1. (70 points) You are required to develop a parallel C/C++ implementation of the above algorithm that uses OpenMP directives to parallelize the `compute_inverse` routine. You should use the task directive for the recursive tasks.

This is done as it can be found in cpp file submitted. The result has also been compared with Matlab to make sure it is accurate.

2. (10 points) Describe your strategy to parallelize the algorithm. Discuss any design choices you made to improve the parallel performance of the code.

1. Instead of using 2-dimensional arrays I have used 1-dimensional arrays and use them as if they are 2-dimensional arrays. This is helpful for few reasons, first the same code can be used for different array sizes without a need for template, two it allows quicker index computation (assuming the compiler is not very smart, which may not be the cases depending on the compiler).
2. Making sure nothing gets copied from one place to another. This is important because in the first glance it may seem like a good idea to copy different portion of the matrix and send them to the functions that take care of the rest. But this the program ends up having a significantly larger memory footprint specially because it is recursively making these copies.

3. All functions are designed to read or write to a portion of matrix without disturbing the other portions. The main challenge here is assurance that you do not go out of bound. This is especially important because this algorithm in nature does not create a race condition.
    - Using computeInverse function that only work with a specific portion

```
// startingI: starting I of the interested area
// startingJ: starting J of the interested area
// n: the N for area of interest
// mainMatrixN: this is the main actual matrix N,
// since I am not using a 2-dimensional array this is needed to make sure I index correctly.
bool computeInverseParallel(const double matrix[], double inverseMatrix[],
    const uint64_t startingI, const uint64_t startingJ, const uint64_t n,
    const uint64_t mainMatrixN)
```

    - For multiplication since there are 3 matrices involved, matrix A, matrix B, matrix containing the result. This means having many arguments resulting too much memory duplication. So a struct is used for each matrix and they are passed by reference to reduce the memory footprint and since they are const the compiler can optimize them to be read once and only the parts needed which could end up being more efficient than duplicating them all (could not produce evidence proving this, since the speed up is so little that (if any assuming the compiler does not do the same thing) that it is in the margin of effort in my measurement).

```
struct MatrixDataForMultiplication
{
    // startingI: starting I of the interested area
    // startingJ: starting J of the interested area
    // iSize: the number of rows for area of interest
    // jSize: the number of columns  for area of interest
    // matrixJSize: this is the main actual matrix N number of columns,
    // since I am not using a 2-dimensional array this is needed to make sure I index correctly.
    uint64_t startingI, startingJ, iSize, jSize, matrixJSize;
};
```

```
void multiplyMatrixParallel(const double matrixA[], const MatrixDataForMultiplication &matrixAData,
    const double matrixB[], const MatrixDataForMultiplication &matrixBData,
    double result[], const MatrixDataForMultiplication &resultData, bool isPositive)
```

4. For multiplication the equation goes as $-R_{11}^{-1}\, R_{12}\, R_{22}^{-1}$ and in order to not to multiply by -1 again after the 2 multiplication, I left a bool in my multiplication function to take care of that and just do a sign flip once the calculation is completed.

```
if (!isPositive)
{
    result[resultIndex] = -result[resultIndex];
}
```

5. For parallelizing the computation of R_{11}^{-1} and R_{22}^{-1} I used task. The task was called from a parallel region created when calling the function. This is so that the code can be ran both parallel or serial based on how it is called, also to change the parallel region setting from one place. Inverse matrix is shared because they all write into it, but since it is in different locations it does not need any lock. Based on Oracle documentation [1] if variables are shared between arrays to only read from, it is best to put it as firstprivate to reduce overhead. Default is set to none to make sure all variables are taken care of explicitly.

```
#pragma omp task shared(inverseMatrix) firstprivate(matrix, startingI, startingJ, newN, mainMatrixN) default(none)
{
    //printf("1 T_ID: %d\n", omp_get_thread_num());
    computeInverseParallel(matrix, inverseMatrix, startingI, startingJ, newN, mainMatrixN);
}
#pragma omp task shared(inverseMatrix) firstprivate(matrix, startingI, startingJ, n, newN, mainMatrixN) default(none)
{
    // printf("2 T_ID: %d\n", omp_get_thread_num());
    computeInverseParallel(matrix, inverseMatrix, startingI + newN, startingJ + newN, n - newN, mainMatrixN);
}
```

6. This is how the main function is called in parallel region, with a single to make sure only it gets called once. Event though myMatrix is only read by different threads but if it is called as firstPrivate, other threads won't be able to access it correctly.

```
#pragma omp parallel shared(myMatrix, inverseMatrix, n, startingI, startingJ)
{
    #pragma omp single
    {
        computeInverseParallel(myMatrix, inverseMatrix, startingI, startingJ, n, n);
    }
}
```

7. For the multiplication there are 3 nested loops, but the best performance is by parallelizing only the first 2. In fact, the process is so computational heavy that even parallelizing the first loop alone would produce very similar numbers to having the first two loops. This is because the cores are almost 100% utilized so it won't make too much difference. So I have paralyzed the first 2 loops in case there are enough cores, so that each core can compute each element in the result matrix. Moreover, since the tasks are smaller, the result matrix gets computed faster, if one core for some reason is not running as fast as the others due to CPU clock, temperature or other issues. The loop is nowait since each element is computed independently. Variables are defined private so each thread can have its own version.

```
uint64_t i, j, k;
uint64_t matrixAIndex, matrixBIndex, resultIndex;
#pragma omp for nowait collapse(2) private(i, j, k, matrixAIndex, matrixBIndex, resultIndex)
    for (i = 0; i < matrixAData.iSize; i++)
    {
        for (j = 0; j < matrixBData.jSize; j++)
        {
            matrixAIndex = matrixAData.startingJ + (matrixAData.startingI + i) * matrixAData.matrixJSize;
            matrixBIndex = matrixBData.startingJ + j + matrixBData.startingI * matrixBData.matrixJSize;
            resultIndex = resultData.startingJ + j + (resultData.startingI + i) * resultData.matrixJSize;
            result[resultIndex] = 0;
            // printf("[%lu, %lu] =>", i, j);
            for (k = 0; k < matrixAData.jSize; k++)
            {
                // printf("%.2lf X %.2lf + ", matrixA[matrixAIndex], matrixB[matrixBIndex]);
                result[resultIndex] += matrixA[matrixAIndex] * matrixB[matrixBIndex];
                matrixAIndex++;
                matrixBIndex += matrixBData.matrixJSize;
            }

            if (!isPositive)
            {
                result[resultIndex] = -result[resultIndex];
            }
            // printf("= %.2lf \n", result[resultIndex]);
        }
    }
```

3. (20 points) Determine the speedup and efficiency obtained by your routine on 1, 2, 4, 10, and 20 processors. You may choose appropriate values for the matrix size to illustrate the features of your implementation.

For number of processors:

Unlike the previous time using Open MPI the Open MP is efficiency and speed up result are not straight forward. This is due to few reasons. First the core clocks are not consistent, and this can be observed by running the same code on the same machine at different times or even consecutively. The variations are sometimes more than double in consecutive runs. Since the one thread performance is the base for calculating the efficiency the numbers can be more ideal rather easily. This happens because tasks take a longer time on a single thread and if the clock drops the whole process become slower. In contrast in multiple core environment the cores have different clocks, so they achieve a more stable average because if one is slower and one is faster, they cover up for each other to certain extend. Another reason is the overhead of creating more threads. Open MP creates many threads and in run time those threads go to available thread. As a result, the many threads overhead reduce the single thread performance because the program is not single thread by design to eliminate the thread overhead. In conclusion these two reason result in single thread timing not being a correct representation of a single thread performance and as a result the efficiency and speed ups do not follow their theoretical means.

When the job does not result in many threads for example for n = 100, the time taken for the task fluctuates significantly, but the theoretical trend is easier to observer because there is less overhead all the threads created.

For N:

Based on my result below N = 2000, the best performing Ns are the ones that have the smaller serial inverse operation. For instance, for 2000, it ends up with 7s and 8s which are the smallest N to call the serial part because for any n/2 equal to 15 we call the serial version. 15/2 = 7.5 so 7 and 8 are the lowest N that goes to serial part. It is important to note that the serial parts are not called only twice. This because all mXm (m being the size that serial function is called with) matrices which share their diagonal with the main matrix diagonal will generate a call to serial function. (for example, if n=10 and m=2, the serial function will be called 10/2 = 5 times)

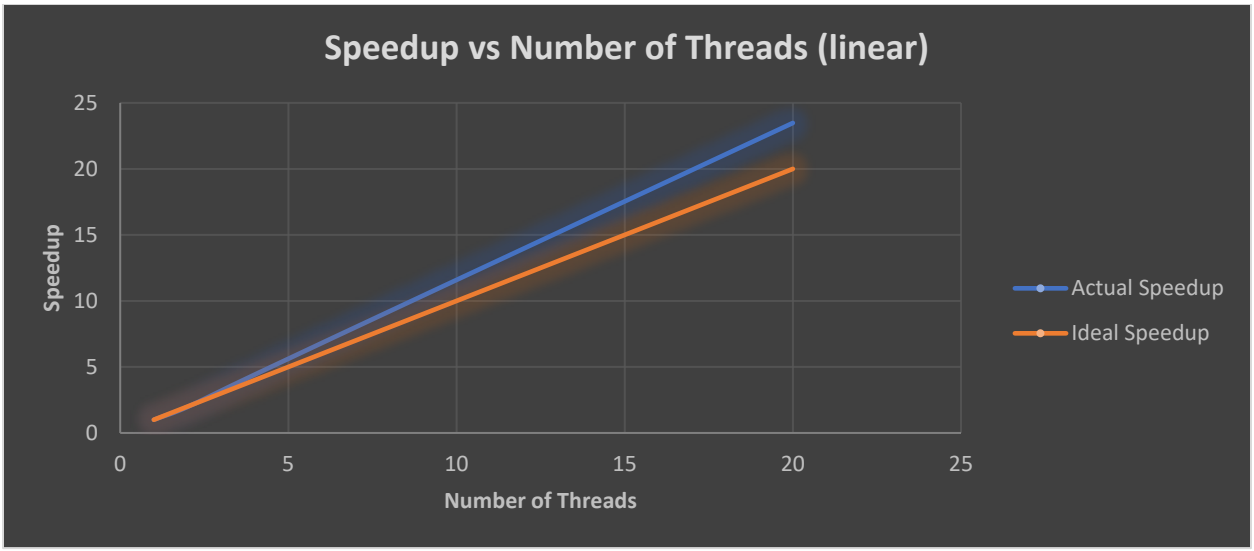
$$2000 \rightarrow\ 1000 \rightarrow\ 500 \rightarrow\ 250 \rightarrow\ 125 \rightarrow \left\{ \begin{array}{l} 63\ \rightarrow \left\{ \begin{array}{l} 32 \rightarrow 16 \rightarrow 8 \\ 31 \rightarrow \left\{ \begin{array}{l} 16 \rightarrow 8 \\ 15 \rightarrow \{8\} \\ \qquad\ \{7\} \end{array} \right. \end{array} \right. \\ 62 \rightarrow 31 \rightarrow \left\{ \begin{array}{l} 16 \rightarrow 8 \\ 15 \rightarrow \{8\} \\ \qquad\ \{7\} \end{array} \right. \end{array} \right.$$

Take note that 1920 and 2048 take significantly more time per n^3. We should use n^3 since this problem is O(N^3) complexity, so dividing by n does not make sense. The rest of table seems to be highly affected by core clock and the inner complexity of threads and how things change as more thread gets created. The complete table is included at the last page.

Alireza Safdari Ghandhari
UIN: 728009403

N = 4048

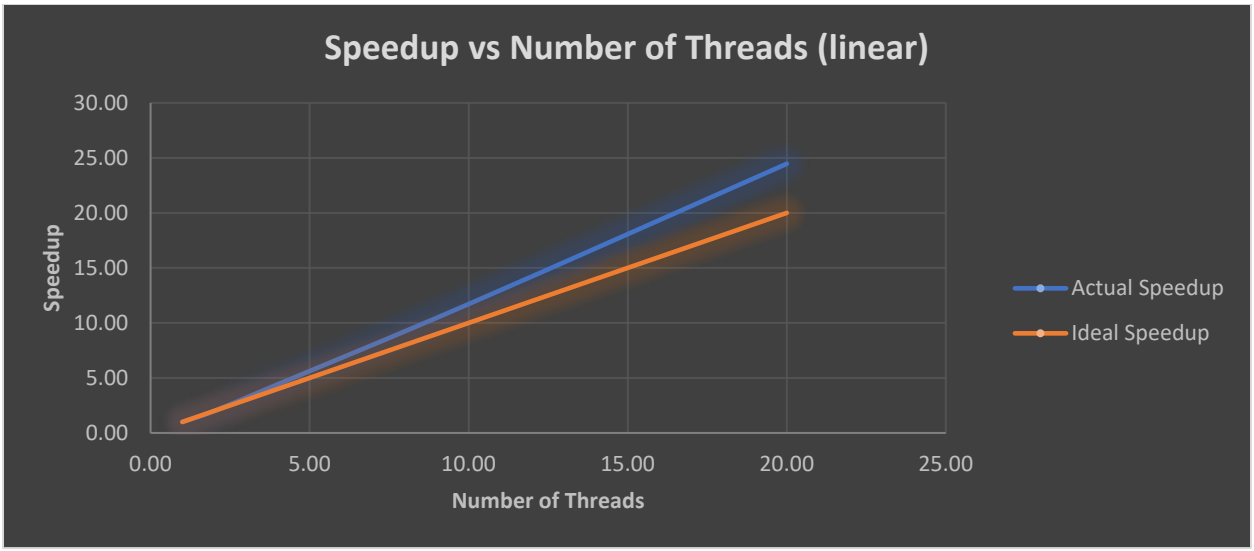
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1	63.79	64.49	63.92	64.06	65.45	64.34	1.00	1.00	1.00
2	32.60	32.69	32.68	32.68	32.72	32.67	1.97	2.00	0.98
4	14.54	14.53	14.54	14.53	14.66	14.56	4.42	4.00	1.10
10	5.52	5.56	5.55	5.54	5.59	5.55	11.59	10.00	1.16
20	2.71	2.72	2.81	2.74	2.72	2.74	23.48	20.00	1.17



Alireza Safdari Ghandhari
UIN: 728009403
Homework2
CSCE-735
10/06/2020

N = 4000

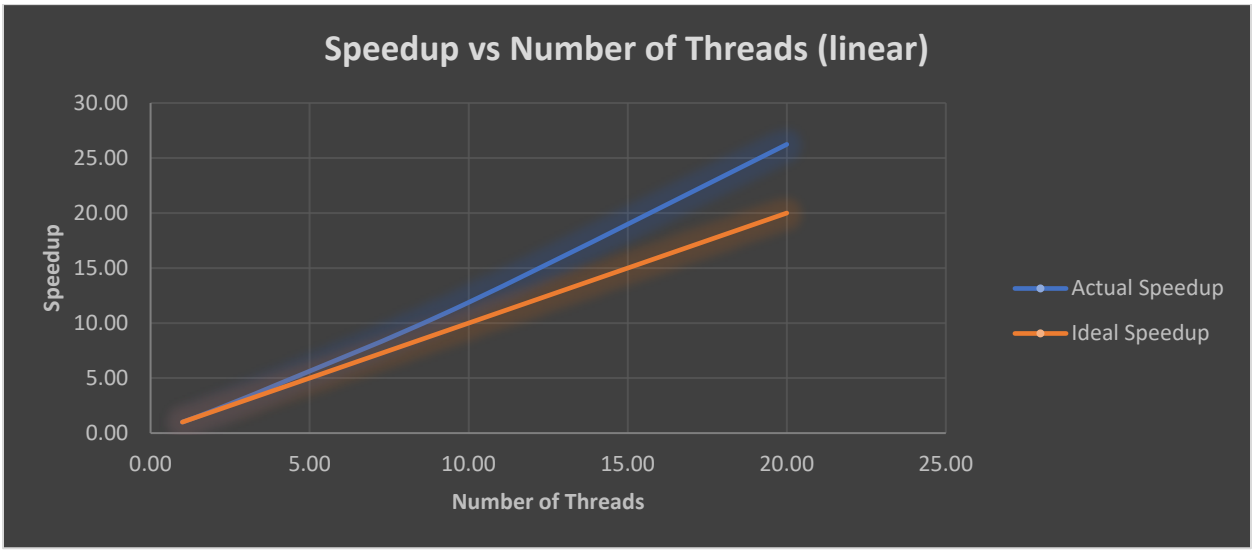
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1.00	68.27	68.18	67.99	68.06	67.56	68.01	1.00	1.00	1.00
2.00	34.03	33.95	33.96	33.99	34.11	34.01	2.00	2.00	1.00
4.00	15.58	15.47	15.28	15.31	15.27	15.38	4.42	4.00	1.11
10.00	5.81	5.79	5.88	5.76	5.79	5.81	11.71	10.00	1.17
20.00	2.80	2.80	2.75	2.83	2.72	2.78	24.47	20.00	1.22



Alireza Safdari Ghandhari
UIN: 728009403

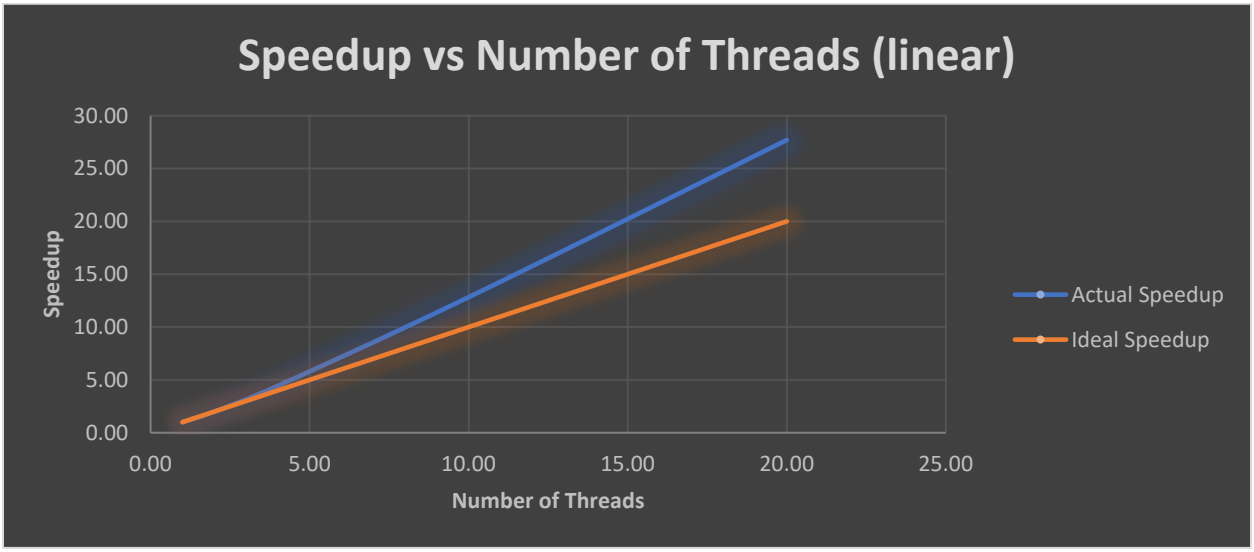
N = 3840

Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	63.62	63.90	63.67	63.48	63.50	63.64	1.00	1.00	1.00
2.00	30.47	30.51	30.55	30.63	30.56	30.55	2.08	2.00	1.04
4.00	14.28	14.29	14.44	14.40	14.30	14.34	4.44	4.00	1.11
10.00	5.60	5.24	5.38	5.25	5.32	5.36	11.88	10.00	1.19
20.00	2.61	2.37	2.38	2.38	2.38	2.43	26.24	20.00	1.31



N = 2048

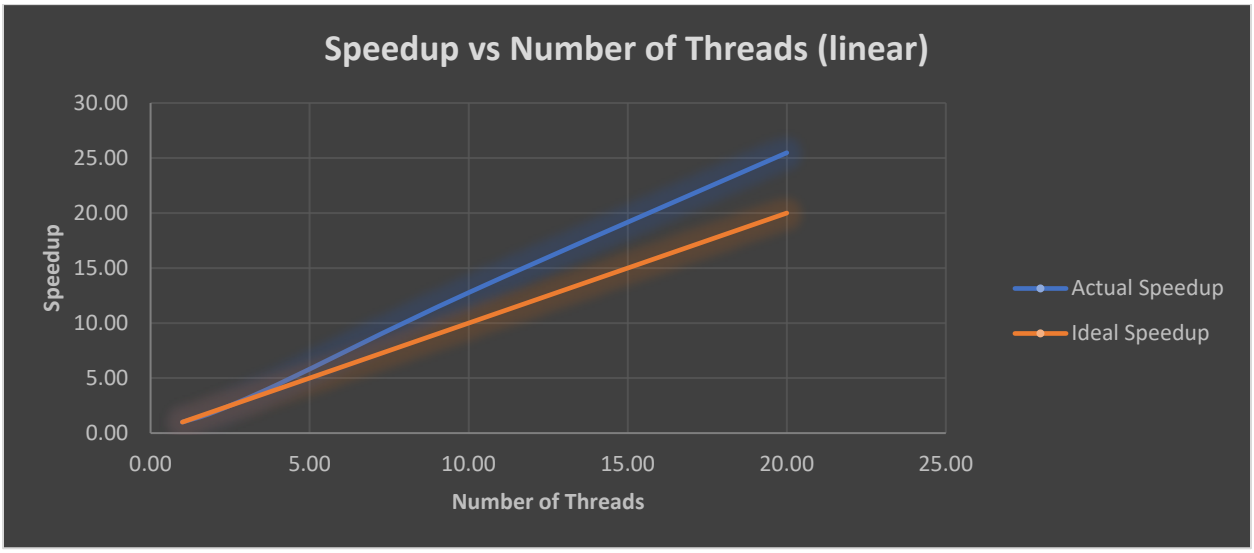
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	9.05	9.00	8.87	9.03	8.96	8.98	1.00	1.00	1.00
2.00	4.53	4.43	4.56	4.47	4.53	4.50	1.99	2.00	1.00
4.00	2.01	2.04	2.01	2.02	2.03	2.02	4.44	4.00	1.11
10.00	0.73	0.74	0.73	0.64	0.66	0.70	12.83	10.00	1.28
20.00	0.31	0.31	0.31	0.39	0.31	0.32	27.70	20.00	1.38



Alireza Safdari Ghandhari
UIN: 728009403

N = 2000

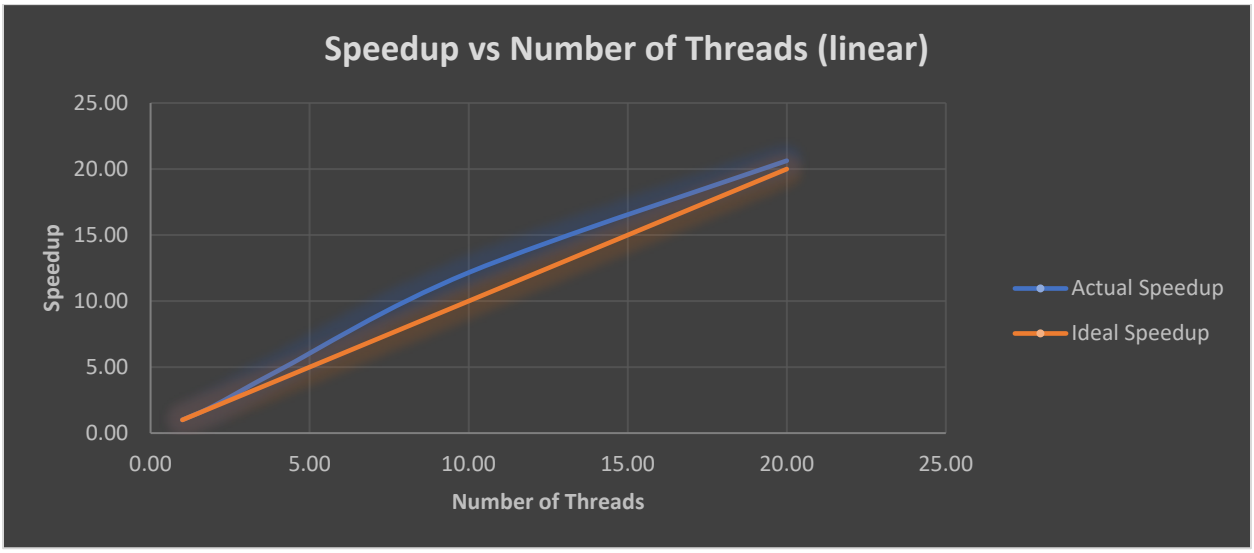
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	3.81	3.88	3.88	3.68	3.76	3.80	1.00	1.00	1.00
2.00	1.92	2.07	2.02	2.08	1.82	1.98	1.92	2.00	0.96
4.00	0.86	0.86	0.85	0.86	0.86	0.86	4.42	4.00	1.11
10.00	0.30	0.30	0.30	0.30	0.29	0.30	12.77	10.00	1.28
20.00	0.15	0.15	0.16	0.15	0.15	0.15	25.48	20.00	1.27



Alireza Safdari Ghandhari
UIN: 728009403

N = 1920

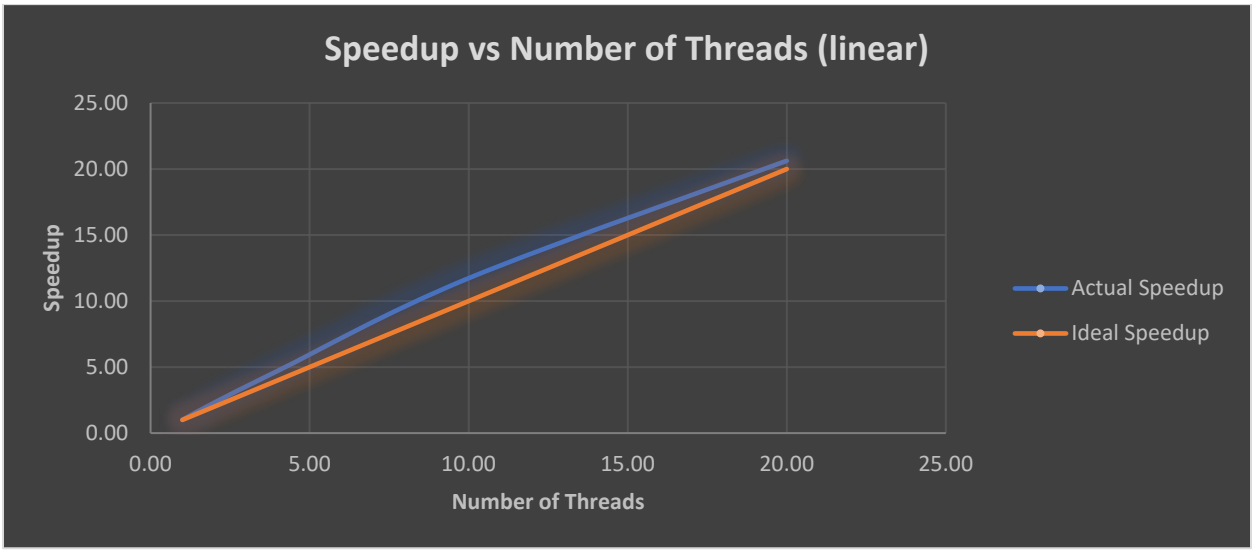
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	5.34	5.33	5.21	5.37	5.29	5.31	1.00	1.00	1.00
2.00	2.58	2.59	2.62	2.55	2.43	2.55	2.08	2.00	1.04
4.00	1.12	1.12	1.12	1.14	1.13	1.13	4.71	4.00	1.18
10.00	0.44	0.43	0.43	0.44	0.44	0.44	12.16	10.00	1.22
20.00	0.23	0.40	0.21	0.23	0.21	0.26	20.63	20.00	1.03



Alireza Safdari Ghandhari
UIN: 728009403

N = 1024

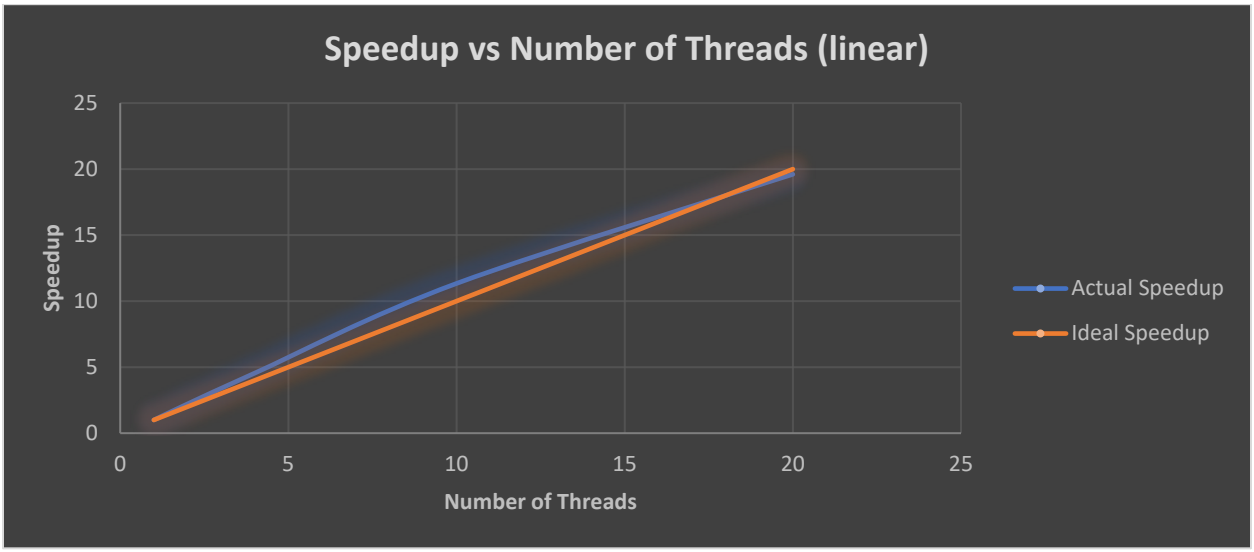
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	0.91	0.91	0.91	0.91	0.91	0.91	1.00	1.00	1.00
2.00	0.40	0.40	0.39	0.39	0.39	0.39	2.31	2.00	1.15
4.00	0.19	0.19	0.19	0.19	0.19	0.19	4.72	4.00	1.18
10.00	0.08	0.08	0.08	0.08	0.08	0.08	11.73	10.00	1.17
20.00	0.04	0.04	0.04	0.06	0.04	0.04	20.63	20.00	1.03



Alireza Safdari Ghandhari
UIN: 728009403
Homework2
CSCE-735
10/06/2020

N = 1000

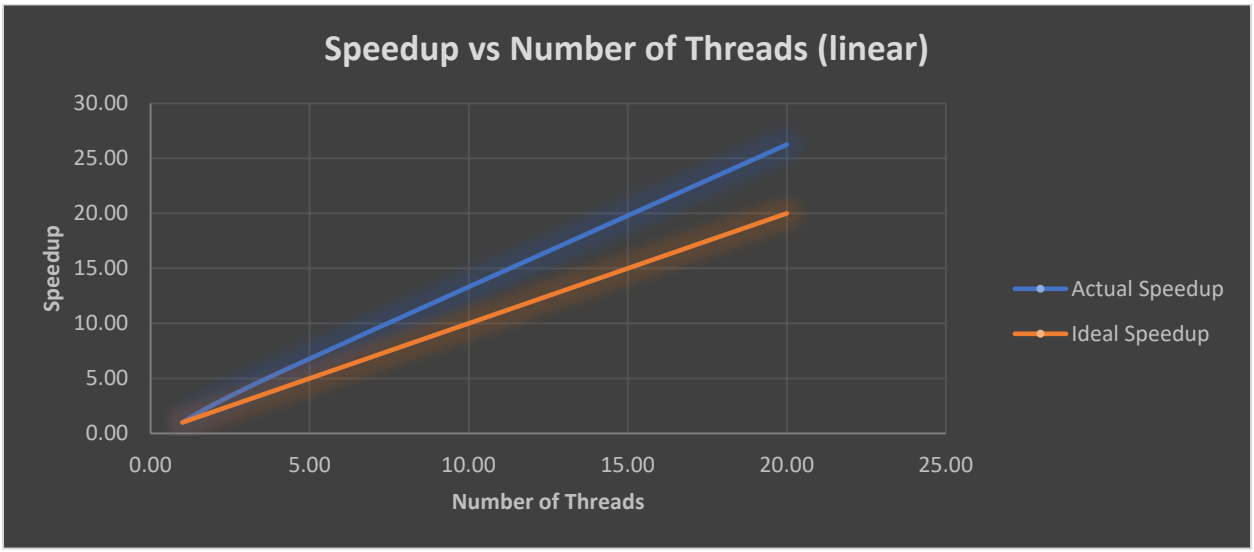
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speed up	Ideal Speedup	Efficiency
1	0.398793	0.39768	0.397844	0.396059	0.398238	0.3977228	1	1	1
2	0.180056	0.180469	0.180446	0.17936	0.177555	0.1795772	2.214773	2	1.10738668
4	0.088169	0.08818	0.086869	0.087052	0.086868	0.0874276	4.549168	4	1.13729188
10	0.035182	0.035162	0.035018	0.034983	0.035038	0.0350766	11.33869	10	1.1338693
20	0.017962	0.017649	0.030584	0.017527	0.017701	0.0202846	19.60713	20	0.98035653



Alireza Safdari Ghandhari
UIN: 728009403

N = 960

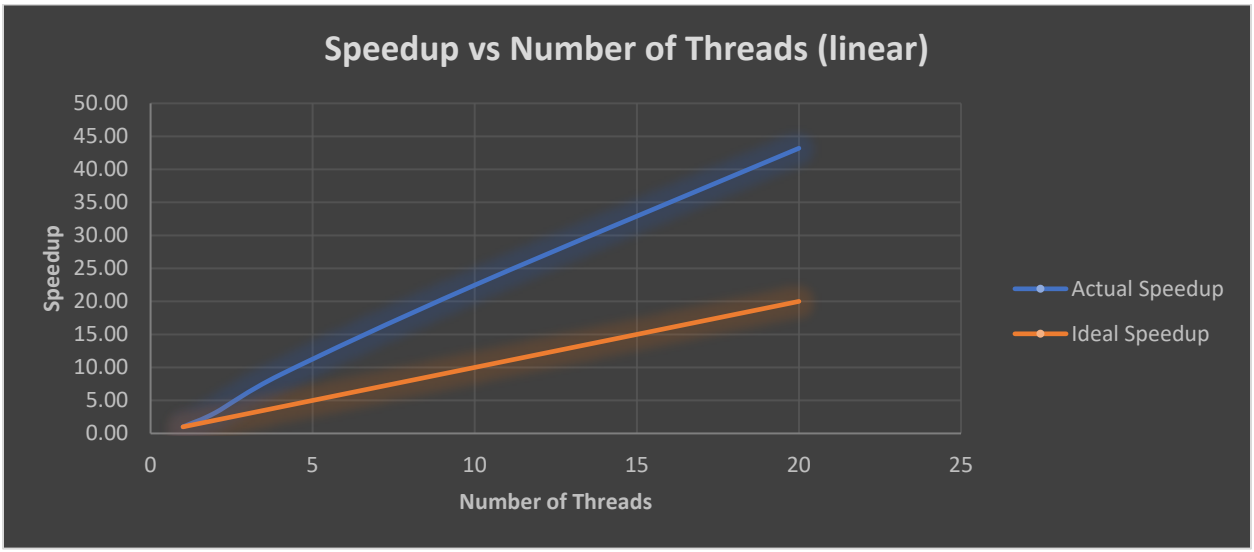
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1.00	0.82	0.45	0.45	0.45	0.45	0.53	1.00	1.00	1.00
2.00	0.20	0.20	0.20	0.20	0.20	0.20	2.66	2.00	1.33
4.00	0.10	0.10	0.10	0.10	0.10	0.10	5.47	4.00	1.37
10.00	0.04	0.04	0.04	0.04	0.04	0.04	13.32	10.00	1.33
20.00	0.02	0.02	0.02	0.02	0.02	0.02	26.25	20.00	1.31



Alireza Safdari Ghandhari
UIN: 728009403

N = 460

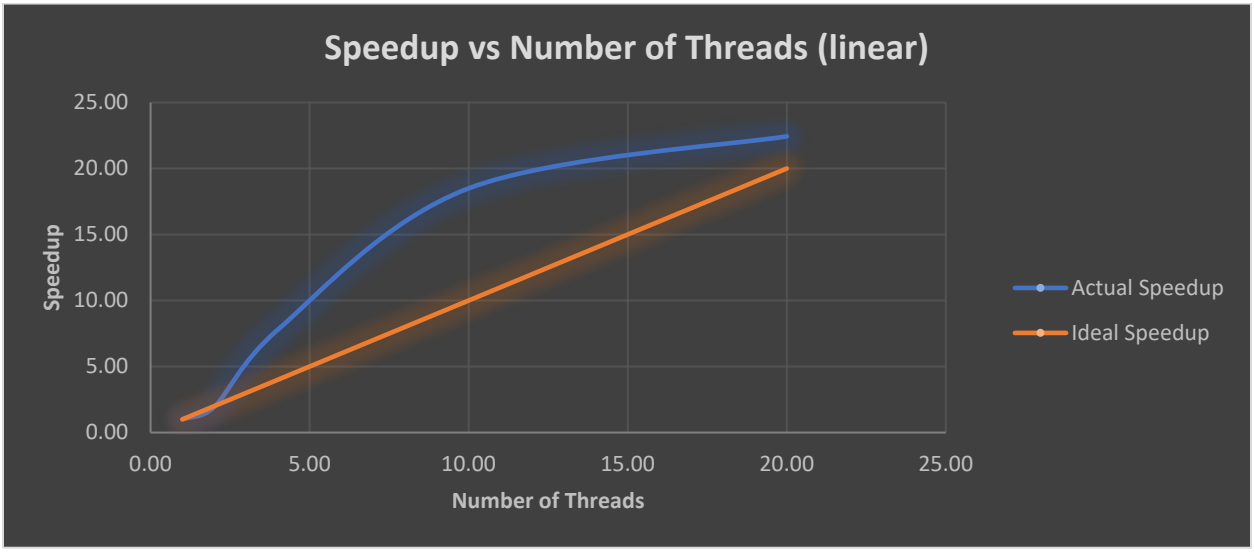
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Spee dup	Ideal Speedup	Efficien cy
1	0.100	0.100	0.100	0.100	0.100	0.100	1.00	1.00	1.00
2	0.046	0.046	0.022	0.022	0.022	0.032	3.15	2.00	1.57
4	0.011	0.011	0.011	0.011	0.011	0.011	8.89	4.00	2.22
10	0.005	0.004	0.004	0.004	0.004	0.004	22.47	10.00	2.25
20	0.002	0.002	0.002	0.002	0.002	0.002	43.20	20.00	2.16



Alireza Safdari Ghandhari
UIN: 728009403

N = 240

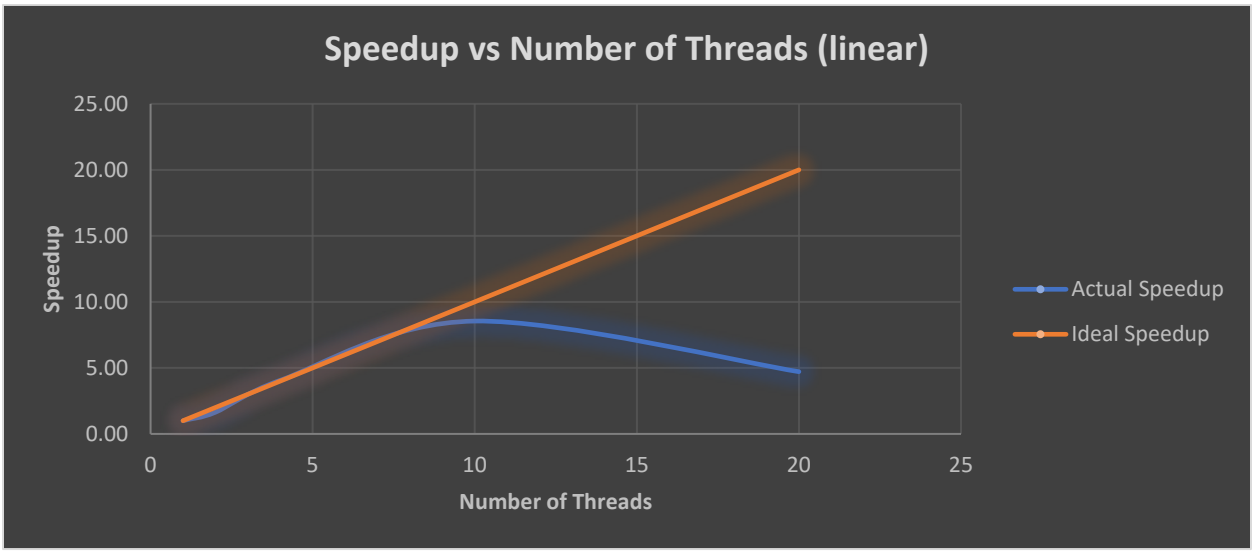
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1.00	0.0049	0.0149	0.0048	0.0149	0.0048	0.0089	1.00	1.00	1.00
2.00	0.0040	0.0047	0.0047	0.0048	0.0048	0.0046	1.93	2.00	0.97
4.00	0.0011	0.0011	0.0012	0.0011	0.0012	0.0011	7.79	4.00	1.95
10.00	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	18.50	10.00	1.85
20.00	0.0005	0.0003	0.0003	0.0004	0.0004	0.0004	22.44	20.00	1.12



Alireza Safdari Ghandhari
UIN: 728009403

N = 128

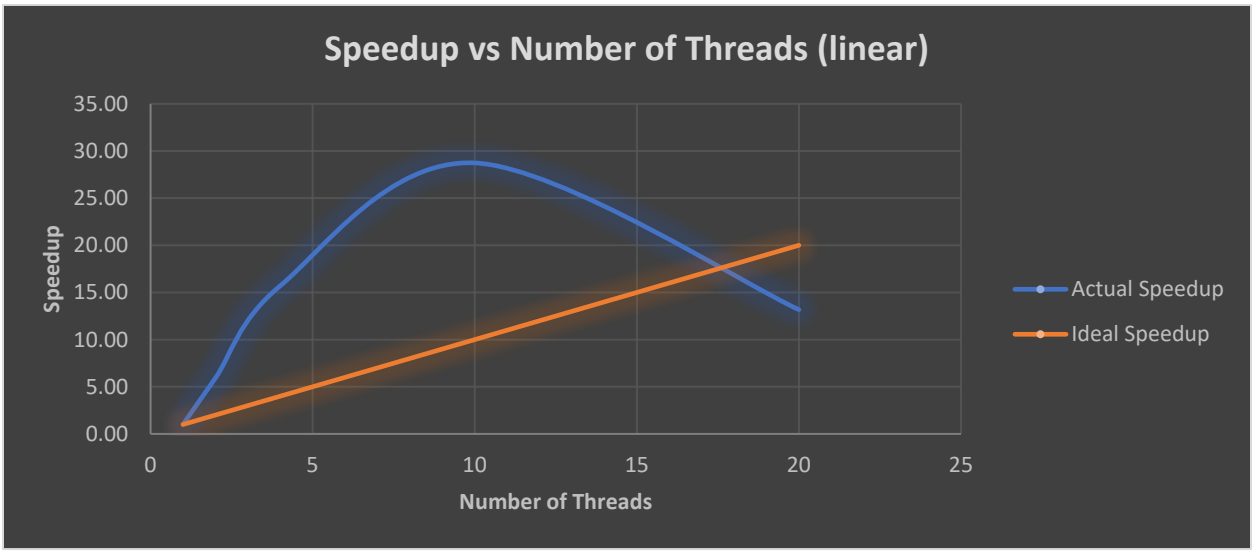
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1	0.0011	0.0011	0.0011	0.0011	0.0011	0.0011	1.00	1.00	1.00
2	0.0007	0.0007	0.0007	0.0007	0.0007	0.0007	1.64	2.00	0.82
4	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	4.06	4.00	1.01
10	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	8.55	10.00	0.85
20	0.0002	0.0003	0.0003	0.0002	0.0002	0.0002	4.71	20.00	0.24



Alireza Safdari Ghandhari
UIN: 728009403
Homework2
CSCE-735
10/06/2020

N = 120

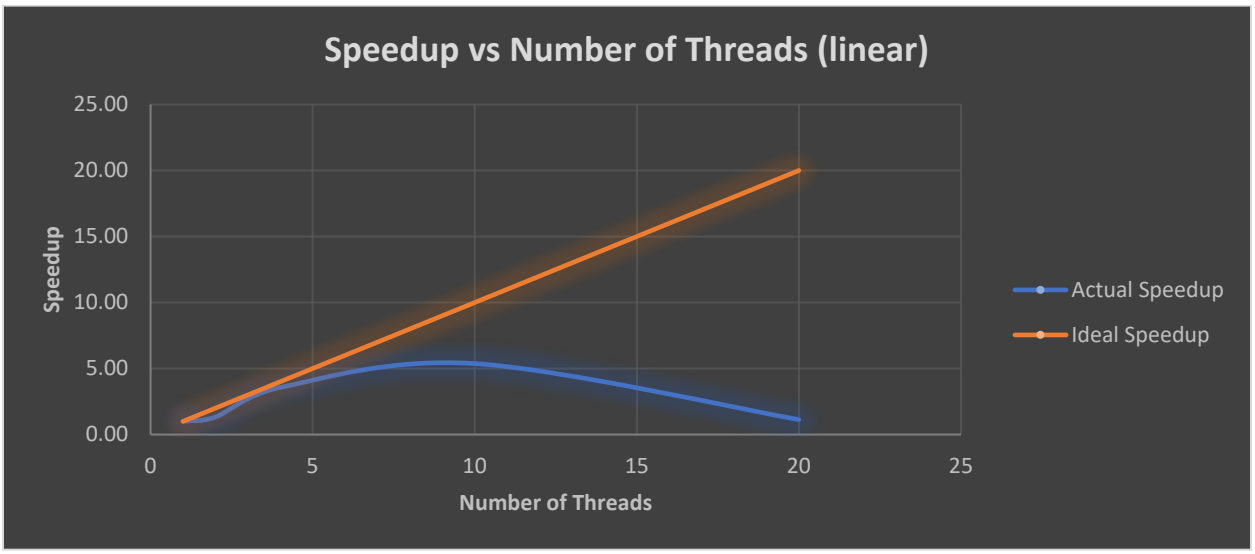
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1	0.0107	0.0007	0.0007	0.0007	0.0007	0.0027	1.00	1.00	1.00
2	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	5.92	2.00	2.96
4	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	15.71	4.00	3.93
10	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	28.74	10.00	2.87
20	0.0003	0.0002	0.0002	0.0002	0.0002	0.0002	13.17	20.00	0.66



Alireza Safdari Ghandhari
UIN: 728009403

N = 100

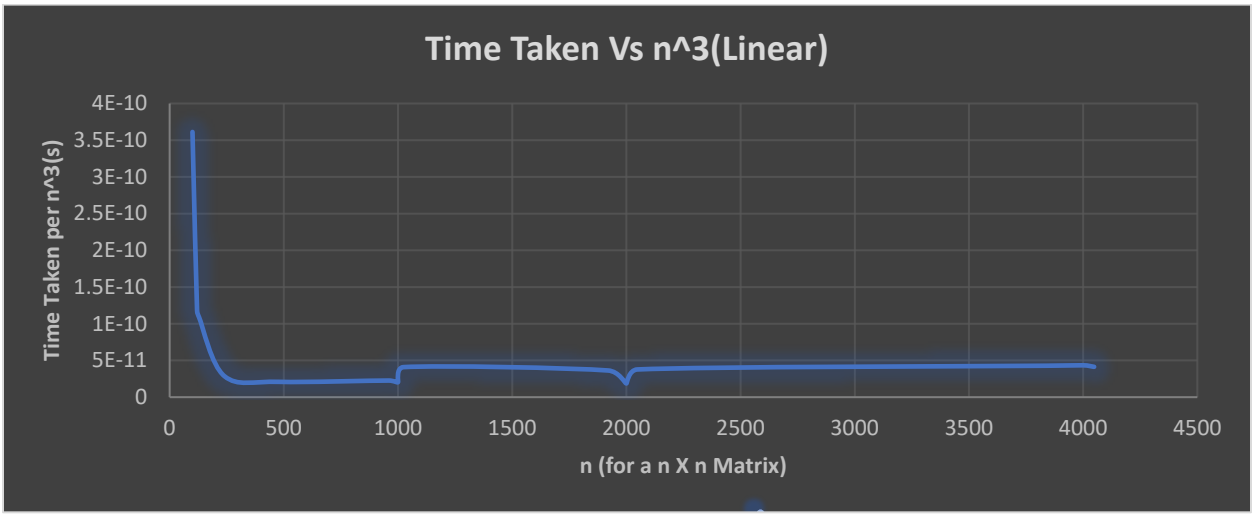
Number of Threads	Test 1 (s)	Test 2 (s)	Test 3 (s)	Test 4 (s)	Test 5 (s)	Average Time (s)	Speedup	Ideal Speedup	Efficiency
1	0.0004	0.0004	0.0004	0.0004	0.0004	0.0004	1.00	1.00	1.00
2	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	1.33	2.00	0.66
4	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	3.57	4.00	0.89
10	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	5.38	10.00	0.54
20	0.0006	0.0002	0.0007	0.0002	0.0002	0.0004	1.13	20.00	0.06



Alireza Safdari Ghandhari
UIN: 728009403

Which value of N is best?

N	Average Time (s)	Cost per N	Cost per N^2	Cost per N^3
100	0.000361	0.000003612	3.612E-08	3.612E-10
120	0.000205	0.000001705	1.42083E-08	1.18403E-10
128	0.000230	1.79375E-06	1.40137E-08	1.09482E-10
240	0.000395	1.6475E-06	6.86458E-09	2.86024E-11
480	0.002310	4.8125E-06	1.0026E-08	2.08876E-11
960	0.020035	2.08702E-05	2.17398E-08	2.26456E-11
1000	0.020285	2.02846E-05	2.02846E-08	2.02846E-11
1024	0.043957	4.29264E-05	4.19203E-08	4.09378E-11
1920	0.257209	0.000133963	6.97725E-08	3.63399E-11
2000	0.149293	7.46463E-05	3.73232E-08	1.86616E-11
2048	0.324304	0.000158352	7.73201E-08	3.7754E-11
3840	2.425054	0.000631525	1.6446E-07	4.2828E-11
4000	2.779926	0.000694981	1.73745E-07	4.34363E-11
4048	2.740610	0.000677028	1.6725E-07	4.13167E-11



[1] https://docs.oracle.com/cd/E19059-01/stud.10/819-0501/7_tuning.html