MA513
Parallel Computing
Assignment-7 Report

Paranjay Bagga
160101050

Problem Statement :-

Perform **Polynomial Multiplication** using Pthreads while observing the speedup for different values of no. of Threads involved as well as the problem size.

System Specifications :-

All the experiments have been performed on **Dell Inspiron 5559** laptop with **Intel i5-6200U dual core processor** and **8 GB RAM**. Each core has **2 threads.** Also, it was made insured that no other applications were running in the background during each experiment which could have incurred biased readings.

Algorithms :-

(A[ ] * B[ ] = C[ ])

The 2 polynomials are represented by the 2 arrays A & B which contain their various coefficients respectively. The coefficients of the resultant multiplication solution are stored in the array C.
Three versions of programs are considered. One using the naive **School Method**, second one using **Karatsuba's Algorithm** and the last one using **Fast Fourier Transform(FFT) Algorithm** for polynomial multiplication.

- **Using School Method :**
  In this, each PE is allotted a specific set of indices of the final solution array(C), each of whose values will be independently calculated and hence updated by that particular PE alone. The PEs(or Threads) are alloted the indices of the C array in a **circular** manner so that the work distribution among various threads remains exactly equal.

- **Using Karatsuba's Algorithm :**
  Here, the divide and conquer technique is used. The problem size is cut into half by dividing both A and B arrays into left and right halves and calculating the required result for the problem using the solution to the 3 multiplication subproblems obtained as a result, thus reducing one extra multiplication involved at each step.

  Let the final partial result be stored in tmpPartialC[ ] array. Then,
  tmpPartialC[ ] = (A[L]*B[L]*10^N) + (A[L]*B[R]+A[R]*B[L])*10^(N/2) + A[R]*B[R]
  {where L => l:mid , R => mid+1:r}

  #1 -> array_LL[ ] = A[L] * B[L]
  #2 -> array_HH[ ] = A[R] * B[R]
  #3 -> array_LH[ ] = halfSumA[ ] * halfSumB[ ]
  {where halfSumA[ ] = (A[L] + A[R]), halfSumB[ ] = (B[L] + B[R])}

  Now, using these 3 multiplications, the final solution tmpPartialC[ ] is formed :-
  array_LH[ ] = array_LH[ ] - (array_LL[ ] + array_HH[ ])
  Thus, array_LH[ ] equals => (A[L]*B[R]) + (A[R]*B[L])
  {since (A[L]*B[R]+A[R]*B[L]) = (A[L]+A[R])*(B[L]+B[R]) - A[L]*B[L] - A[R]*B[R]}
  Thus finally,

tmpPartialC[ ] => (array_LL[ ]*10^N) + (array_LH[ ])*10^(N/2) + (array_HH[ ])

For implementing this, the process starts with 1 thread which then recursively creates 2 more threads for carrying on the 3 multiplication subproblems, of which 2 recursive calls are made by the 2 newly created threads and the third call made by the parent thread itself. The new threads are created only upto a certain specified MAX_LEVEL, after which all the 3 calls are made by the parent thread with no more thread creation, since then there would be no limit on the number of created threads used in a process.
In the experiments, MAX_LEVEL is taken from 0 to 5 linearly, thus producing a maximum of 243 threads.

- **Using FFT Algorithm :**
  Here, the amazing concept of Fast Fourier Transform is utilized for carrying on the polynomial multiplication in O(logN) time. The basic idea of the FFT is to apply divide and conquer. We divide the coefficient vector of the polynomial into two vectors, recursively compute the DFT for each of them, and combine the results to compute the DFT of the complete polynomial.
  Firstly, the Discrete Fourier Transform (**DFT**) is computed for each of the polynomials A (*DFT_A*) & B (*DFT_B*) using their respective coefficient arrays. (Here computation of DFT refers to converting the problem space from dealing with coefficients of the array to a set of discrete points which satisfy the polynomial which are thus generated from the array. Also, these points can further be used to generate the coefficients again, which refers to the computation of the **Inverse DFT**.)
  Now, using the individual DFTs (which generate the same x-coordinate points), DFT for A*B is calculated using
  DFT(A·B) = DFT(A)·DFT(B)

{by multiplying each element of one vector by the corresponding elements of the other vector}

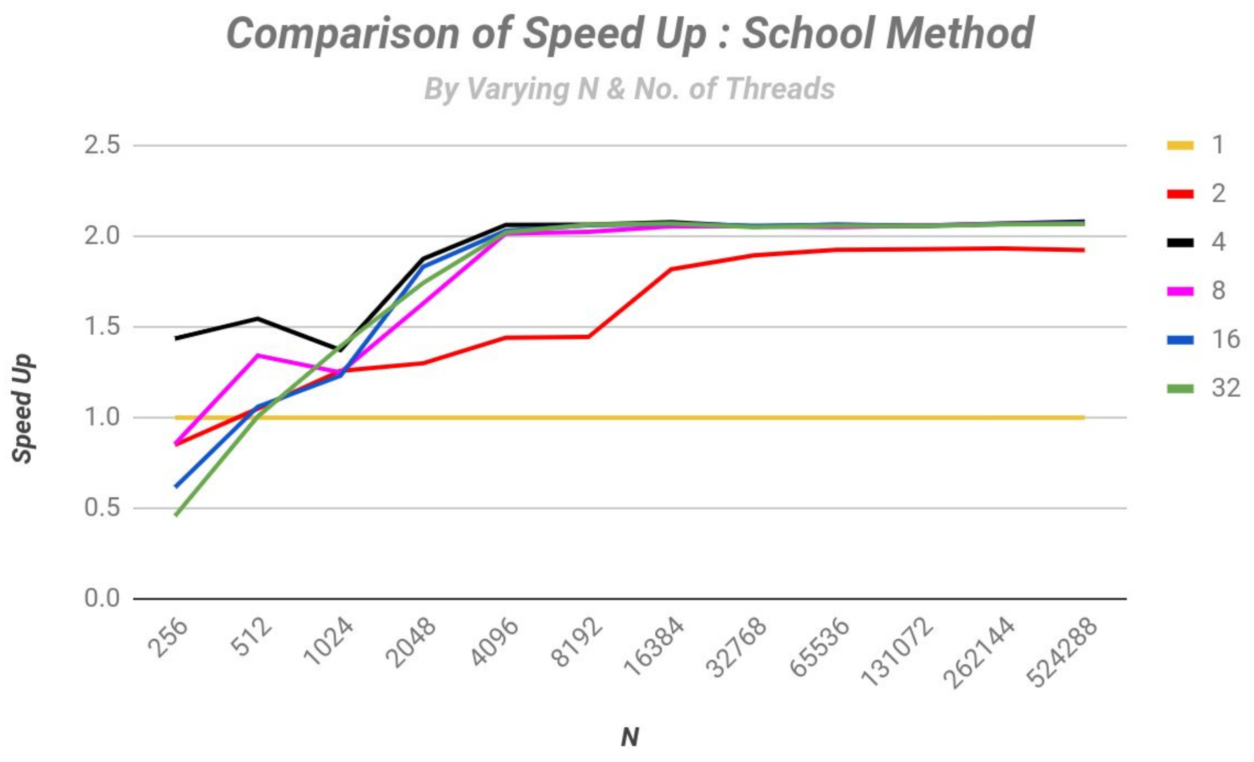
Finally, applying the inverse DFT, we obtain the required result:
A·B = InverseDFT(DFT(A·B))
For implementing this, while calculating the DFT, the individual indices of the final solution array are given only to a particular individual thread_id. The division of these indices is further done on the basis of the level at which a process is in, which further corresponds to the separation b/w the pair of indices, of the original array, under each thread at a particular instant of time. The thread_ids are allotted the indices in a cyclic manner for easy implementation. Also, the elements of the original array are reordered at the beginning in such a way that the algorithm demands and thus could help in facilitating in-place computation.

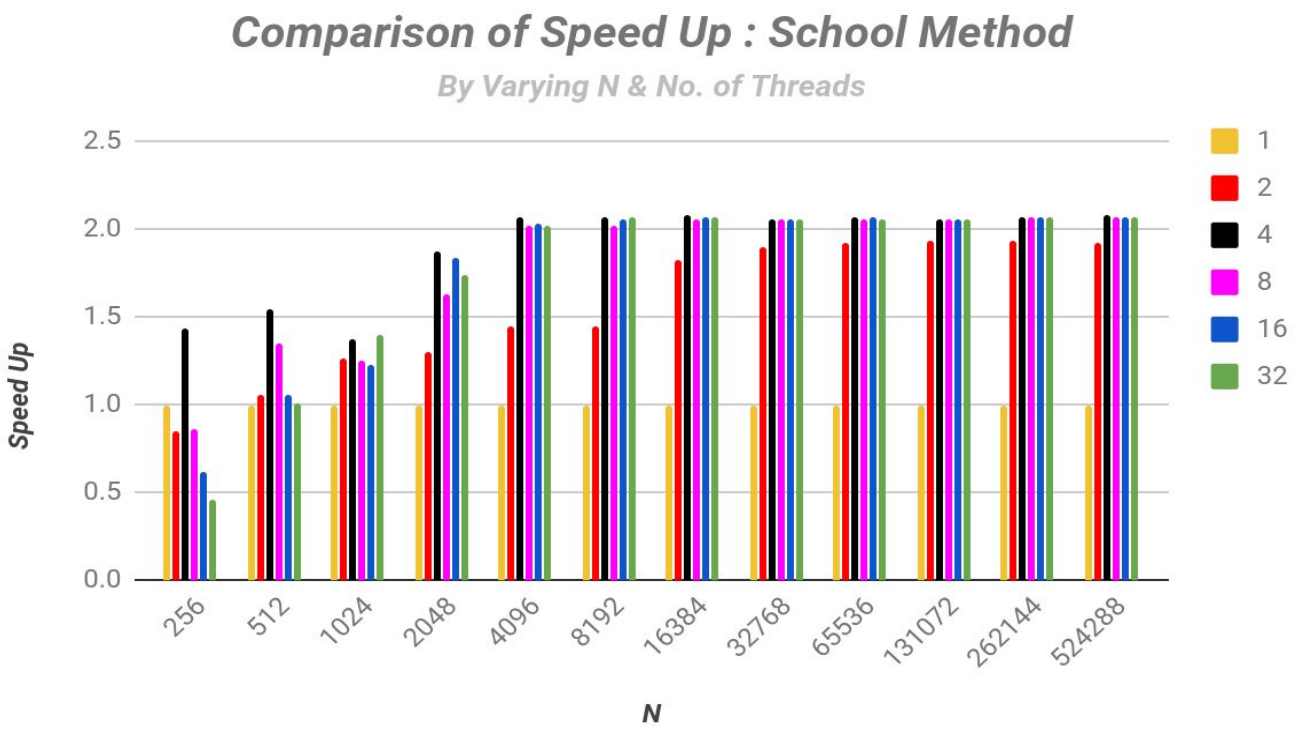
Experiments :-

The value of *N* is varied from **2^8(256)** upto **2^19(524288)** (& upto **2^25(33554432)** for FFT algorithm). The results are shown in the tables and graphs below. The values in the table are average values of 8 (or 2 for larger N) executions to avoid any unexpected behaviour introduced as a result of thread switching.

1. ***Speed Up*** | ***School Method*** | Varying both N & No. of Threads :-

N	**No. of Threads**					
	1	**2**	**4**	**8**	**16**	**32**
256	1	0.849707	1.436024	0.854714	0.616579	0.458257
512	1	1.04987	1.545202	1.342327	1.060245	1.006897
1024	1	1.258302	1.373211	1.249227	1.231124	1.393585
2048	1	1.299707	1.875244	1.629051	1.830804	1.743064
4096	1	1.440969	2.06207	2.015286	2.030731	2.02071
8192	1	1.444587	2.063278	2.023524	2.059798	2.063199
16384	1	1.818158	2.077229	2.054958	2.06727	2.069552
32768	1	1.893912	2.053518	2.05486	2.05774	2.050372
65536	1	1.924833	2.063738	2.049505	2.061287	2.056737
131072	1	1.927632	2.057507	2.056821	2.056337	2.056347
262144	1	1.932841	2.068908	2.068703	2.067155	2.06548
524288	1	1.923998	2.079943	2.069126	2.071002	2.067609


Comparison of Speed Up : School Method
By Varying N & No. of Threads
Speed Up
2.5
2.0
1.5
1.0
0.5
0.0
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
N
1
2
4
8
16
32

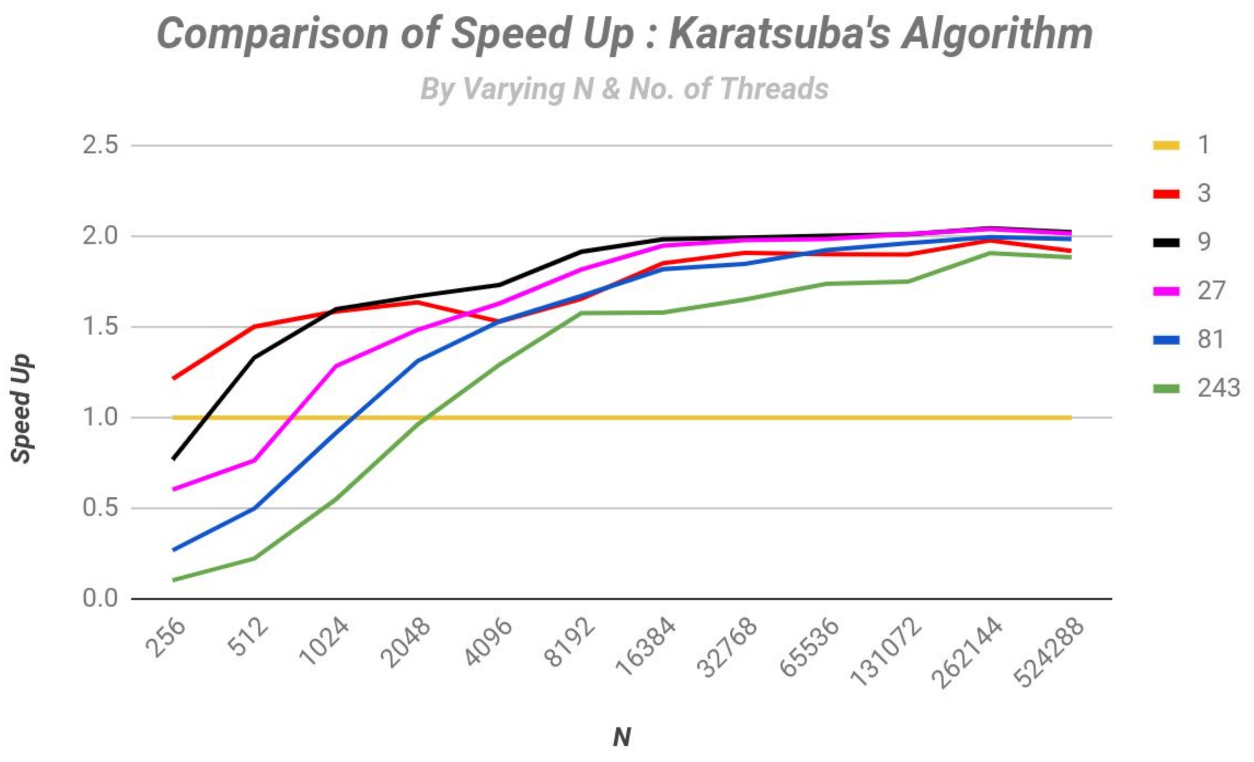

Comparison of Speed Up : School Method
By Varying N & No. of Threads
Speed Up
2.5
2.0
1.5
1.0
0.5
0.0
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
N
1
2
4
8
16
32

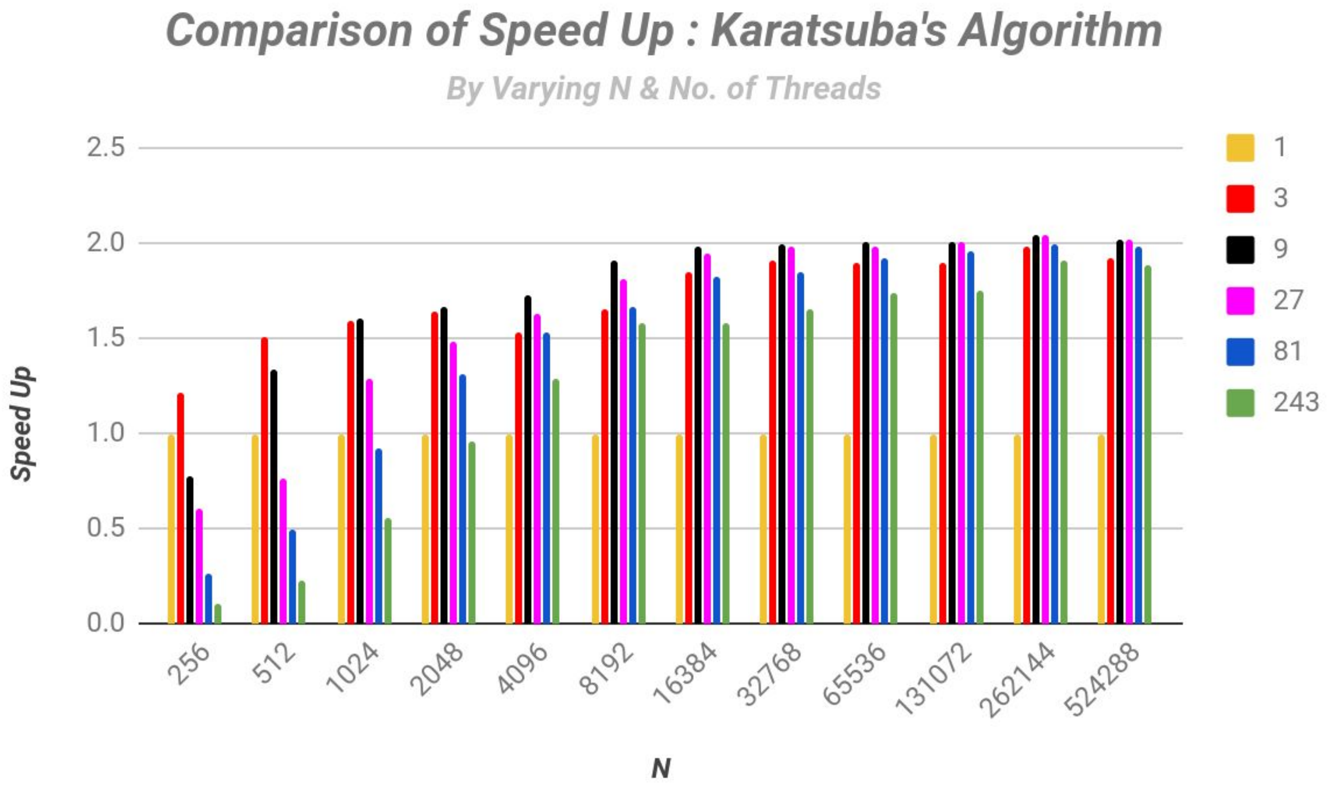
Observations :-

- From the above two graphs, which just represent the same thing differently, it is clear that the Optimal No. of threads required in School Method are **4**, since it provides the greatest speed up among all for any value of N.
- From N=*512* onwards, it is beneficial to use any number of threads strictly greater than **1** (hence parallelism is better) since the speed up becomes greater than **1** for all values of no. of threads taken.
- Also, from N=*1024* onwards, it is more optimal in taking strictly more than **2** threads for the process, since all other thread line graphs cross its mark from thereon. It finally settles at a speed up of **1.93.**
- Also, it can be observed that the rest three line graphs{8,16,32} take nearly the same execution time after N=*4096*, with all 4 of them finally settling at a speed up of about **2.07**.

2. *Speed Up* | *Karatsuba Algo* | Varying both N & No. of Threads :-

N	No. of Threads					
	1	3	9	27	81	243
256	1	1.213671	0.769325	0.603231	0.269373	0.104368
512	1	1.501704	1.330992	0.764104	0.49983	0.224824
1024	1	1.585781	1.59838	1.284877	0.916743	0.550778
2048	1	1.635515	1.669659	1.484575	1.313442	0.963744
4096	1	1.529306	1.730798	1.629393	1.53073	1.292137
8192	1	1.655036	1.914027	1.816284	1.670517	1.576221
16384	1	1.851144	1.982468	1.947132	1.81783	1.579023
32768	1	1.907641	1.992266	1.978421	1.846533	1.650193
65536	1	1.899821	2.001859	1.984573	1.923827	1.738279
131072	1	1.898969	2.009584	2.011368	1.961671	1.749515
262144	1	1.976894	2.044042	2.038586	1.994154	1.905979
524288	1	1.91867	2.021929	2.015103	1.983854	1.884108


Comparison of Speed Up : Karatsuba's Algorithm
By Varying N & No. of Threads
2.5
2.0
1.5
1.0
0.5
0.0
Speed Up
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
N
1
3
9
27
81
243

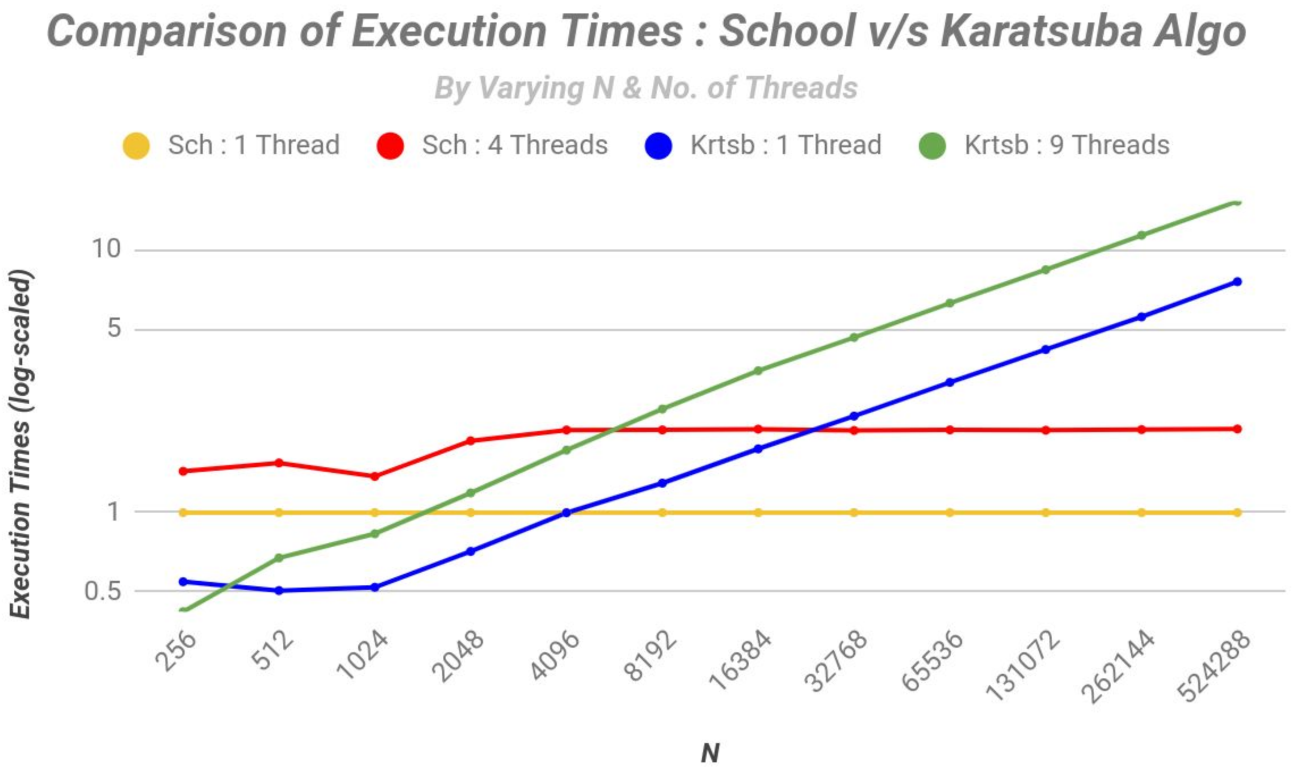

Comparison of Speed Up : Karatsuba's Algorithm
By Varying N & No. of Threads
2.5
2.0
1.5
1.0
0.5
0.0
Speed Up
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
N
1
3
9
27
81
243

Observations :-

- From the above two graphs, it is clear that the Optimal No. of threads required in Karatsuba's Algorithm are **9**, since it provides the greatest speed up among all for any N>=1024.
- It can be observed that the value of N, after which using parallel threads, strictly greater than **1**, is beneficial than serial code(using a single thread), increases with the number of threads being used.
- Also, from N=*4096* onwards, it is more optimal in taking strictly more than **3** threads but less than 243 threads for the process, since all three thread line graphs{9,27,81} cross its mark from thereon. It finally settles at a speed up b/w **1.9** and **2**, being closer to the former limit.
  Also, uptill N=*1024*, its line graph tops above all and hence signifies its priority of being optimal till this range.
- It is observed that the difference b/w the execution times of all line graphs, with greater than 3 threads, eventually reduces with N & hence each one of them crosses the 3-thread line. However, the point of crossing increases with the no. of threads used.
- Also, it can be observed that all the rest line graphs {other than 1 & 3} eventually settle at the speed up of about **2**, with the **9**-thread line graph being the optimal followed by 27, 81, 243 and so on (on increasing threads…), with the final speed up settling near **2.03**.

3. Comparison of Execution Times | School v/s Karatsuba Algo :-

N	No. of Threads			
	Sch : 1	Sch : 4	Krtsb : 1	Krtsb : 9
256	1	1.436024	0.545968	0.420027
512	1	1.545202	0.505363	0.672634
1024	1	1.373211	0.52	0.831158
2048	1	1.875244	0.711997	1.188792
4096	1	2.06207	0.99899	1.729049
8192	1	2.063278	1.294548	2.477799
16384	1	2.077229	1.74826	3.46587
32768	1	2.053518	2.328557	4.639106
65536	1	2.063738	3.129915	6.265649
131072	1	2.057507	4.173238	8.386471
262144	1	2.068908	5.552377	11.349292
524288	1	2.079943	7.559761	15.285297



Observations :-

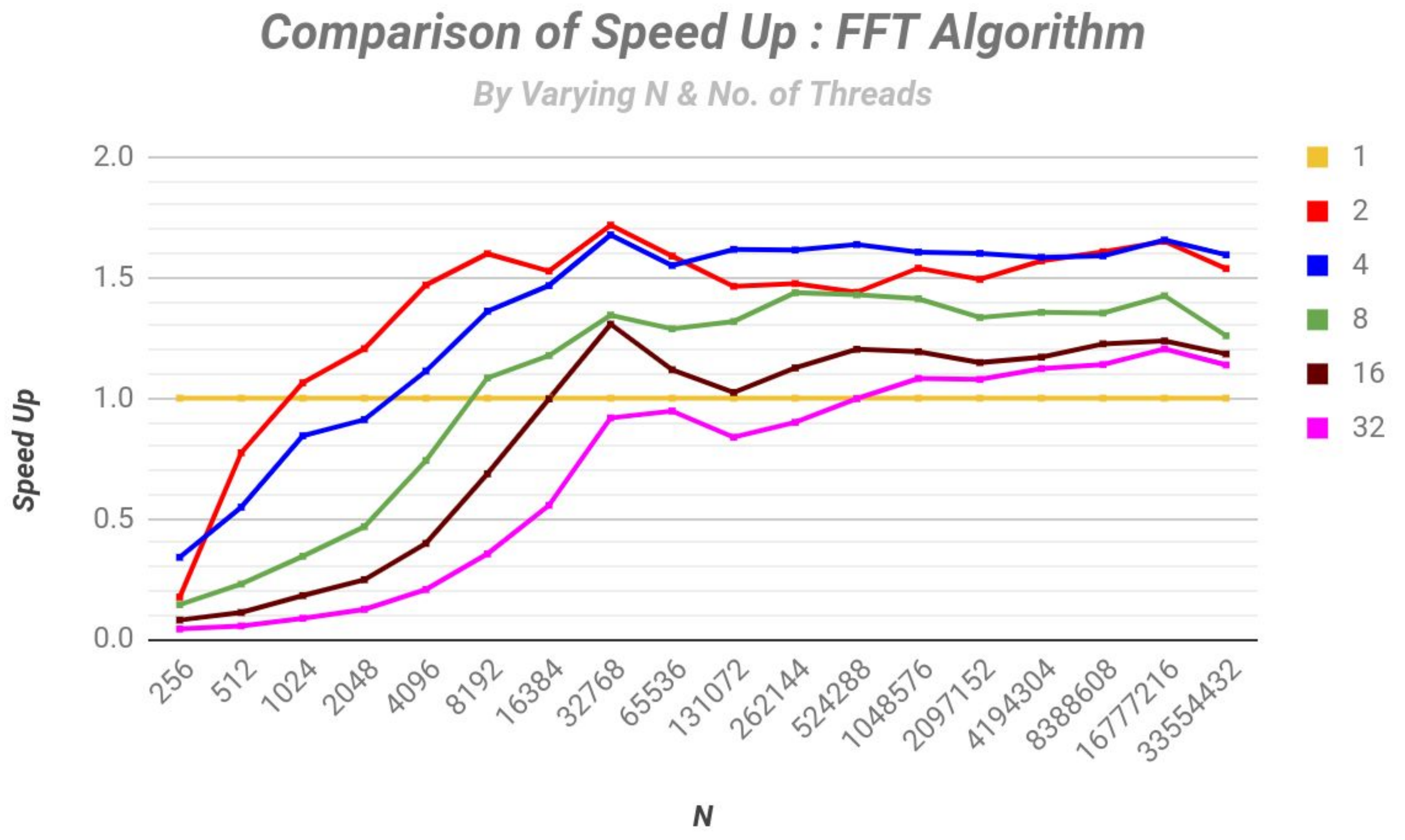
- From the above graph, it is clear that the optimal algorithm among them is Karatsuba's Algo for N>=**4096**, while for N<4096, the school method is more optimal.
- Also, the parallel versions of both algorithms show the same behaviour, with Karatsuba's 9-thread line graph crossing the mark with School method's 4-thread line at N approx. equal to 6144 {(4096+8192)/2}.
- Both of these thread numbers aur specially taken since they behave optimally in their resp. algorithms.
- Also, it can be clearly seen that the graph finally settles to 2 sets of parallel lines showing their respective max speed up in their resp. algorithm.

- The distance b/w these two parallel lines is also observed to be nearly the same as was seen from the earlier graphs since the max speed up reached in each algo individually was approx. **2**.
- Finally, it can be seen that the ratio of execution times, of Karatsuba's algorithm and the School method, keeps on increasing by a **constant factor** with increasing N {since the slope remains the same}.

4. ***Speed Up*** | ***FFT Algorithm*** | Varying both N & No. of Threads :-

N	No. of Threads					
	1	2	4	8	16	32
256	1	0.174933	0.339686	0.143233	0.079492	0.042520
512	1	0.773609	0.547768	0.228786	0.110667	0.055119
1024	1	1.063725	0.843960	0.344113	0.180921	0.086753
2048	1	1.205212	0.911064	0.467148	0.246990	0.123711
4096	1	1.469228	1.112626	0.741213	0.397574	0.206143
8192	1	1.599240	1.360988	1.083974	0.686075	0.353997
16384	1	1.527352	1.466909	1.176724	0.996907	0.555502
32768	1	1.717874	1.677565	1.344818	1.306880	0.918055
65536	1	1.589763	1.550571	1.287679	1.117949	0.946681
131072	1	1.464124	1.617320	1.318054	1.023776	0.837777

262144	1	1.475548	1.614506	1.437558	1.125564	0.899675
524288	1	1.439557	1.637444	1.428698	1.202868	0.998067
1048576	1	1.539351	1.606052	1.412499	1.192864	1.081870
2097152	1	1.493054	1.600911	1.334659	1.147652	1.077689
4194304	1	1.569683	1.584365	1.355993	1.170175	1.122532
8388608	1	1.607215	1.590376	1.353171	1.225500	1.139254
16777216	1	1.650433	1.656061	1.424880	1.237414	1.204346
33554432	1	1.537510	1.594682	1.258756	1.183439	1.137861

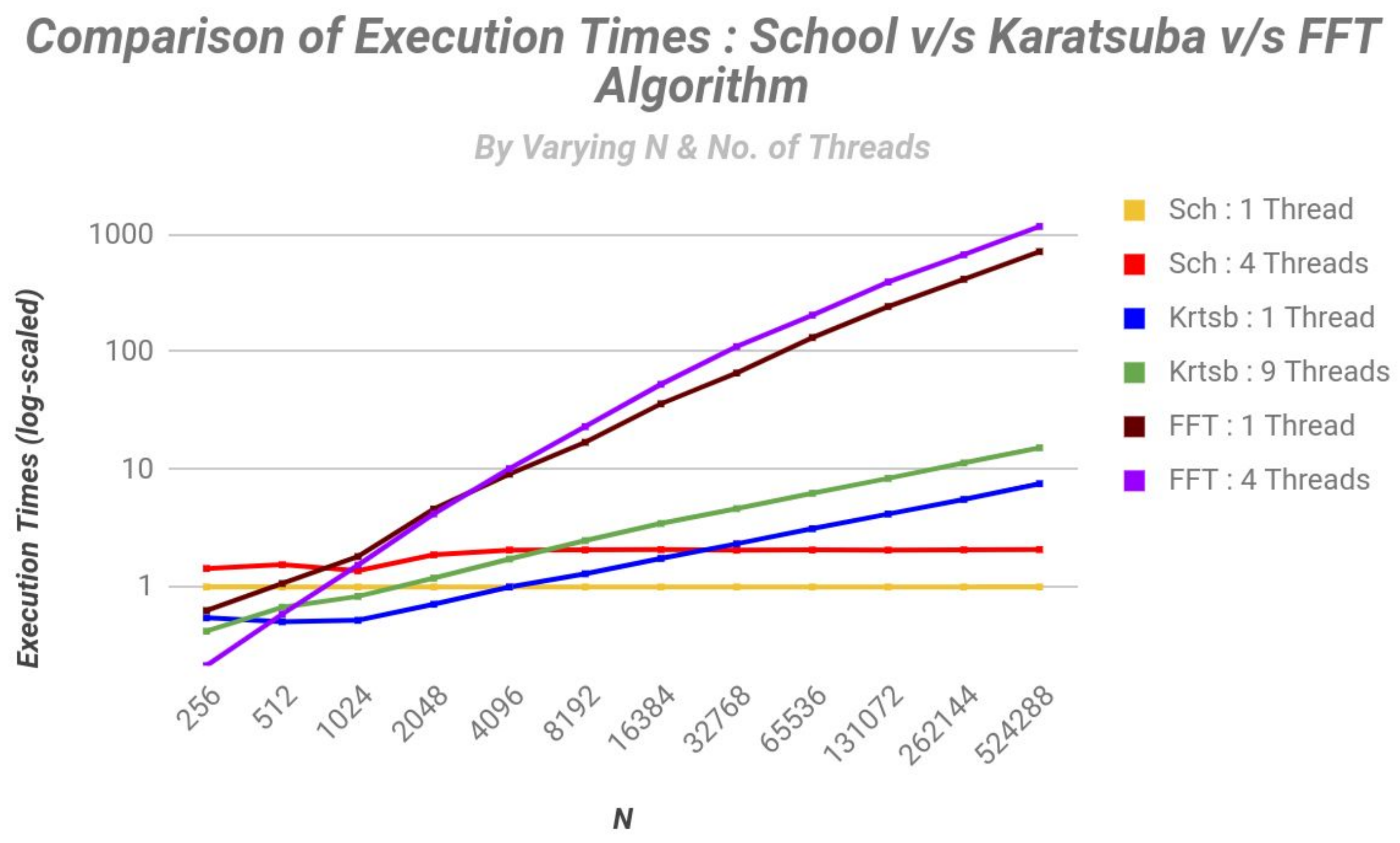


Observations :-

- From the above two graphs, it is clear that the Optimal No. of threads required in the FFT Algorithm are **4**, since it provides the greatest speed up among all for any N>=131072. Also, this result can be attributed to the fact that the system contains 4 threads in total, which as a result contributed to better parallelism. In other cases, the reason behind taking more time might lie in the time consumption because of context switching.
- It can be observed that the specific value of N, after which using parallel threads, strictly greater than **1**, is beneficial than serial code(using a single thread), increases as the number of threads being used is increased.
- Uptill N<*1024*, the serial code is better than any parallel version.
- When *1024*<=N<=*65536*, it is the most optimal in taking **2** threads for the process, since all other thread line graphs lie below its own.
- Similarly, when *65536*<N, using **4** threads is the most optimal. Also, both **2 & 4** thread line-graphs finally settle at a speed up of about **1.7**.
- It can also be seen that the difference b/w the execution times of all line graphs, with greater than 1 thread, eventually reduces with N & hence each one of them crosses the 1-thread line. Also, finally each one of them settle at an approximately constant speed up value which decreases with further increase in no. of threads. Approx. :- {4,2 : **1.7** | 8 : **1.4** | 16, 32 : **1.2** }

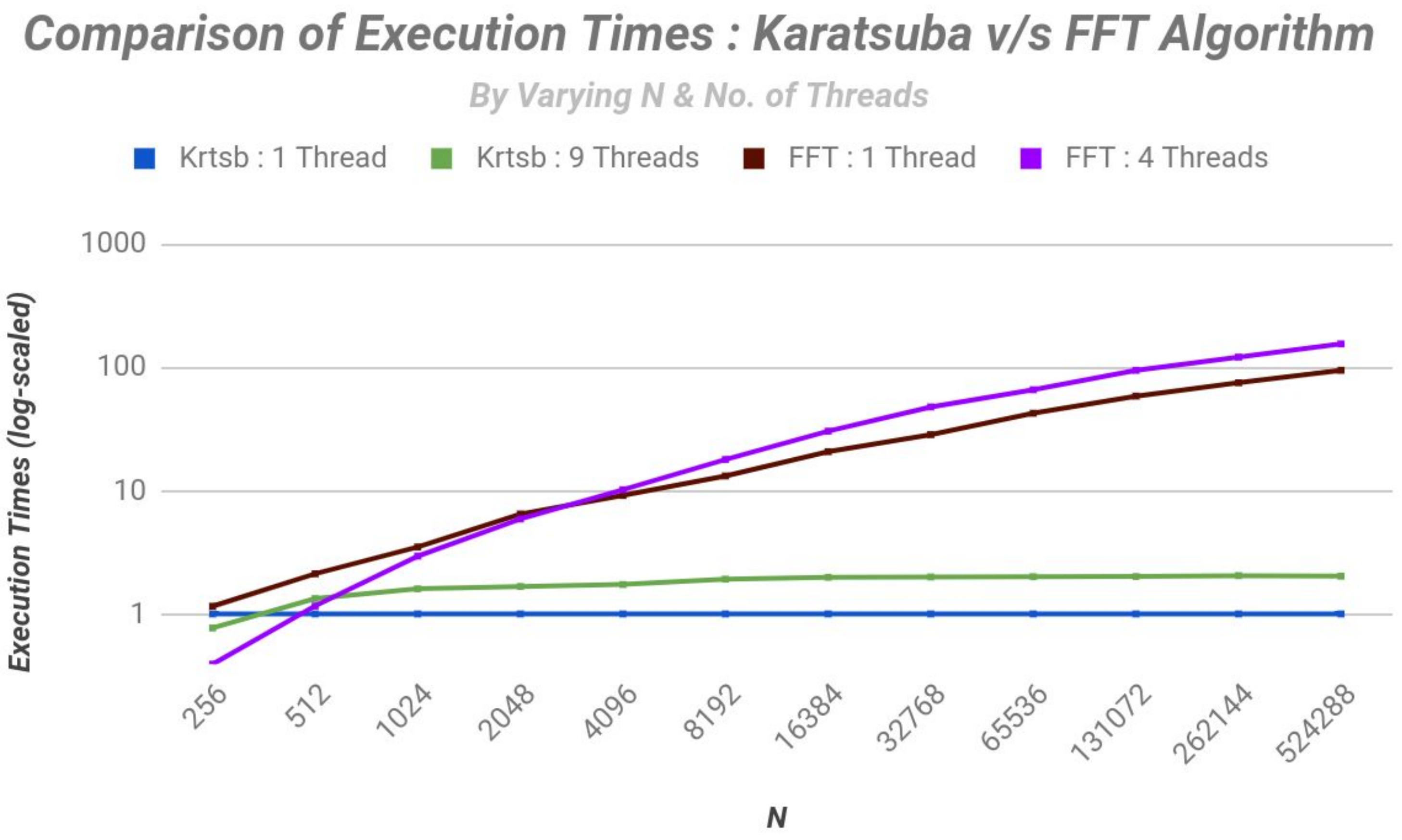
5. ***Comparison of Execution Times*** **|** ***School v/s Karatsuba v/s FFT Algorithm*** **:-**

N	No. of Threads					
	Sch : 1	**Sch : 4**	**Krtsb : 1**	**Krtsb : 9**	**FFT : 1**	**FFT : 4**
256	1	1.436024	0.545968	0.420027	0.629654	0.213885
512	1	1.545202	0.505363	0.672634	1.068899	0.585509
1024	1	1.373211	0.520000	0.831158	1.813953	1.530904
2048	1	1.875244	0.711997	1.188792	4.597530	4.188644
4096	1	2.062070	0.998990	1.729049	9.115346	10.141971
8192	1	2.063278	1.294548	2.477799	16.996600	23.132175
16384	1	2.077229	1.748260	3.465870	36.118840	52.983037
32768	1	2.053518	2.328557	4.639106	66.101363	110.889316
65536	1	2.063738	3.129915	6.265649	132.435684	205.350906
131072	1	2.057507	4.173238	8.386471	243.186288	393.309941
262144	1	2.068908	5.552377	11.349292	415.098161	670.178278
524288	1	2.079943	7.559761	15.285297	713.150695	1167.744247



N	No. of Threads			
	Krtsb : 1	Krtsb : 9	FFT : 1	FFT : 4
256	1	0.769325	1.153281	0.391753
512	1	1.330992	2.115112	1.158590
1024	1	1.598380	3.488372	2.944046
2048	1	1.669659	6.457235	5.882954
4096	1	1.730798	9.124564	10.152228
8192	1	1.914027	13.129373	17.868923

16384	1	1.982468	20.659881	30.306157
32768	1	1.992266	28.387264	47.621472
65536	1	2.001859	42.312867	65.609097
131072	1	2.009584	58.272805	94.245747
262144	1	2.044042	74.760445	120.701152
524288	1	2.021929	94.335084	154.468407



<u>Observations :-</u>

- From the above two graphs, it is clear that the optimal algorithm among them is the FFT Algo for all **N**.
- Also, the parallel versions of all the algorithms show the same behaviour, with FFT's 4-thread line graph behaving most optimally wr.t. Karatsuba's 9-thread line and School method's 4-thread line for all N>=*1024*.
- All these thread numbers aur specially taken since they behave optimally in their respective algorithms.
- Also, it can be clearly seen that the line-graphs finally settle to 3 sets of parallel lines, with the constant distance b/w them showing their respective max speed up in their corresponding algorithms.
- The distance b/w these parallel lines is also observed to be nearly the same for School Method and Karatsuba Algo as was seen from the earlier graphs (since the max speed up reached in each algo individually was approx. **<u>2</u>**), while this separation values is a bit less in case of FFT algo (since the max speed up reached in it was only approx. **<u>1.7</u>**).
- Finally, it can be seen that the ratio of execution times, of FFT algorithm wr.t. both Karatsuba's algorithm and the School method, keeps on increasing by a **constant factor** with increasing N {since the slope remains the same finally}.