int thread_num = stoi(argv[1]);
    int array_len = stoi(argv[2]);

    // starting the timer
    auto currentTime_start = chrono::system_clock::now();
    auto millis_start =
chrono::duration_cast<chrono::milliseconds>(currentTime_start.time_since_epoch()).coun
t();

    // taking the file pointer access to read
    FILE *fp;
    if ((fp = fopen("input.txt", "r")) == NULL) {
      fprintf(stderr, "Error: Unable to open the file.\n");
      return 1;
    }
    int x, actualCount=0;
    vector<int> array_3;
    // reading the file and getting the integer values of the array
    for(int i=0; i<array_len; i++) {
        fscanf(fp,"%d", &x);
        array_3.push_back(x);
        if(x==3) actualCount++;
    }
    // actual count 3 by serial calculation
    cout << "Actual Count of 3 by serial: "<< actualCount<<endl;
    count3s_C_By_Thread_Func(array_3, array_len, thread_num);

    // end the timer and calculate runtime
    auto currentTime_end = chrono::system_clock::now();
    auto millis_end =
chrono::duration_cast<chrono::milliseconds>(currentTime_end.time_since_epoch()).count(
);
    auto execTime = millis_end - millis_start;
    cout << "Total Execution time: "<<execTime <<" ms"<< endl;
    return 0;
}
```

Analysis of C-OpenMP

For Size = 10,000: All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	3	5	5	5	5	4.6	0.8
8	6	5	5	5	5	4.8	0.6
16	6	6	6	6	6	6	0
32	7	7	7	7	8	7.2	0.4

For Size = 100,000: All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	29	30	31	31	32	30.6	1.02
8	29	30	31	29	29	29.6	.75
16	29	28	32	29	29	29.5	1.2
32	29	22	30	30	28	27.8	3

For Size = 1,000,000: **All the time are in millisecond**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	169	167	166	167	165	166	.5
8	164	165	165	164	165	164	.5
16	161	163	164	164	163	163	1.1
32	162	168	162	163	161	163	2.5

For Size = 10,000,000: **All the time are in millisecond**

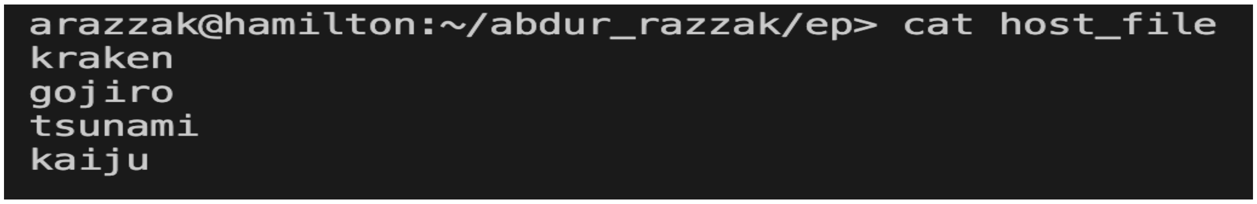
Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	1518	1513	1509	1515	1511	1513	3.12
8	1496	1498	1504	1503	1498	1500	3.14
16	1492	1491	1490	1488	1494	1491	18.0
32	1530	1502	1485	1484	1483	1497	18.1

Code:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>
#include <omp.h>
using namespace std;

int count;

// This method will run OpenMP parallilism on array_3 to find the count of 3
void count3s_usingOpenMP(vector<int>& array_3, int thread_numbers, int array_len){
    int i, count_p;
    // setting up the thread number
    omp_set_num_threads(thread_numbers);
    #pragma omp parallel shared(array_3, count, array_len) private(count_p, i)
    {
        // thread local area to find the 3 of local part
        count_p = 0;
        #pragma omp for
        for(i=0; i<array_len; i++){
            if(array_3[i]==3){
                count_p++;
            }
        }
        #pragma omp critical
        {
            // adding back to the global sum
            count+=count_p;
        }
    }
    cout << "Count of 3 from OpenMP: " << count << endl;

}

int main(int argc, char *argv[])
{
    if (argc != 3) {
        cerr << "Usage: " << argv[0] << " <number of threads>" << endl;
        return 1;
    }

    //Accessing command line arguments
    int thread_num = stoi(argv[1]);
    int array_len = stoi(argv[2]);

    // staring the timer
    auto currentTime_start = chrono::system_clock::now();
    auto millis_start =
chrono::duration_cast<chrono::milliseconds>(currentTime_start.time_since_epoch()).coun
t();

    // accessing the file pointer to read the values
    FILE *fp;
    if ((fp = fopen("input.txt", "r")) == NULL) {
      fprintf(stderr, "Error: Unable to open the file.\n");
      return 1;
    }
    int x, actualCount=0;
    // reading the values from the file
    vector<int> array_3;
    for(int i=0; i<array_len; i++) {
        fscanf(fp,"%d", &x);
        array_3.push_back(x);
        if(x==3) actualCount++;
    }

    cout << "Actual Count of 3 by serial: "<< actualCount<<endl;
    count = 0;
    count3s_usingOpenMP(array_3, thread_num, array_len);

    // taking the end time and calculating the execution time
    auto currentTime_end = chrono::system_clock::now();
    auto millis_end =
chrono::duration_cast<chrono::milliseconds>(currentTime_end.time_since_epoch()).count(
);
    auto execTime = millis_end - millis_start;
    cout << "Total Execution time: "<<execTime <<" ms"<< endl;

    return 0;
}
```

Analysis of C-MPI

MPI was running with 4 nodes connected cluster.



```
arazzak@hamilton:~/abdur_razzak/ep> cat host_file
kraken
gojiro
tsunami
kaiju
```

For Size = 10,000:   All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	225	260	245	233	246	242	10.7
8	230	256	232	242	215	235	13.4
16	245	256	244	275	252	254	11.2
32	412	437	414	417	415	419	9.0

For Size = 100,000: All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	267	261	255	272	254	261	6.31
8	265	260	254	258	256	258	3.77
16	294	256	308	373	304	254	32
32	475	468	454	467	462	465	6.36

For Size = 1,000,000: **All the time are in millisecond**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	393	386	365	371	360	375	9.6
8	358	379	389	377	378	276	7.2
16	437	377	405	415	412	409	18.9
32	489	485	489	524	537	505	7.8

For Size = 10,000,000: **All the time are in millisecond**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	1525	1502	1481	1509	1490	1501	9.6
8	1522	1485	1491	1503	1492	1496	7.2
16	1524	1532	1548	1538	1486	1525	18.9
32	1703	1688	1731	1736	1700	1711	7.8

Code:

```c
#include <mpi.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int globalCount;

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

  if (argc != 2) {
    printf("Argument Error\n");
    return 1;
  }
  // Fetching the length from arguments
  int length = atoi(argv[1]);

  clock_t start, end;
  double runtime;
  // starting the time
  start = clock();

  int world_size;
  int myId, value, numProcs, tag = 101;

  // setting up the environemnt to run MPI
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myId);

  // preparing the subarray for each threads
  int length_per_process=length/numProcs;
  int *myArray=(int *) malloc(length_per_process*sizeof(int));

  FILE *fp;
  int p, i,j,k,l;

  if (myId == 0){
    // the parent thread will read the file for the rest of the child thread
    if ((fp = fopen("input.txt", "r")) == NULL) {
      fprintf(stderr, "Error: Unable to open the file.\n");
      return 1;
    }
    for(p=0; p<numProcs-1; p++) {
      for(i=0; i<length_per_process; i++) {
        fscanf(fp,"%d", myArray+i);
      }
      // sending the sub array to each child thread.
      MPI_Send(myArray, length_per_process, MPI_INT, p+1, tag, MPI_COMM_WORLD);
    }
    // processing some part for the parent thread itself.
    for(i=0; i<length_per_process; i++) {
        fscanf(fp,"%d", myArray+i);
    }
  }
  else {
    // will receive the responses from the child threads.
    MPI_Recv(myArray, length_per_process, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
  }

  //thread local part
  int myCount = 0;
  for(i=0; i<length_per_process; i++) {
    if(myArray[i]==3) {
      myCount++;
    }
  }
  // doing the reduce operation to calculate all the sum of 3 into global count.
  MPI_Reduce(&myCount, &globalCount, 1, MPI_INT, MPI_SUM, 0 , MPI_COMM_WORLD);

  if(myId==0) {
    // ending the execution time.
    end = clock();
    printf("Count of 3 from MPI: %d\n", globalCount);
    runtime = ((double) (end - start)) / CLOCKS_PER_SEC * 1000.0;
    printf("Total Execution time: %f ms\n", runtime);
  }
  MPI_Finalize();
}
```

Analysis of Java-Threads

For Size = 10,000: All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	98	93	94	106	91	96	5.3
8	95	95	93	91	94	94	1.5
16	95	96	94	96	105	97	3.9
32	118	114	102	104	113	110	7.22

For Size = 100,000: All the time are in millisecond

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	209	197	197	197	197	199	4.8
8	198	198	198	197	193	196.8	1.92
16	191	189	194	190	189	196.55	1.85
32	187	189	200	193	192	192.2	4.44

For Size = 1,000,000: **All the time are in millisecond**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	738	725	768	725	728	737	16.21
8	749	793	784	758	795	778	17.18
16	767	747	763	770	758	781	8.7
32	709	717	703	721	707	711	6.6

For Size = 10,000,000: **All the time are in millisecond**

Number of Threads	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
4	5572	5572	5568	5443	5553	5541	49.8
8	5578	5508	5438	5673	5599	5559	73.2
16	5622	5611	5600	5254	5501	5517	138.7
32	5641	5637	5786	5780	5743`	5717	22.3

Code:

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
import java.util.concurrent.*;

public class EvaluatePerformanceJavaThreads implements Runnable {
    private static int ARRAY_LENGTH = 1000000;
    private static int MAX_THREADS=10;
    private static final int MAX_RANGE = 5;
    private static final Random random = new Random();
    private static int count = 0;
    private static Object lock=new Object();
    private static int[] array;
    private static Thread[] t;

    public static void main(String[] args) {
        if (args.length != 0){
            // fetching the argument values for thread number and array size
            MAX_THREADS = Integer.parseInt(args[0]);
            ARRAY_LENGTH = Integer.parseInt(args[1]);
        }
        // start counting time
        long startTime = System.currentTimeMillis();

        // initializing variables
        array = new int[ARRAY_LENGTH];
        t = new Thread[MAX_THREADS];

        int i=0;
        int actualCount=0;
        try {
            // accessing the input file to read the values
            File file = new File("input.txt");
            Scanner scanner = new Scanner(file);

            // Read input from the file using the Scanner
            while (scanner.hasNextInt()) {
                int number = scanner.nextInt();
                if(number == 3) actualCount++;
                array[i] = number;
                i++;
            }

            // Close the Scanner
            scanner.close();
        } catch (FileNotFoundException e) {
            // Handle the case where the file is not found
            e.printStackTrace();
        }

        // creating the threads
        EvaluatePerformanceJavaThreads[] counters = new
EvaluatePerformanceJavaThreads[MAX_THREADS];
        int lengthPerThread = ARRAY_LENGTH / MAX_THREADS;
        for (i = 0; i < counters.length; i++) {
            counters[i] = new EvaluatePerformanceJavaThreads(i * lengthPerThread,
                    lengthPerThread);
        }
        // run the threads
        for (i = 0; i < counters.length; i++) {
            t[i] = new Thread(counters[i]);
            t[i].start();
        }
        // wait for all child threads to finish and join
        for (i = 0; i < counters.length; i++) {
            try {
                t[i].join();
            } catch (InterruptedException e) {
                /* do nothing */ }
        }
        int notTakenCareIndex;
        int localCount = 0;
        if (ARRAY_LENGTH % MAX_THREADS != 0){
            notTakenCareIndex = (ARRAY_LENGTH / MAX_THREADS) * MAX_THREADS;
            while(notTakenCareIndex < ARRAY_LENGTH) {
                if(array[notTakenCareIndex] == 3){
                    localCount++;
                }
                notTakenCareIndex++;
            }
            count += localCount;
        }

        // end time and calculating the runtime
        long endTime = System.currentTimeMillis();
        System.out.println("Number of threes in Serial Count: "+actualCount);
        System.out.println("Number of threes in Java Threads: "+count);
        System.out.println("Parallel counting time: "+ String.valueOf(endTime-
startTime)+" ms");
    }

    private int startlndex;
    private int elements;
    private int myCount=0;

    public EvaluatePerformanceJavaThreads(int start, int elem) {
        startlndex=start;
        elements=elem;
    }
    //Overload of run method in the Thread class
    public void run() {
        //count the number of threes
        for(int i=0; i<elements; i++) {
            if(array[startlndex+i]==3) {
                myCount++;
            }
        }
        //implementing cricitical code section with synchronization
        synchronized(lock) {
            count+=myCount;
        }
    }
}
```

Analysis of CUDA: Max Block

For Size = 10,000: All the time are in millisecond

Block and Grid	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
767 96	91	83	80	80	97	86	6.7

For Size = 100,000: All the time are in millisecond

Block and Grid	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
767 96	108	105	108	94	92	101	6.97

For Size = 1,000,000: All the time are in millisecond

Block and Grid	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
767 96	175	150	179	150	184	167	13.4

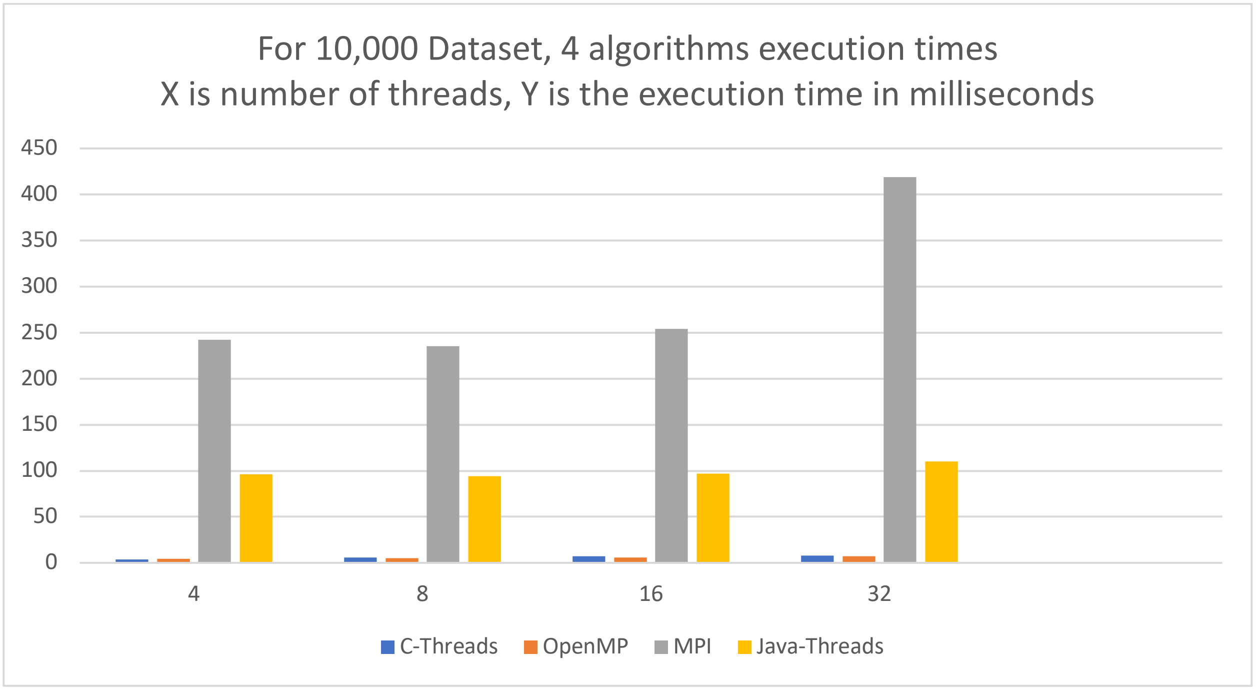
For Size = 10,000,000: All the time are in millisecond

Block and Grid	Run 1	Run 2	Run 3	Run 4	Run 5	Avg Run	SD Run
767 96	631	608	602	610	590	608	13.4

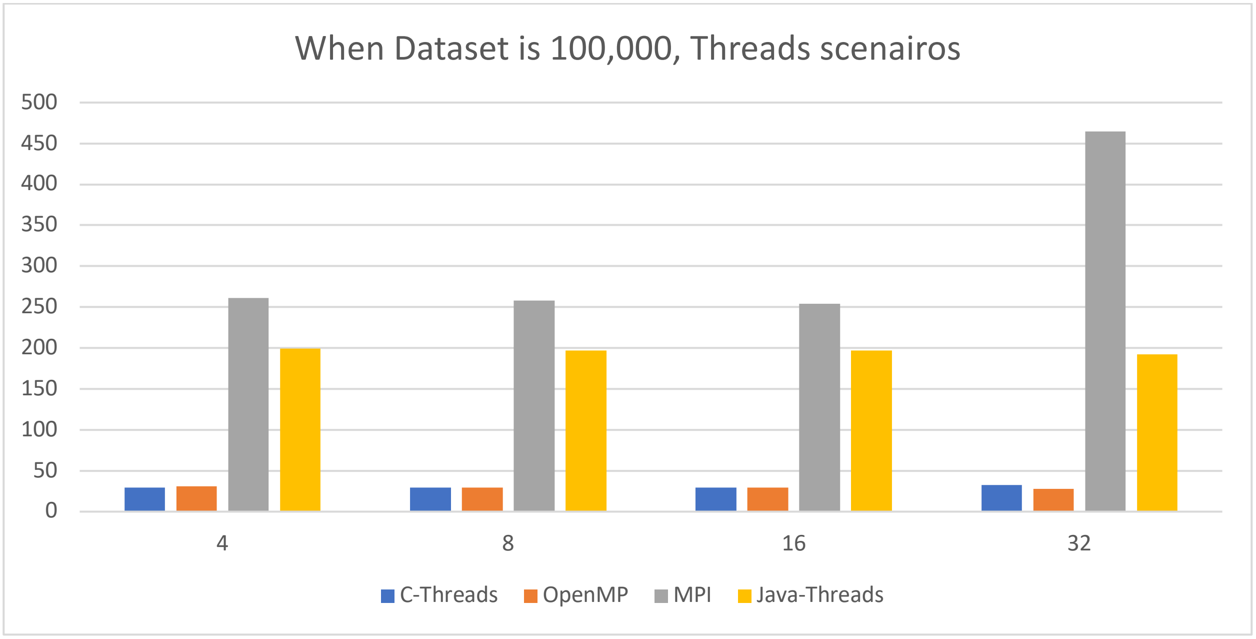
Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <device_launch_parameters.h>
#include <cuda_runtime.h>

// CUDA kernel to count occurrences of the value 3 in the chunk of array
__global__ void count3s_kernel(const int* array, size_t size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int localCount = 0;

    for (int i = tid; i < size; i += blockDim.x * gridDim.x) {
        if (array[i] == 3) {
            localCount++;
        }
    }

    atomicAdd(result, localCount);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Error: Argument is not proper");
        return 1;
    }
    // Fetching the array length from argument
    int arraySize = atoi(argv[1]);

    clock_t start, end;
    double runtime;
    // timer starts
    start = clock();

    // accessting the file pointer to read
    FILE *fp;
    if ((fp = fopen("input.txt", "r")) == NULL) {
      fprintf(stderr, "Error: Unable to open the file.\n");
      return 1;
    }
    int x, actualCount=0, i;
    // reading the values from file and keeping them in host array.
    int* hostArray = new int[arraySize];
    for(i=0; i<arraySize; i++) {
        fscanf(fp,"%d", &x);
        hostArray[i] = x;
        if(x==3) actualCount++;
    }
    // allocating memory for the gpu threads and their local result
    int* deviceArray;
    int* deviceResult;
    cudaMalloc((void**)&deviceArray, arraySize * sizeof(int));
    cudaMalloc((void**)&deviceResult, sizeof(int));

    // Copy array from host to device
    cudaMemcpy(deviceArray, hostArray, arraySize * sizeof(int),
cudaMemcpyHostToDevice);

    // Initialize result on the device
    cudaMemset(deviceResult, 0, sizeof(int));

    // finding the maximum block size in order to ensure maximum parallilism.
    int maxBlockSize, GridSize;
    cudaOccupancyMaxPotentialBlockSize(&GridSize, &maxBlockSize, count3s_kernel, 0,
0);
    printf("Block used: %d\n", maxBlockSize);
    printf("Grid used: %d\n", GridSize);

    // Launch CUDA kernel to count occurrences of '3' in deviceArray
    count3s_kernel<<<GridSize, maxBlockSize>>>(deviceArray, arraySize, deviceResult);

    // Copy result back to host
    int hostResult;
    cudaMemcpy(&hostResult, deviceResult, sizeof(int), cudaMemcpyDeviceToHost);

    // Print the result
    printf("Count from CUDA: %d\n", hostResult);

    // timer end
    end = clock();
    runtime = ((double) (end - start)) / CLOCKS_PER_SEC * 1000.0;
    printf("Total Execution time: %f ms\n", runtime);

    // Cleanup
    delete[] hostArray;
    cudaFree(deviceArray);
    cudaFree(deviceResult);
    return 0;
}
```

Graphs



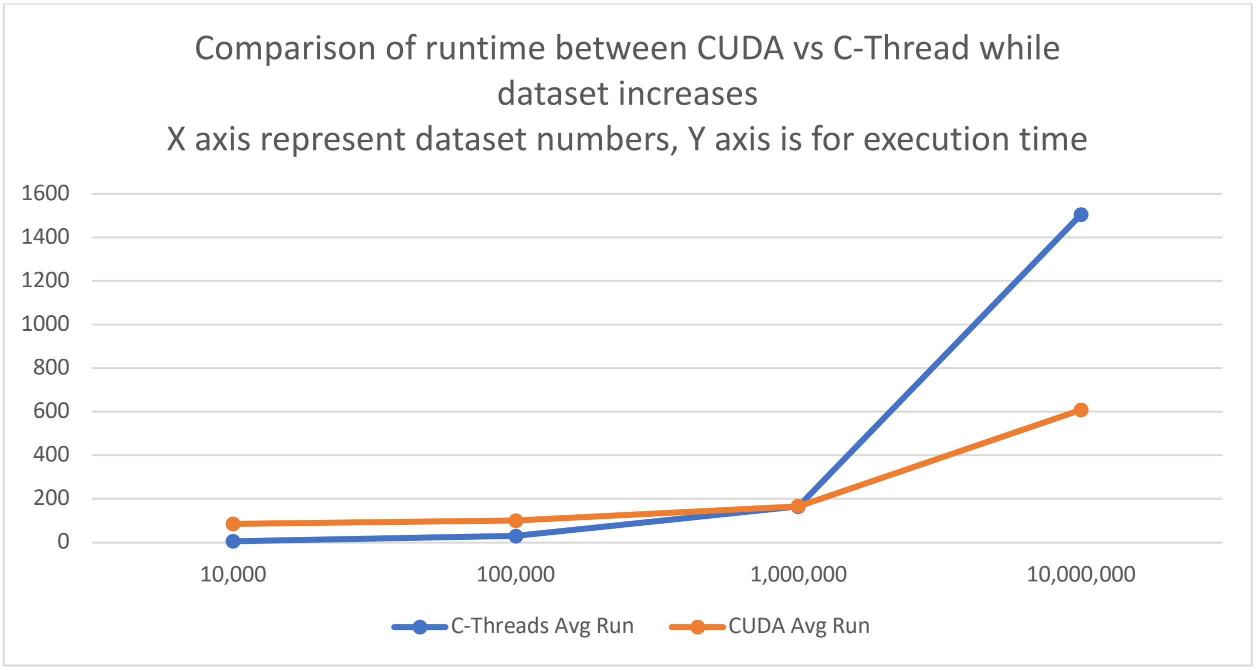
Both graphs [top and bottom one] where data is 10k or 100k, MPI and Java threads are performing bad, because MPI works with inter processing messaging in which processes communication always add overhead. C-threads and OpenMP is performing better than them way better.



However, when the data set increased, MPI's execution time become reduced and closer to the C-threads or OpenMP runtime.

As we can see C-threads and OpenMP is performing better regardless the scenarios whether data size is big or small, now let's compare them with CUDA performance.

In the below picture, when the dataset increases a lot, then fixed number of parallelisms like c-threads cannot offer good performance. Cuda shows much improvement as it is running with a high number of threads such as 768 blocks and 96 grids.



Which One is Better?

In my honest opinion, It depends on the dataset size. If the data is small then, C-thread and OpenMP would be better than others. Because we can scale the code in more detail and delicate level that enhance the performance such as padding. Also, introducing a lot of processing small number of data would increase communication overhead rather reducing the runtime. However, when the data is huge, then we need more and more parallel processors rather than small number of perfect and optimized c-threads. Therefore, CUDA with I think go to tool for really large number of data and c-thread or OpenMP for small parallelism or fixed parallelism.

Conclusion

This performance evaluation in different platform was interesting. It nicely explained how various platform of threads can be differ based on the dataset. Also showed by the graph that how too many threads would cost a high communication overhead between the nodes.

C++ Threads provides more explicit and fine-grained control over threads compared to OpenMP. It is suitable for scenarios where precise control over synchronization, data sharing, and thread lifecycle is crucial.

OpenMP: It provides an easy way to run code in parallel.

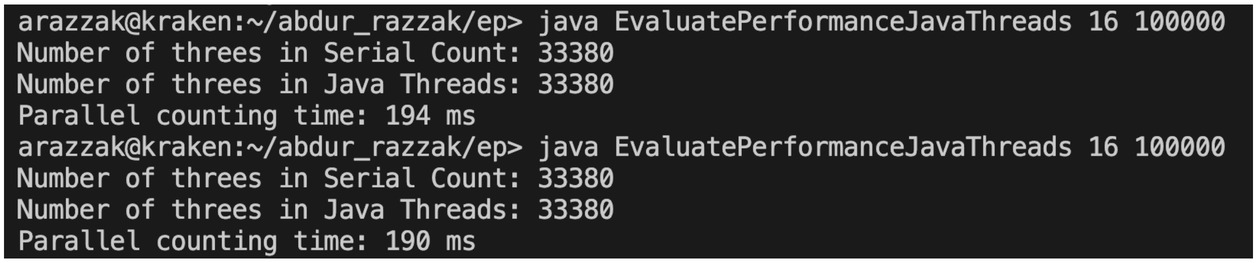
CUDA: CUDA has Nvidia GPUs. It offers massive number of parallel applications. CUDA providing high-performance parallelism on compatible Nvidia hardware.

Java Multithreading: Java Multithreading, with its built-in support for threads and synchronization, offers a platform-independent approach to parallelism. However, it is slow compared to others.

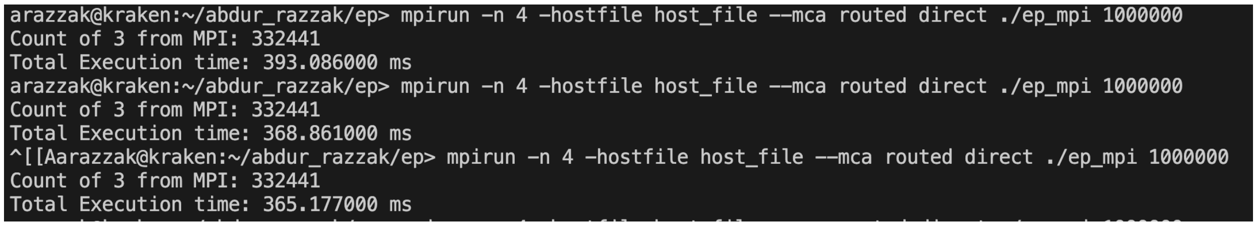
Reference:

Most of the codes are tested and modified after taking from previous assignments. The CUDA is was collected from chatgpt however, it required a log of changes in the which requires deep understanding of the code. Graphs are generated from Microsoft excel.

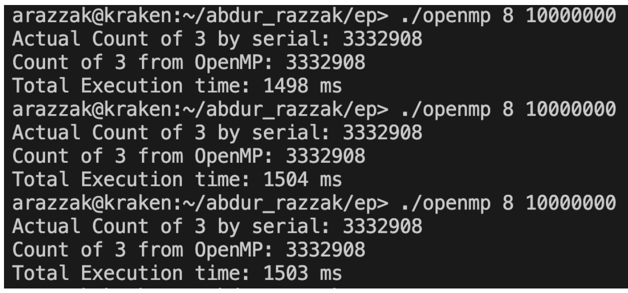
Some Screenshots for Program Running



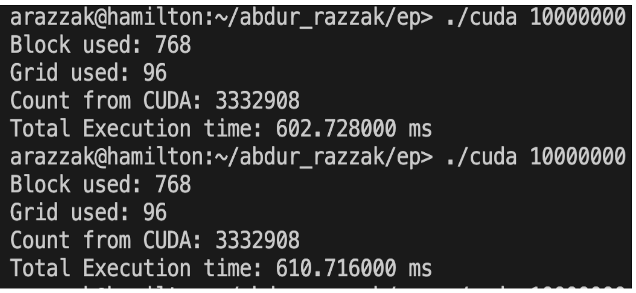
```
arazzak@kraken:~/abdur_razzak/ep> java EvaluatePerformanceJavaThreads 16 100000
Number of threes in Serial Count: 33380
Number of threes in Java Threads: 33380
Parallel counting time: 194 ms
arazzak@kraken:~/abdur_razzak/ep> java EvaluatePerformanceJavaThreads 16 100000
Number of threes in Serial Count: 33380
Number of threes in Java Threads: 33380
Parallel counting time: 190 ms
```



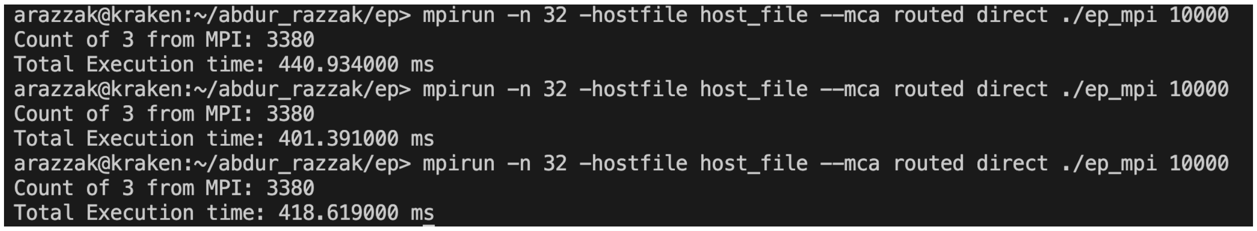
```
arazzak@kraken:~/abdur_razzak/ep> mpirun -n 4 -hostfile host_file --mca routed direct ./ep_mpi 1000000
Count of 3 from MPI: 332441
Total Execution time: 393.086000 ms
arazzak@kraken:~/abdur_razzak/ep> mpirun -n 4 -hostfile host_file --mca routed direct ./ep_mpi 1000000
Count of 3 from MPI: 332441
Total Execution time: 368.861000 ms
^[[Aarazzak@kraken:~/abdur_razzak/ep> mpirun -n 4 -hostfile host_file --mca routed direct ./ep_mpi 1000000
Count of 3 from MPI: 332441
Total Execution time: 365.177000 ms
```



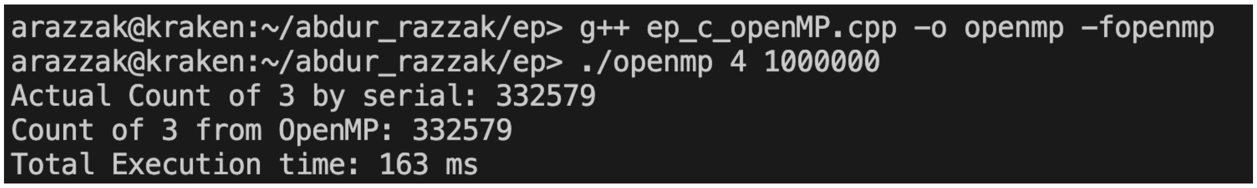
```
arazzak@kraken:~/abdur_razzak/ep> ./openmp 8 10000000
Actual Count of 3 by serial: 3332908
Count of 3 from OpenMP: 3332908
Total Execution time: 1498 ms
arazzak@kraken:~/abdur_razzak/ep> ./openmp 8 10000000
Actual Count of 3 by serial: 3332908
Count of 3 from OpenMP: 3332908
Total Execution time: 1504 ms
arazzak@kraken:~/abdur_razzak/ep> ./openmp 8 10000000
Actual Count of 3 by serial: 3332908
Count of 3 from OpenMP: 3332908
Total Execution time: 1503 ms
```



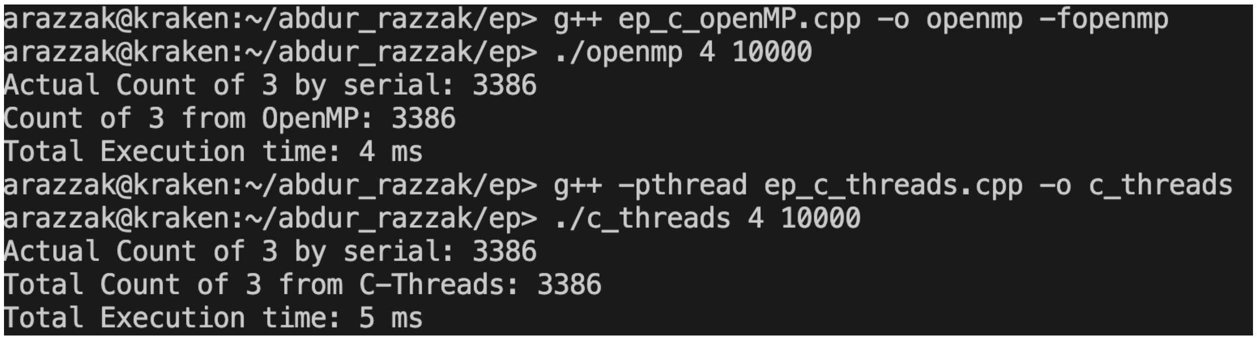
```
arazzak@hamilton:~/abdur_razzak/ep> ./cuda 10000000
Block used: 768
Grid used: 96
Count from CUDA: 3332908
Total Execution time: 602.728000 ms
arazzak@hamilton:~/abdur_razzak/ep> ./cuda 10000000
Block used: 768
Grid used: 96
Count from CUDA: 3332908
Total Execution time: 610.716000 ms
```



```
arazzak@kraken:~/abdur_razzak/ep> mpirun -n 32 -hostfile host_file --mca routed direct ./ep_mpi 10000
Count of 3 from MPI: 3380
Total Execution time: 440.934000 ms
arazzak@kraken:~/abdur_razzak/ep> mpirun -n 32 -hostfile host_file --mca routed direct ./ep_mpi 10000
Count of 3 from MPI: 3380
Total Execution time: 401.391000 ms
arazzak@kraken:~/abdur_razzak/ep> mpirun -n 32 -hostfile host_file --mca routed direct ./ep_mpi 10000
Count of 3 from MPI: 3380
Total Execution time: 418.619000 ms
```



```
arazzak@kraken:~/abdur_razzak/ep> g++ ep_c_openMP.cpp -o openmp -fopenmp
arazzak@kraken:~/abdur_razzak/ep> ./openmp 4 1000000
Actual Count of 3 by serial: 332579
Count of 3 from OpenMP: 332579
Total Execution time: 163 ms
```



```
arazzak@kraken:~/abdur_razzak/ep> g++ ep_c_openMP.cpp -o openmp -fopenmp
arazzak@kraken:~/abdur_razzak/ep> ./openmp 4 10000
Actual Count of 3 by serial: 3386
Count of 3 from OpenMP: 3386
Total Execution time: 4 ms
arazzak@kraken:~/abdur_razzak/ep> g++ -pthread ep_c_threads.cpp -o c_threads
arazzak@kraken:~/abdur_razzak/ep> ./c_threads 4 10000
Actual Count of 3 by serial: 3386
Total Count of 3 from C-Threads: 3386
Total Execution time: 5 ms
```