DAPA ASSIGNMENT

Generation of Combinations
Lexicographically

Project done by GROUP 14:

• Subhradeep Saha	106118095
• Yash Shah	106118107
• RVS SATYANAND	106118083

Problem Definition

Combinations refer to the combination of n things taken k at a time without repetition. To refer to combinations in which repetition is allowed, the terms k-selection, k-multiset, or k-combination with repetition are often used.

The combinations are generated in the groups of ascending nature in a lexicographical order, i.e. a general order of the way words are alphabetically ordered based on the alphabetical order of their component letters.

The corresponding algorithm is performed sequentially (normal execution) as well as parallelly (execution via threads and CUDA).

Assumptions

Several assumptions can be made via using the architecture used to make efficient processing such as, to enable a faster computation.

- The parallel algorithm uses an *EREW SM SIMD* model of computation.
- The time needed for thread creation is significant, which affects the time taken in parallel part of the algorithm, when the value of *n* is less.
- All the threads work independently with enough computations resources, so that there is never a case of deadlock, that can arise between them.

Algorithms

Sequential

```
procedure sequentialCombination (n, m)
    Step 1:
        (1.1) combinationArr ← [1, 2, 3, … , m]
        (1.2) produce combinationArr as output
    Step 2:
        (2.1) totalCombination ← nCr (n,m) - 1
        (2.2) for i=1 to totalCombination do
                  nextCombination (combinationArr,n,m)
              endfor

procedure nextCombination (combinationArr, n, m)
    Step 1:  j ← m
    Step 2:  while (j>0) do
                if (combinationArr[ j-1 ] < n-m+j) do
                    (2.1) combinationArr[ j-1 ] ← combinationArr[ j-1 ] + 1
                    (2.2) for i=j to m-1 do
                              combinationArr[i] ← combinationArr[ i-1 ] + 1
                    (2.3) produce combinationArr as output
                    (2.4) break
                else
                    j ← j-1
                endif
             endwhile

procedure nCr (n, r)
    Step 1:
        (1.1) p ← 1
        (1.2) k ← 1
        (1.2) if (n - r < r) do
                  r ← n - r
              endif
    Step 2:
        if (r != 0) do
            while (r) do
```

```
                (2.1) p ← p*n
                (2.2) k ← k*r
                (2.3) m ← gcd(p, k)   // gcd - greatest common divisor
                (2.4) p ← p/m
                (2.5) k ← k/m
                (2.6) n ← n-1
                (2.7) r ← r-1
            endwhile
        else
            p ← 1
        endif
    Step 3:
        return p
```

Parallel

```
procedure parallelCombination (n, m)
    Step 1:
        (1.1) MAX ← 100
        (1.2) declare integer variables k
        (1.2) declare integer arrays x[MAX], y[MAX], c[MAX]
        (1.3) declare boolean array z[MAX]
    Step 2:
        (2.1) for i=1 to m do in parallel
                initialize(i)
              endfor
        (2.2) produce c as output
    Step 3:
        while (z[0] == 0) do
            (3.1) k ← 0
            (3.2) for i=2 to min(m,k) do in parallel
                      task1(i)
                  endfor
            (3.3) if (k==0) do
                      y[ m-1 ] ← y[ m-1 ] + 1
                  else
                      for i=k to m do in parallel
                          task2(i)
                      endfor
                  endif
            (3.4)
            for i=1 to m do in parallel
                      task3(i)
            endfor
            (3.5) produce c as output
    endwhile

procedure initialize (i)
    Step 1: x[ i-1 ] ⬅ n-m+i
    Step 2: y[ i-1 ] ⬅ i
    Step 3: if (y[ i-1 ] == x[ i-1 ]) do
            z[ i-1 ] ⬅  1
        else
            z[ i-1 ] ⬅ 0
        endif
    Step 4: c[ i-1 ] ⬅ i

procedure task1 (i)
    if (z[ i-2 ] == 0 and z[ i-1 ] == 1) do
        (a) y[ i-2 ] ⬅ y [ i-2 ] + 1
        (b) k ⬅ i
    endif

procedure task2 (i)
    y[ i-1 ] ⬅ y[ k-2 ] + i - k + 1

procedure task3 (i)
    Step 1: c[ i-1 ] ⬅ y[ i-1 ]
    Step 2:
        if (y[ i-1 ] == x[ i-1 ]) do
        z[ i-1 ] ⬅ 1
        else
        z[ i-1 ] ⬅ 0
    endif
```

Implemented code (in C++)

Sequential code

```cpp
#include <bits/stdc++.h>
using namespace std;

long long nCr(long long n, long long r)
{
    long long p = 1, k = 1;

    if (n - r < r)
        r = n - r;

    if (r != 0) {
        while (r) {
            p *= n;
            k *= r;

            long long m = __gcd(p, k);

            p /= m;
            k /= m;

            n--;
            r--;
        }
    }

    else
        p = 1;

    return p;
}

void printArray(long long arr[], long long n)
{
    for (long long i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
void nextCombination(long long combinationArr[], long long n, long long m)
{
    long long j = m;
    while (j > 0)
    {
        if (combinationArr[j - 1] < n - m + j)
        {
            combinationArr[j - 1]++;

            for (int i = j; i < m; i++)
                combinationArr[i] = combinationArr[i - 1] + 1;

            printArray(combinationArr, m);
            break;
        }
        else
            j--;
    }
}


void sequentialCombination(long long n, long long m)
{
    long long combinationArr[m];

    //initialize the array
    for (long long i = 0; i < m; i++)
        combinationArr[i] = i + 1;

    printArray(combinationArr, m);

    //number of iteration for the loop to run
    long long totalCombination = nCr(n, m) - 1;

    for (long long i = 0; i < totalCombination; i++)
        nextCombination(combinationArr, n, m);
}
int main()
```

```
{
    long long n, m;
    cout << "\nEnter the values of n and m (n>m) respectively: ";
    cin >> n >> m;

    clock_t start, end;
    start = clock();

    cout << "\nFollowing are the different combinations formed:\n";
    sequentialCombination(n, m);

    end = clock();
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "\nTime taken for sequential algorithm: " << time_taken << "secs" << "\n\n";
}
```

Parallel Code(using threads)

```
#include <bits/stdc++.h>
#include <thread>
#define MAX 100

using namespace std;

int N, M, k;
int x[MAX], y[MAX], c[MAX];
bool z[MAX];

void initialize(int i)
{
    int n = N;
    int m = M;
    x[i - 1] = n - m + i;
    y[i - 1] = i;
    if (y[i - 1] == x[i - 1])
        z[i - 1] = 1;
    else
```

```
        z[i - 1] = 0;
    c[i - 1] = i;
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void printboolArray(bool arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void task1(int i)
{
    if (z[i - 2] == 0 && z[i - 1] == 1)
    {
        y[i - 2] = y[i - 2] + 1;
        k = i;
    }
}

void task2(int i)
{
    y[i - 1] = y[k - 2] + (i - k + 1);
}

void task3(int i)
{
    c[i - 1] = y[i - 1];
    if (y[i - 1] == x[i - 1])
        z[i - 1] = 1;
    else
        z[i - 1] = 0;
```

```
}

int main()
{
    cout << "\nEnter the values of n and m (n>m) respectively: ";
    cin >> N >> M;

    cout << "\nFointowing are the combinations formed: \n";

    clock_t start, end;
    start = clock();

    vector<thread> threads;
    for (int i = 1; i <= M; i++)
    {
        threads.push_back(thread(initialize, i));
    }
    for (auto &th : threads)
    {
        th.join();
    }
    threads.clear();

    printArray(c, M);

    while (z[0] == 0)
    {
        threads.clear();
        k = 0;
        for (int i = 2; i <= M; i++)
        {
            threads.push_back(thread(task1, i));
            if (k == i)
                break;
        }
        for (auto &th1 : threads)
        {
            th1.join();
        }
        threads.clear();
        if (k == 0)
        {
            y[M - 1] = y[M - 1] + 1;
        }
        else
        {
            for (int i = k; i <= M; i++)
            {
                threads.push_back(thread(task2, i));
            }
        }
        for (auto &th2 : threads)
        {
            th2.join();
        }
        threads.clear();
        for (int i = 1; i <= M; i++)
        {
            threads.push_back(thread(task3, i));
        }
        for (auto &th3 : threads)
        {
            th3.join();
        }
        threads.clear();
        printArray(c, M);
    }

    end = clock();
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "\nTime taken for parallel algorithm: " << time_taken << "secs" << "\n\n";
}
```

Parallel Code(using CUDA)

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

#define int long long int

__device__ int Choose(int *fact, int n, int k) {
 return (n < k) ? 0 : (fact[n] / (fact[n - k] * fact[k]));
}

__global__ void PrintCombinations(int *fact, int n, int k, int *output) {
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 int m = tid;
 int nCr = fact[n] / (fact[n - k] * fact[k]);

 if (m >= nCr) {
   // printf("[thread %lld] Abort.\n", tid);
   return;
 }

 int idx = 0;

 int a = n, b = k;
 int x = (Choose(fact, n, k) - 1) - m;
 for (int i = 0; i < k; ++i) {
   a = a - 1;

   while (Choose(fact, a, b) > x) {
     a = a - 1;
   }
   output[m * k + idx++] = n - 1 - a;
   x = x - Choose(fact, a, b);
   b = b - 1;
 }
```

```
}

int32_t main() {
 int n = 5, k = 3;

 int *fact;
 fact = (int *)malloc(sizeof(int) * (n + 1));

 fact[0] = 1;
 for (int i = 1; i <= n; ++i) {
   fact[i] = fact[i - 1] * i;
 }

 printf("Factorial: ");
 for (int i = 0; i <= n; ++i) {
   printf("%lld ", fact[i]);
 }
 printf("\n");

 int nCk = fact[n] / (fact[n - k] * fact[k]);

 int *output = new int[nCk * k];
 int *d_fact, *d_output;
 cudaMalloc((void **)&d_fact, sizeof(int) * (n + 1));
 cudaMalloc((void **)&d_output, sizeof(int) * (nCk * k));
 cudaMemcpy(d_fact, fact, sizeof(int) * (n + 1), cudaMemcpyHostToDevice);
 cudaMemcpy(d_output, output, sizeof(int) * (nCk * k), cudaMemcpyHostToDevice);

 // Executing kernel
 int block_size = 256;
 int grid_size = ((nCk + block_size) / block_size);

 printf("Block size: %lld\n", block_size);
 printf("Grid size: %lld\n", grid_size);

 clock_t time_taken = clock();
 PrintCombinations<<<grid_size, block_size>>>(d_fact, n, k, d_output);
 cudaMemcpy(output, d_output, sizeof(int) * (nCk * k), cudaMemcpyDeviceToHost);
 time_taken = clock() - time_taken;

 printf("Time taken: %f s\n", ((float)time_taken / CLOCKS_PER_SEC) );

 for (int i = 0; i < nCk; ++i) {
   printf("[Case #%lld] ", i);
   for (int j = 0; j < k; ++j) {
     printf("%lld ", output[i * k + j]);
   }
   printf("\n");
 }

 cudaFree(d_fact);
 free(fact);
}
```

Analysis

Sequential

procedures **nCr** takes O(n*log(r)) and **nextCombination** takes O(m) time in the worst case.

For the main procedure **sequentialCombination**,

Step 1 will take a worst case time-complexity of O(m) for building and producing as output an array of size m.

Step 2.1 takes the time corresponding to procedure **nCr** i.e O(n*log(m)), and Step 2.2 runs a sequential for-loop for (^nC_m - 1) times which executes the procedure nextCombination inside it, so Step 2.2 takes (^nC_m - 1) * O(m) = O(m * ^nC_m) time.

Therefore the time-complexity of the whole procedure **sequentialCombination** = O(m) + O(n*log(m)) + O(m * ^nC_m) = O(m * ^nC_m)

Parallel

procedures **initialize**, **task1**, **task2** and **task3** - all take O(1) worst-case time-complexity since all the statements inside the procedures are assignment or branch statements which take constant time(O(1)) for execution.

For the main procedure **parallelCombination**,

Step 1 will take O(1) time, Step 2 will take O(1)(Step 2.1) + O(m)(Step 2.2) = O(m) time.

Step 3.2, 3.3 and 3.4 consists of parallel loops which are executing **task1**, **task2**, **task3** respectively which requires **broadcasting** to be done before execution, that results in O(log(m)) time for these steps.

The while-loop in Step 3 will take O(nCm) time for execution which leads to total execution time of the whole procedure parallelCombination =

$$O(m) + O({}^{n}C_{m}) * O(\log(m)) = O({}^{n}C_{m} * \log(m))$$

No. of processors used in **parallelCombination** = m.

Therefore,

the total cost of parallelCombination = $m * O({}^{n}C_{m} * \log(m))$

$= O(m * {}^{n}C_{m} * \log(m))$,

which is more than that of the sequential algorithm(**sequentialCombination**).

So the parallel algorithm is not optimal.

Results and Comparison

Data used for making Graph:

M = 5

N = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Parallel code time(in secs) = [0.000074, 0.000026, 0.000032, 0.000032, 0.000034, 0.000059, 0.000052, 0.000043, 0.000058, 0.000080, 0.000091, 0.000128, 0.000155, 0.000198, 0.000249]

Sequential code time(in secs) = [
0.000001, 0.000048, 0.000038, 0.000144, 0.000357, 0.000742, 0.001978, 0.003054, 0.007136, 0.009940, 0.012100, 0.026192, 0.030796, 0.043713, 0.054005]

M = 8

N = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Parallel code time(in secs) = [0.000140, 0.000033, 0.000034, 0.000037, 0.000058, 0.000076, 0.000115, 0.000228, 0.000409, 0.000572, 0.002734, 0.001474]

Sequential code time(in secs) = [0.000002, 0.000209, 0.000271, 0.001524, 0.003608, 0.011365, 0.013766, 0.025012, 0.070656, 0.186326, 0.374656, 0.532259]

M = 10

N = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

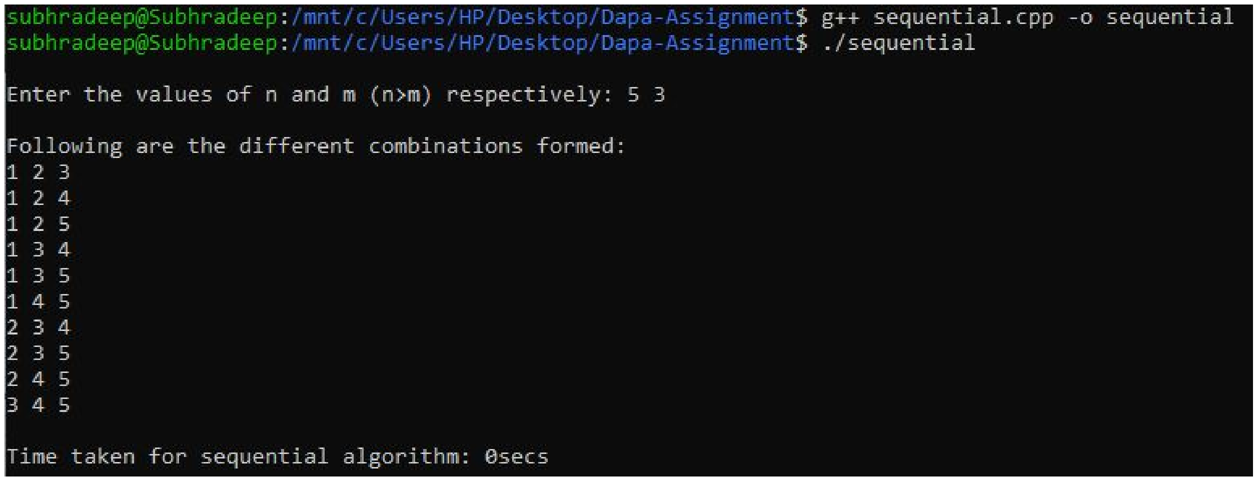
Parallel code time(in secs) = [0.000032, 0.000033, 0.000033, 0.000048, 0.000088, 0.000351, 0.000329, 0.000575, 0.001096, 0.005969, 0.013348]

Sequential code time(in secs) = [0.000001, 0.000061, 0.000350, 0.001613, 0.006815, 0.023217, 0.044254, 0.099836, 0.238743, 0.471609, 1.011403]

Output Screenshot:

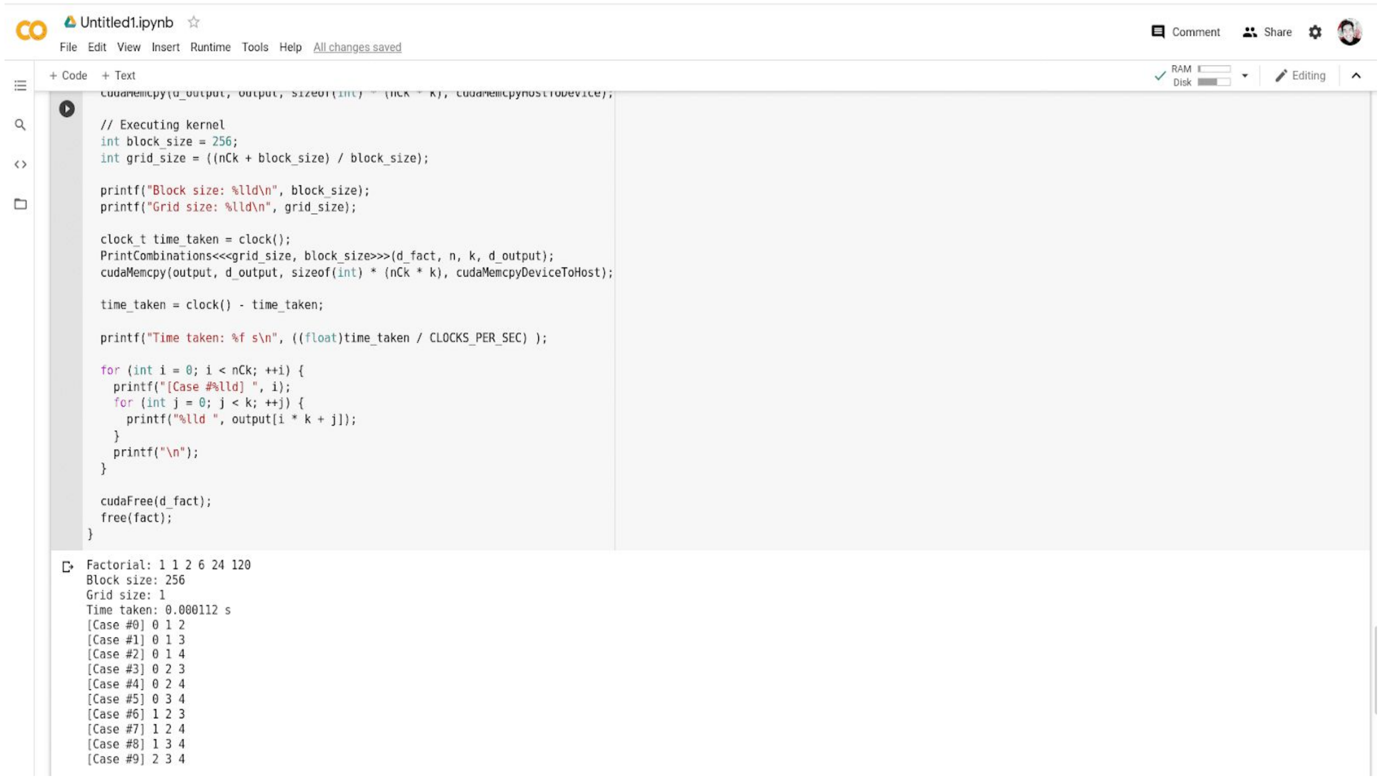
Input: n = 5, m = 3

Sequential Code:



```
subhradeep@Subhradeep:/mnt/c/Users/HP/Desktop/Dapa-Assignment$ g++ sequential.cpp -o sequential
subhradeep@Subhradeep:/mnt/c/Users/HP/Desktop/Dapa-Assignment$ ./sequential

Enter the values of n and m (n>m) respectively: 5 3

Following are the different combinations formed:
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

Time taken for sequential algorithm: 0secs
```

CUDA Code:



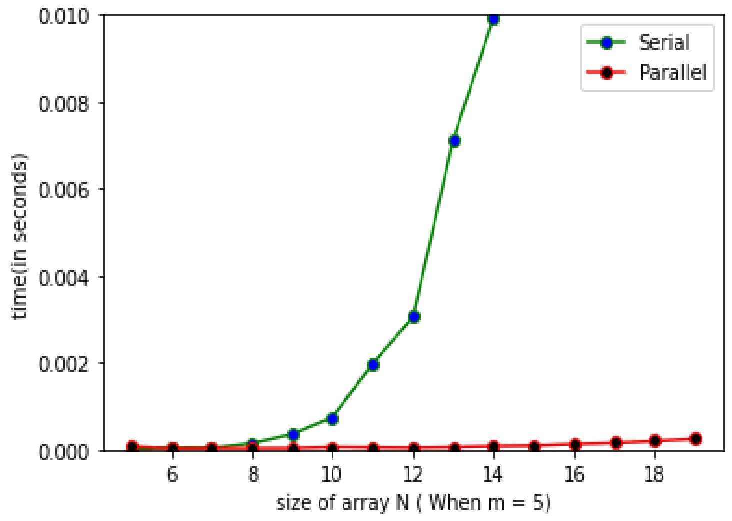
```
  cudaMemcpy(d_output, output, sizeof(int) * (nCk * k), cudaMemcpyHostToDevice);

  // Executing kernel
  int block_size = 256;
  int grid_size = ((nCk + block_size) / block_size);

  printf("Block size: %lld\n", block_size);
  printf("Grid size: %lld\n", grid_size);

  clock_t time_taken = clock();
  PrintCombinations<<<grid_size, block_size>>>(d_fact, n, k, d_output);
  cudaMemcpy(output, d_output, sizeof(int) * (nCk * k), cudaMemcpyDeviceToHost);

  time_taken = clock() - time_taken;

  printf("Time taken: %f s\n", ((float)time_taken / CLOCKS_PER_SEC) );

  for (int i = 0; i < nCk; ++i) {
    printf("[Case #%lld] ", i);
    for (int j = 0; j < k; ++j) {
      printf("%lld ", output[i * k + j]);
    }
    printf("\n");
  }

  cudaFree(d_fact);
  free(fact);
}
```

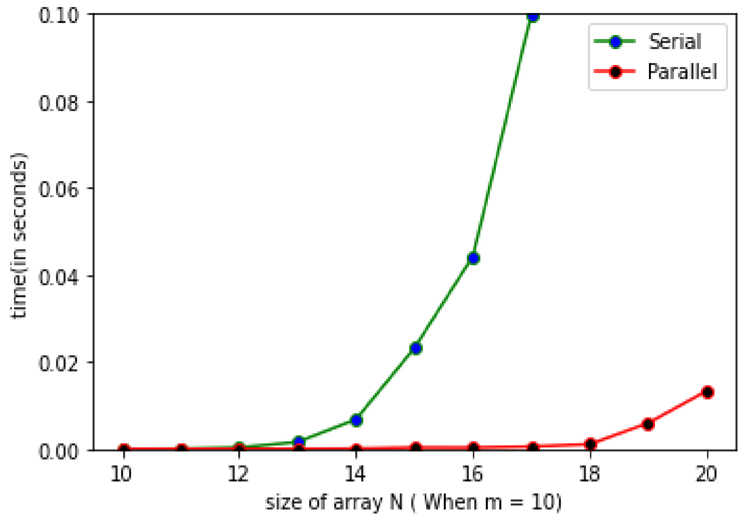
```
Factorial: 1 1 2 6 24 120
Block size: 256
Grid size: 1
Time taken: 0.000112 s
[Case #0] 0 1 2
[Case #1] 0 1 3
[Case #2] 0 1 4
[Case #3] 0 2 3
[Case #4] 0 2 4
[Case #5] 0 3 4
[Case #6] 1 2 3
[Case #7] 1 2 4
[Case #8] 1 3 4
[Case #9] 2 3 4
```

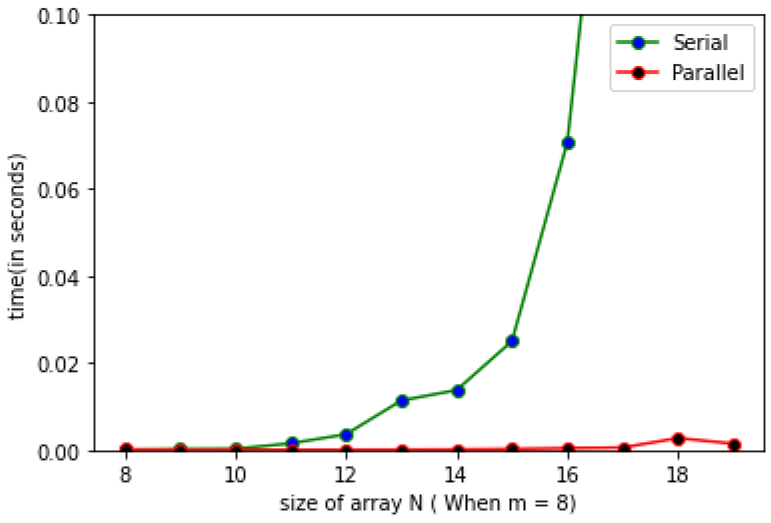
Code for Graph generation(in Python)

```
import math

import random

import time

import matplotlib.pyplot as plt

sarr = [5, 6, 7,8,9,10,11,12,13,14,15,16,17,18,19]

ptarr = [0.000074, 0.000026, 0.000032, 0.000032, 0.000034, 0.000059, 0.000052,
0.000043, 0.000058, 0.000080, 0.000091, 0.000128, 0.000155, 0.000198, 0.000249]

starr = [0.000001, 0.000048, 0.000038, 0.000144, 0.000357, 0.000742, 0.001978,
0.003054, 0.007136, 0.009940, 0.012100, 0.026192, 0.030796, 0.043713, 0.054005]

plt.plot(sarr, starr, color = 'green', marker='o', markerfacecolor='blue', label = 'Serial')

plt.plot(sarr, ptarr, color = 'red', marker='o', markerfacecolor='black', label = 'Parallel')

plt.ylim(0,0.01)



plt.xlabel('size of array N ( When m = 5)')

plt.ylabel('time(in seconds)')

plt.legend()

plt.show()
```

Graph


0.010
0.008
0.006
0.004
0.002
0.000
time(in seconds)
6
8
10
12
14
16
18
size of array N (When m = 5)
Serial
Parallel


0.10
0.08
0.06
0.04
0.02
0.00
time(in seconds)
10
12
14
16
18
20
size of array N (When m = 10)
Serial
Parallel


0.10
0.08
0.06
0.04
0.02
0.00
time(in seconds)
8
10
12
14
16
18
size of array N (When m = 8)
Serial
Parallel

Conclusion

- When the input is small, the time taken by the parallel algorithm is more than the sequential algorithm. That's because for a small input, the time of splitting and assigning of elements to each thread is significant. This forms the extra overhead for the parallel algorithm.
- On increasing the number of elements, the difference between the time taken by both algorithms decreases.
- At one junction of time, we have no time difference between parallel and sequential algorithm.
- With increasing number of elements, we have significant time improvement for parallelly over the sequential one.

On a small number of elements, the sequential algorithm would be the best choice of algorithm since the process would be quicker than the parallel algorithm.

The parallel algorithm would be the best choice if the number of elements is large, because the thread creation time can be compensated for the number of threads which are operating in parallel, and thus, the overhead for thread creation can be saved.