High Performance Computing

Design Activity 1

Roll No: **CED18I042**
Name: **Reuben Skariah Mathew**

Date: **9th September, 2021**

Hardware Configuration:

Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Sockets: 1
Cores per Socket: 4
Threads per Core: 2
L1 Cache: 32 kB
L2 Cache: 256 kB
L3 Cache: 6 MB
RAM: 8 GB

Question 1

Parallelize the following sequence and write openMP and estimate the parallelization fraction for the sequence's larger number 100000. Write your comments on each problem.

Sequence 5, 8, 11, 14, 17, 20, 23...

Approach:

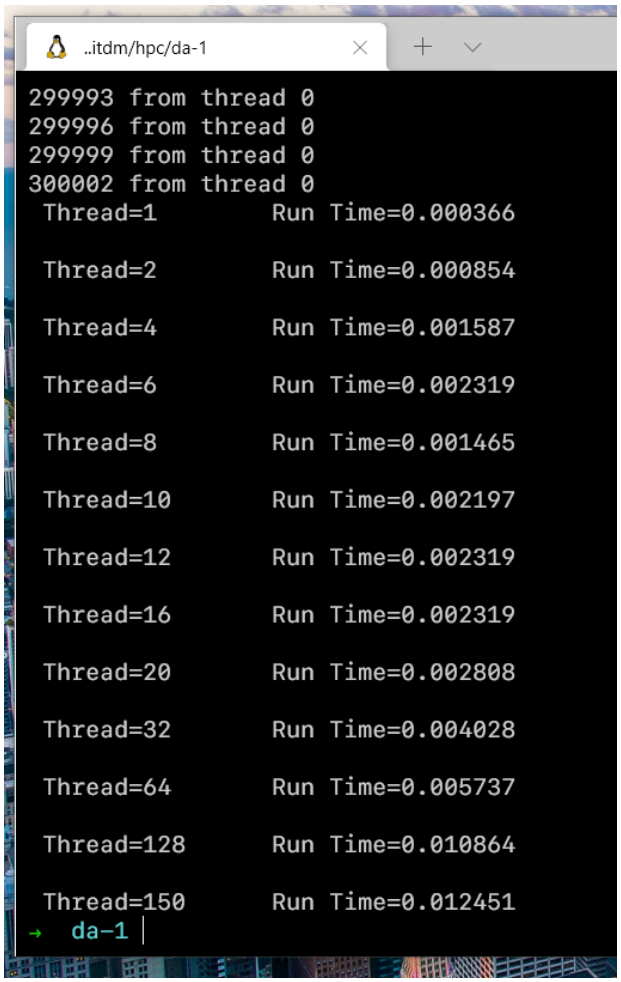
This sequence consecutively adds 3 to 5. This can be generated by the formula 5 + (i + 3) where i is incremented by 1. This is parallelizable and hence it will be placed in the parallel part of the code.

OpenMP Code:

```
#include <stdio.h>
#include <omp.h>
#define n 100000
int main()
{
    int seq[n], num = 0;
    float starttime, endtime, exectime;
    int i, k;
    int omp_rank;
    float etime[20];
    int thread[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64, 128, 150};
    int thread_arr_size = 13;
    for (k = 0; k < thread_arr_size; k++)
    {
        omp_set_num_threads(thread[k]);
        starttime = omp_get_wtime();
```

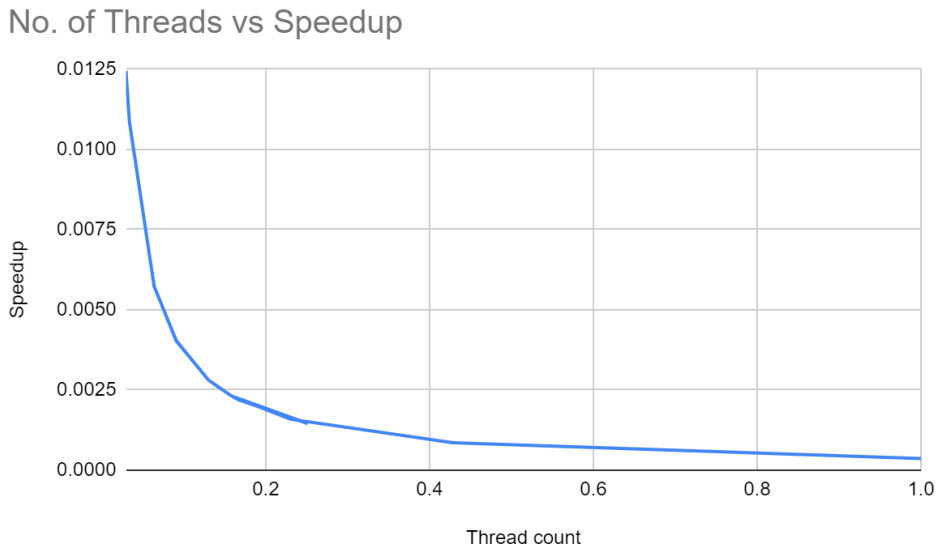
```
#pragma omp parallel private(i) shared(seq, num)
        {
#pragma omp for
            for (i = 0; i < n; i++)
            {
                omp_rank = omp_get_thread_num();
                seq[i] = 5 + (3 * i);
            }
        }
        endtime = omp_get_wtime();
        exectime = endtime - starttime;
        etime[k] = exectime;
        for (i = 0; i < n; i++)
            printf("\n%d from thread %d", seq[i], omp_rank);
    }
    for (k = 0; k < thread_arr_size; k++)
        printf("\n Thread=%d\t Run Time=%f\n", thread[k], etime[k]);

    return 0;
}
```

Output:



Analysis:

Number of Threads	Execution Time	Speed-Up	Parallelization Factor
1	0.000366	1	
2	0.000854	0.4285714286	-266.6666667
4	0.001587	0.2306238185	-444.8087432
6	0.002319	0.1578266494	-640.3278689
8	0.001456	0.2513736264	-340.3590945
10	0.002197	0.1665908056	-555.8591378
12	0.002319	0.1578266494	-582.1162444
16	0.002319	0.1578266494	-569.1803279
20	0.002808	0.1303418803	-702.3295945
32	0.004028	0.09086395233	-1032.82214
64	0.005737	0.06379640927	-1490.779773
128	0.010864	0.0336892489	-2890.891098
150	0.012451	0.0293952293	-3324.073055



Observation:

As this problem is not very computationally intensive the runtime for thread count = 1 is low. The run time increases as thread count increases further.

Question 2

Sequence 1*sin a, 3*sin b, 6*sin c, 10*sin d, 15*sin a, 21*sin b, 28*sin c, 36*sin d, 45*sin a, ..
Where a, b, c, d follows the following sequence: 0, 30, 60 and 90 respectively.

Approach:

This sequence consecutively can be split into two parts: 1, 3, 6, 10… and the sin part. The first part can be generated by the formula i * (i + 1) / 2. And the sin part cycle between 4 values. To improve performance, the 4 sin values can be calculated and stored in an array, then called when needed.

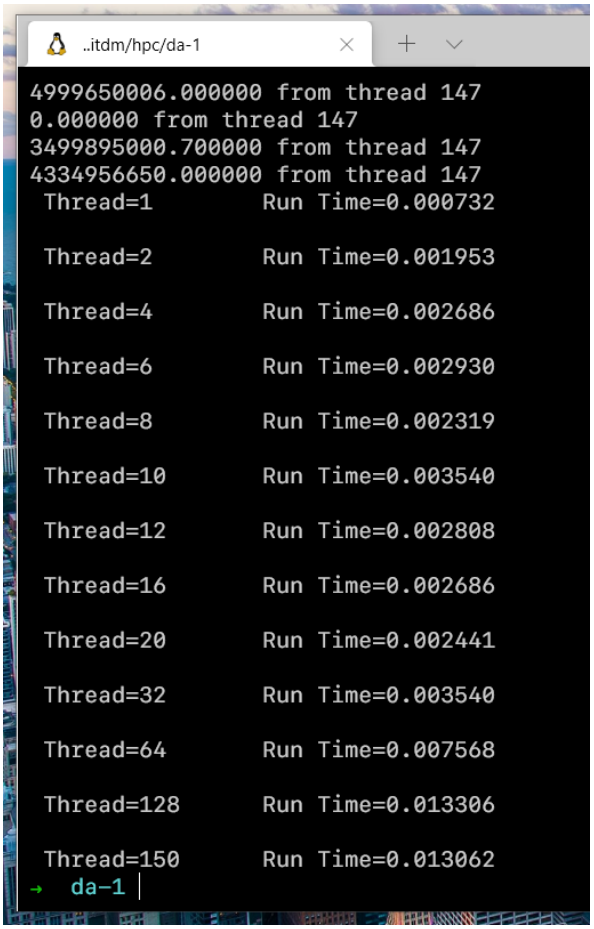
Hence in the parallel section we can print the sequence using the formula ((i * (i + 1.0)) / 2.0) * ang[(i - 1) % 4] where ang[] contains the 4 sin values, and i is incremented by 1.

OpenMP Code:

```
#include <stdio.h>
#include <omp.h>
#define n 100000
int main()
{
    double ang[4] = {0.0, 0.7, 0.867, 1.0};
    double seq[n], num = 0.0;
    float starttime, endtime, exectime;
    int i, k, c = 0;
    int omp_rank;
    float etime[20];
    int thread[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64, 128, 150};
    int thread_arr_size = 13;
    for (k = 0; k < thread_arr_size; k++)
    {
        omp_set_num_threads(thread[k]);
        starttime = omp_get_wtime();
#pragma omp parallel private(i) shared(seq, num)
        {
#pragma omp for
            for (i = 1; i < n; i++)
            {
                omp_rank = omp_get_thread_num();
                num = ((i * (i + 1.0)) / 2.0) * ang[(i - 1) % 4];
                seq[i] = num;
            }
        }
```

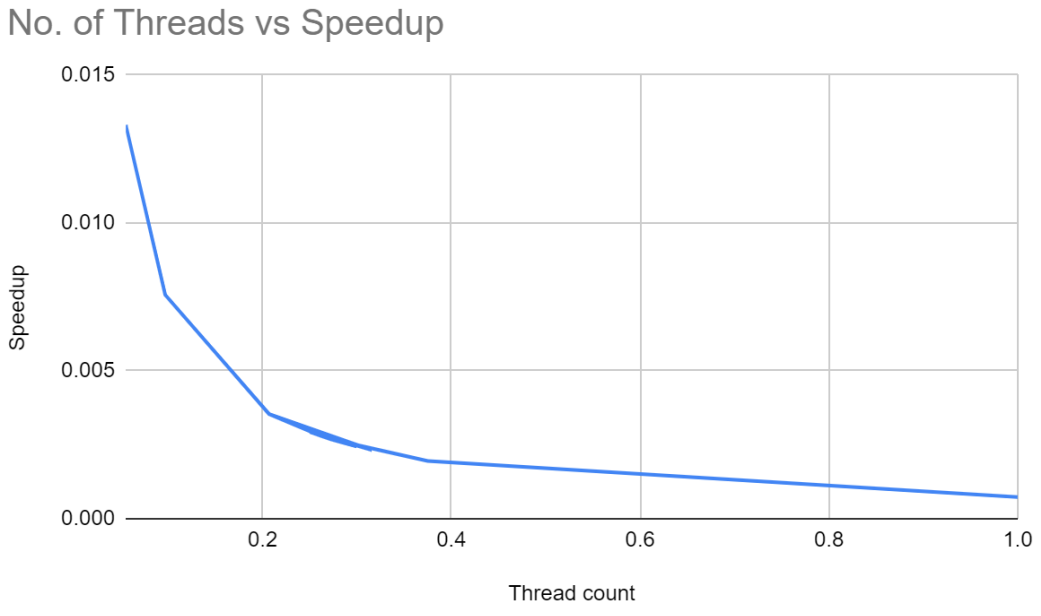
```
        endtime = omp_get_wtime();
        exectime = endtime - starttime;
        etime[k] = exectime;
        for (i = 0; i < n; i++)
            printf("\n%lf from thread %d", seq[i], omp_rank);
    }
    for (k = 0; k < thread_arr_size; k++)
        printf("\n Thread=%d\t Run Time=%f\n", thread[k], etime[k]);

    return 0;
}
```

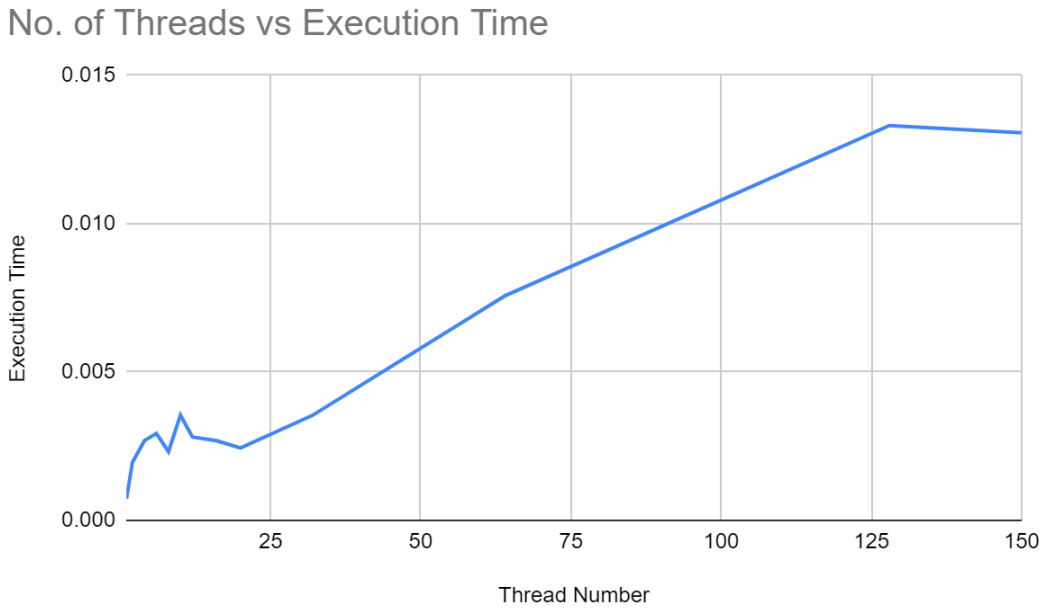
Output:



Analysis

Number of Threads	Execution Time	Speed-Up	Parallelization Factor
1	0.000732	1	
2	0.001953	0.3748079877	-333.6065574
4	0.002686	0.2725241996	-355.9198543
6	0.00293	0.2498293515	-360.3278689
8	0.002319	0.3156532988	-247.7751756
10	0.00354	0.206779661	-426.2295082
12	0.002808	0.2606837607	-309.3889717
16	0.002686	0.2725241996	-284.7358834
20	0.002441	0.2998770995	-245.7578372
32	0.00354	0.206779661	-395.9809625
64	0.007568	0.0967230444	-948.70327
128	0.013306	0.05501277619	-1731.285229
150	0.013062	0.0560404226	-1695.731104





Observation:

In this sequence, similar to the first sequence the run time is minimum for thread count = 1. The run time is increasing as thread count increases. As this problem is also not computationally intensive, the effect of parallelization cannot be visualized properly.

Question 3

Sequence $8, 4\sqrt{2}, 4, 2\sqrt{2}$

Approach:

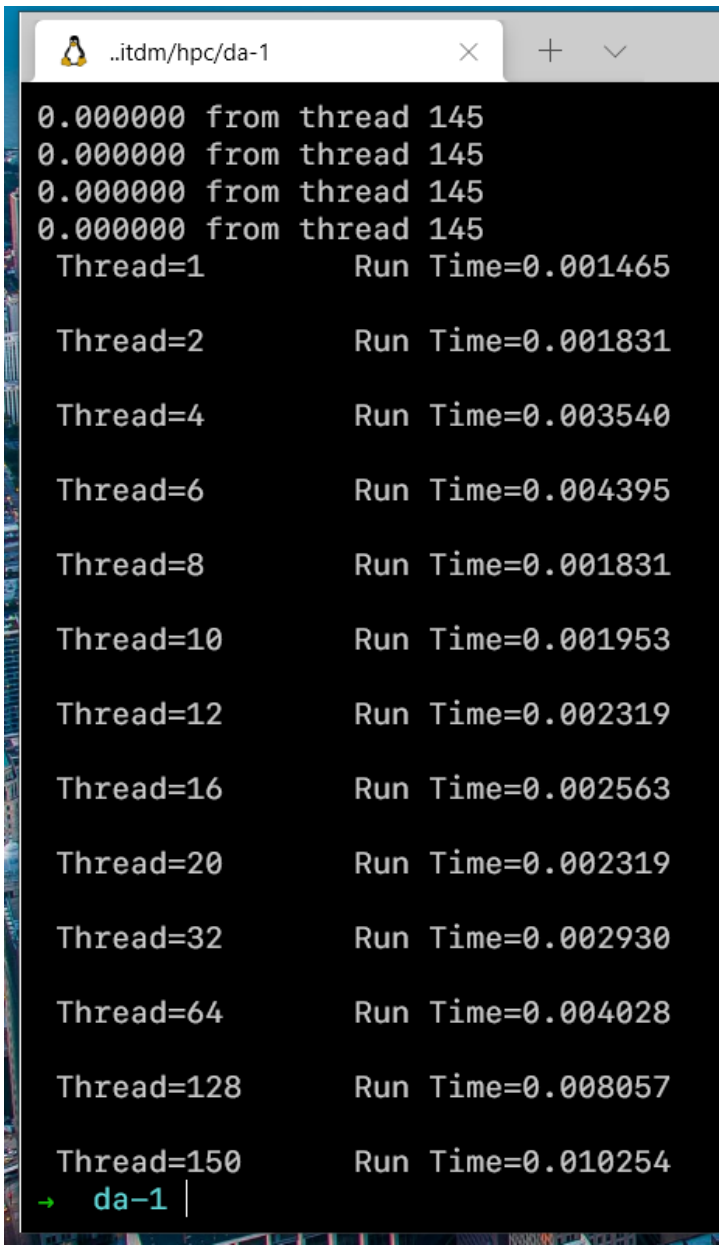
This sequence can be seen as a geometric progression. Where a = 8 and r $= 1\sqrt{2}$. To print a GP the terms are according to the formula a + r^i. Where i increases by 1. This will be in the parallel section: 8 * pow(0.707, i);

OpenMP Code:

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
#define n 100000

int main()
{
    double r = 0.707;
    double seq[n];
    double p, num;
    float starttime, endtime, exectime;
    int i, k;
    int omp_rank;
    float etime[20];
    int thread[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64, 128, 150};
    int thread_arr_size = 13;
    for (k = 0; k < thread_arr_size; k++)
    {
        omp_set_num_threads(thread[k]);
        starttime = omp_get_wtime();
#pragma omp parallel shared(seq, i)
        {
#pragma omp for
            for (i = 0; i < n; i++)
            {
                omp_rank = omp_get_thread_num();
                seq[i] = 8 * pow(0.707, i);
            }
        }
        endtime = omp_get_wtime();
        exectime = endtime - starttime;
        etime[k] = exectime;
        for (i = 0; i < n; i++)
            printf("\n%lf from thread %d", seq[i], omp_rank);
    }
```

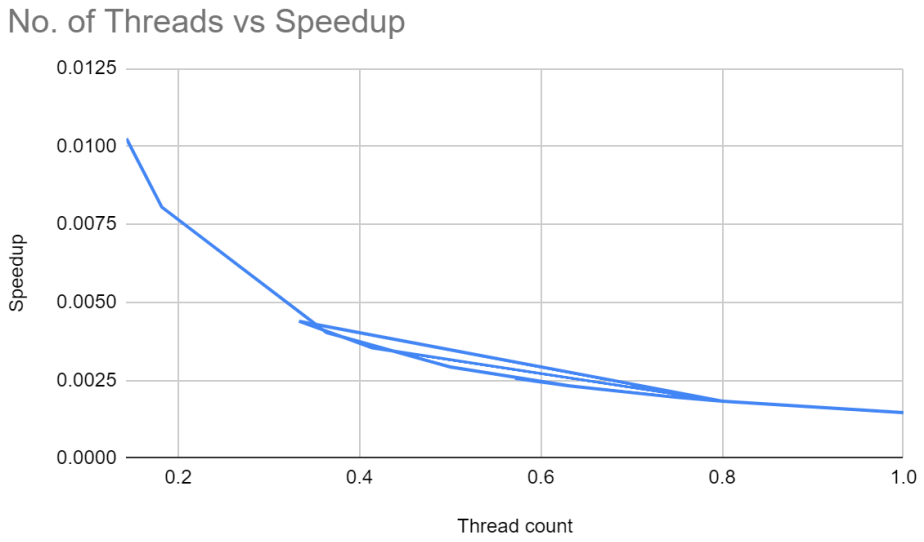
```
    for (k = 0; k < thread_arr_size; k++)
        printf("\n Thread=%d\t Run Time=%f\n", thread[k], etime[k]);

    return 0;
}
```

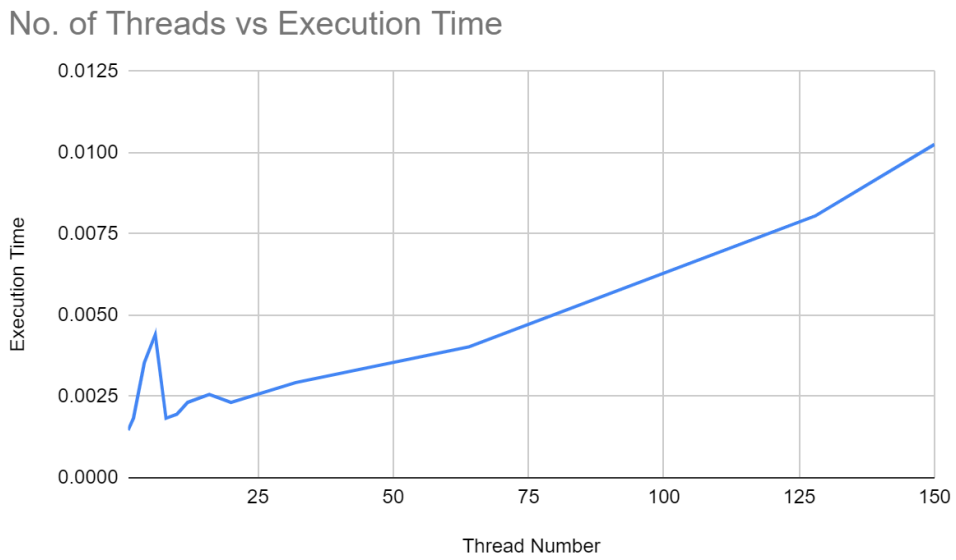
Output:



Analysis:

Number of Threads	Execution Time	Speed-Up	Parallelization Factor
1	0.001465	1	
2	0.001831	0.8001092299	-49.96587031
4	0.00354	0.4138418079	-188.850967
6	0.004395	0.3333333333	-240
8	0.001831	0.8001092299	-28.55192589
10	0.001953	0.7501280082	-37.01175578
12	0.002319	0.631737818	-63.59292585
16	0.002563	0.5715957862	-79.94539249
20	0.002319	0.631737818	-61.36159511
32	0.00293	0.5	-103.2258065
64	0.004028	0.3637040715	-177.7257706
128	0.008057	0.1818294651	-453.5089087
150	0.010254	0.1428710747	-603.9581281





Observations:

For this sequence the run time is minimum at thread count = 1. But since this is more computationally intensive due to the power calculation, we see a dip in run time at thread count = 8, as this is the maximum number of threads supported by my system. On increasing the thread count beyond 8 the run time keeps increasing.