Assignment 7

Inderpreet Singh Chera - 160101035

Two Polynomial Multiplication:

This multiplication of polynomials is carried out by 3 methods:

1. School Method
2. Karatsuba's Algorithm
3. FFT divide and conquer

System Settings

All tests were done on **Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz** processor. This computer is **dual core** where each core has **2 threads**. Also during each experiment it was insured that **no other applications** were running in the background, so that we don't have any biased readings.

Implementations

We have 2 polynomials, say a and b with n-1 max power, i.e., the total number of elements in each polynomial is n.

School Method

This is a naive way of multiplying 2 polynomials. We multiply each element of b with all elements of a and get the output. This takes O(n) time complexity.

To do the same in parallel and to distribute almost the same amount of work to each processor, we tell each processor to get the answer of each mod p, which has value equal to processor id, coefficient of final aray. If p divides n, then it can be shown that the same amount of work is done by all the threads.

So, say we have 4 processors, then to get the coefficients of resultant array, processors will be allocated as $P_0P_1P_2P_3P_0P_1$... and so on.

Karatsuba's Algorithm

This is a divide and conquer algorithm. So, here doing completely parallel work, we will require to have exponential number of threads($O(3^n)$), as to get the result of each subpart we have to make a new thread. But, the system didn't allow to make exponential number of threads.

So, partial parallel algorithm is implemented. Upto a certain level, all the work is distributed parallely and then serial work is done. By this, if level is 0, then 1 thread is used. If the level is 1, then 4 threads. If it is 2, then 13 threads, and so on. Then after the certain max level all the work is done serially using the karatsuba algorithm.
After experimentations, I found that this gave speed up less than 1 for a large number of threads, because the main thread that created the other 3 sub threads didn't do much work. So, to make the main thread do more work, I created only 2 sub processes and job for the third process was given to the main thread to be done. As seen in observations this gave better results.

FFT Divide and Conquer

We first find DFT of two n size polynomials at 2*n points. Then do the product between the two 2*n size polynomials we got, pair wise, and find inverse DFT to get the resultant polynomials which is the product of the given two polynomials. FFT is a method to perform DFT.

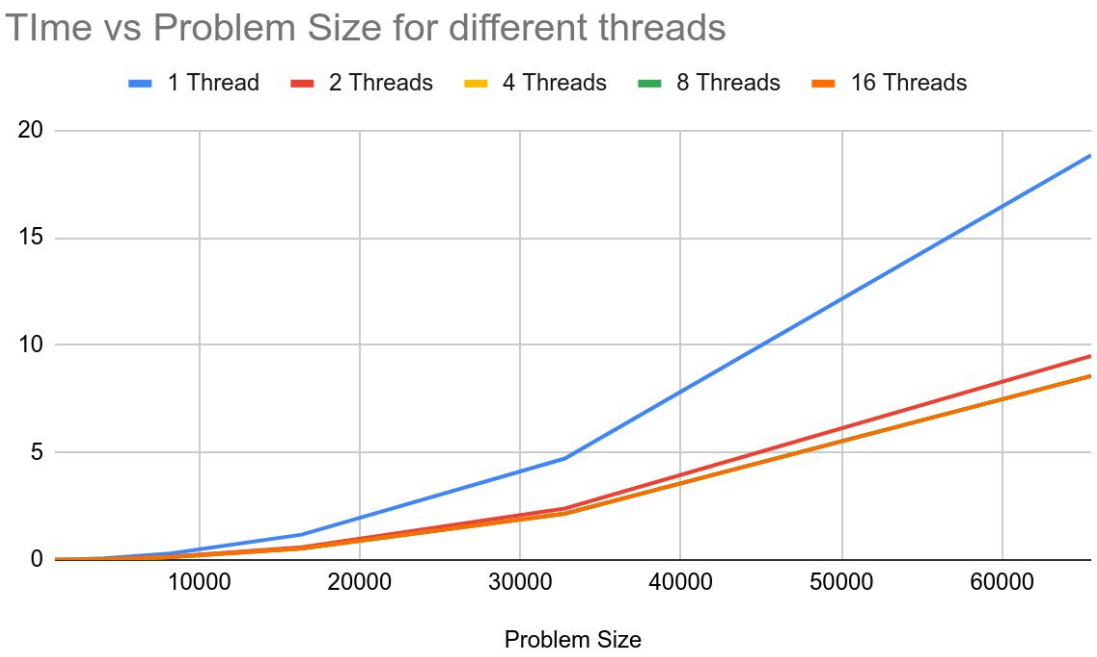
Bit reversal array tells us at which position each coefficient will be after doing odd even position splits, which is required to perform FFT. So, we divide the coefficients between the processes according to the bit reversal array. Each thread will get n/p elements continous elements of the bit reversal array. Now, it can be observed that for $\log_2(2*n) - \log_2$(num of threads) steps, each thread has all the required information with itself. But, beyond that it will have to get it from other threads.

Observations

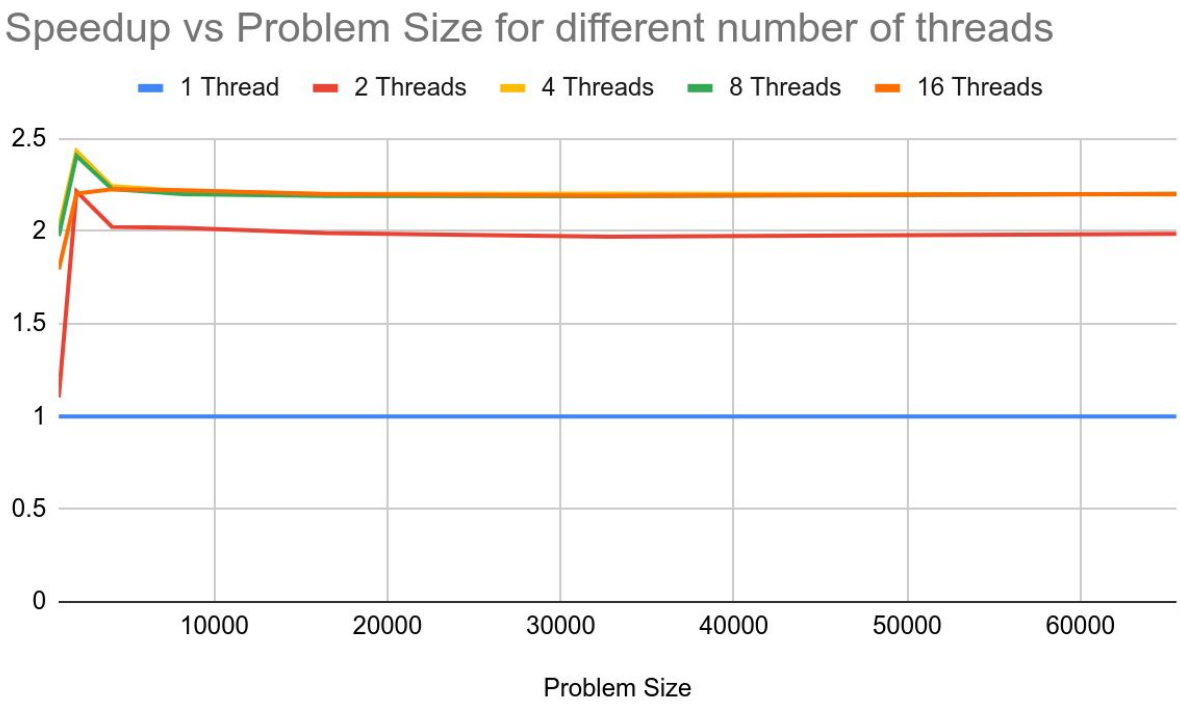
School Method:

1. **TIme (and speedup) vs Size for different number of threads:**

Problem Size	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1024	0.00467371	0.0042328	0.00231602	0.00236647	0.00260466
2048	0.0216977	0.00979527	0.00891673	0.00901242	0.00984955
4096	0.075046	0.0370683	0.0334506	0.0336668	0.0336938
8192	0.297658	0.14748	0.134205	0.135282	0.133997
16384	1.17742	0.591356	0.534705	0.537322	0.53486
32768	4.7213	2.39507	2.14306	2.15579	2.15348
65536	18.8549	9.49411	8.57282	8.56366	8.56261



Problem Size	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1024	1	1.104165092	2.017992073	1.97497116	1.794364716
2048	1	2.215120155	2.433369632	2.40753316	2.202912823
4096	1	2.024533092	2.243487411	2.229080281	2.227294042
8192	1	2.018294006	2.217935248	2.200277938	2.221378091
16384	1	1.991051076	2.201999233	2.191274506	2.201361104
32768	1	1.971257625	2.203064777	2.190055618	2.192404852
65536	1	1.98595761	2.1993813	2.201733838	2.202003828



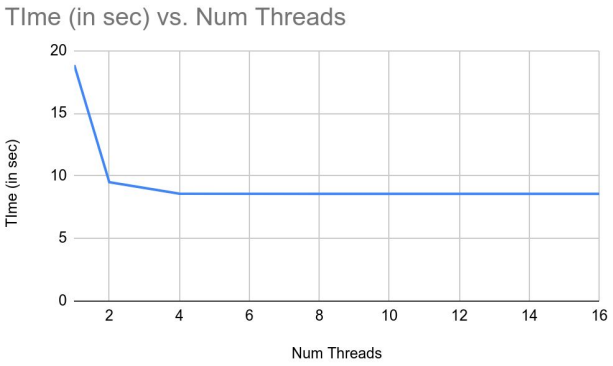
Observations:

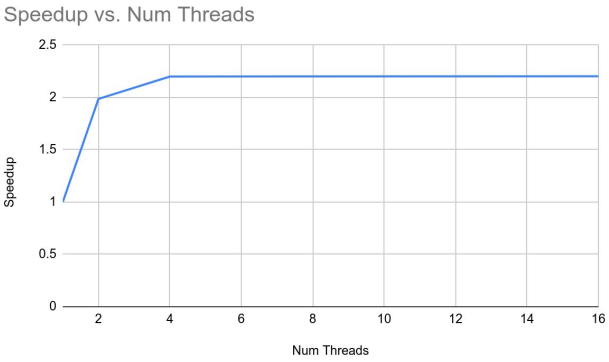
Time taken by a single threaded program is more than a multi threaded program. Time taken reduces till 4 threads, and then remains almost constant because my pc can handle a maximum of 4 threads that can run parallel.

Speedup is observed maximum for problem size 2048 and after that it reduces and remains almost constant.

2. **Time(and speedup) vs number of threads for problem size = 2^16.**

Num Threads	TIme (in sec)	Num Threads	Speedup
1	18.8549	1	1
2	9.49411	2	1.98595761
4	8.57282	4	2.1993813
8	8.56366	8	2.201733838
16	8.56261	16	2.202003828





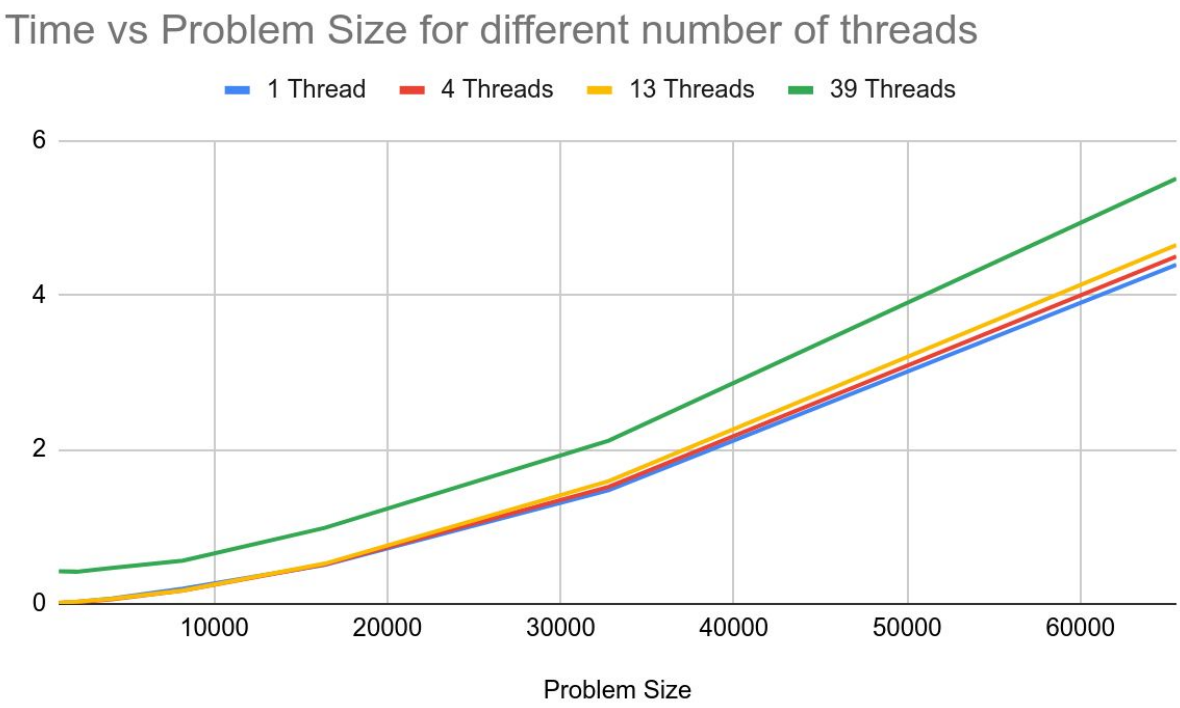
Observations:

We observe that time decreases upto 4 threads and then remain constant. Similarly, speedup increases till 4 threads and then remains constant.

Karatsuba Algorithm(making three sub processes for each node):

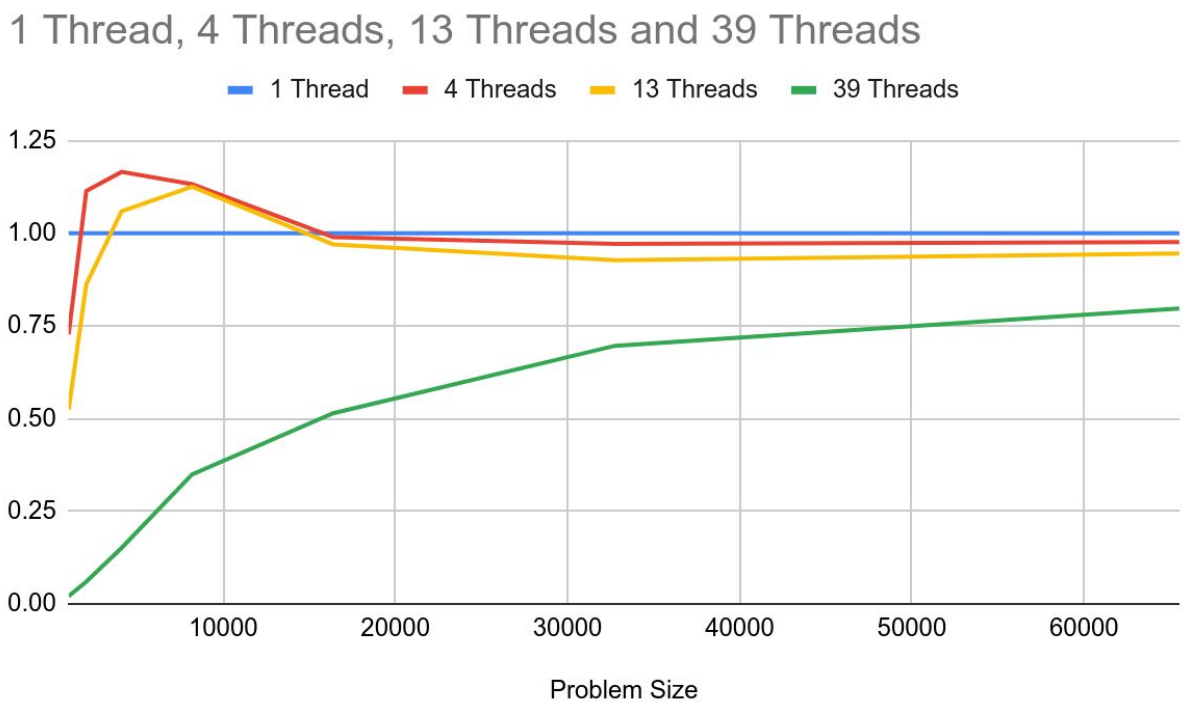
1. **TIme (and speedup) vs Size for different number of threads:**

Problem Size	1 Thread	4 Threads	13 Threads	39 Threads
1024	0.00869528	0.011958	0.0165868	0.420957
2048	0.0250423	0.0224731	0.0290358	0.417112
4096	0.0703828	0.0603751	0.0664368	0.466027
8192	0.195823	0.172887	0.173897	0.560788
16384	0.506459	0.511564	0.522004	0.984732
32768	1.47302	1.51661	1.58884	2.1143
65536	4.39404	4.49971	4.64765	5.50956



Speedup:

Problem Size	1 Thread	4 Threads	13 Threads	39 Threads
1024	1	0.7271516976	0.5242289049	0.02065598149
2048	1	1.114323347	0.8624628906	0.06003735208
4096	1	1.165758732	1.059394793	0.151027301
8192	1	1.132664688	1.126086131	0.3491925647
16384	1	0.990020799	0.9702205347	0.5143115081
32768	1	0.9712582668	0.9271040508	0.6966939413
65536	1	0.9765162644	0.945432638	0.7975301113



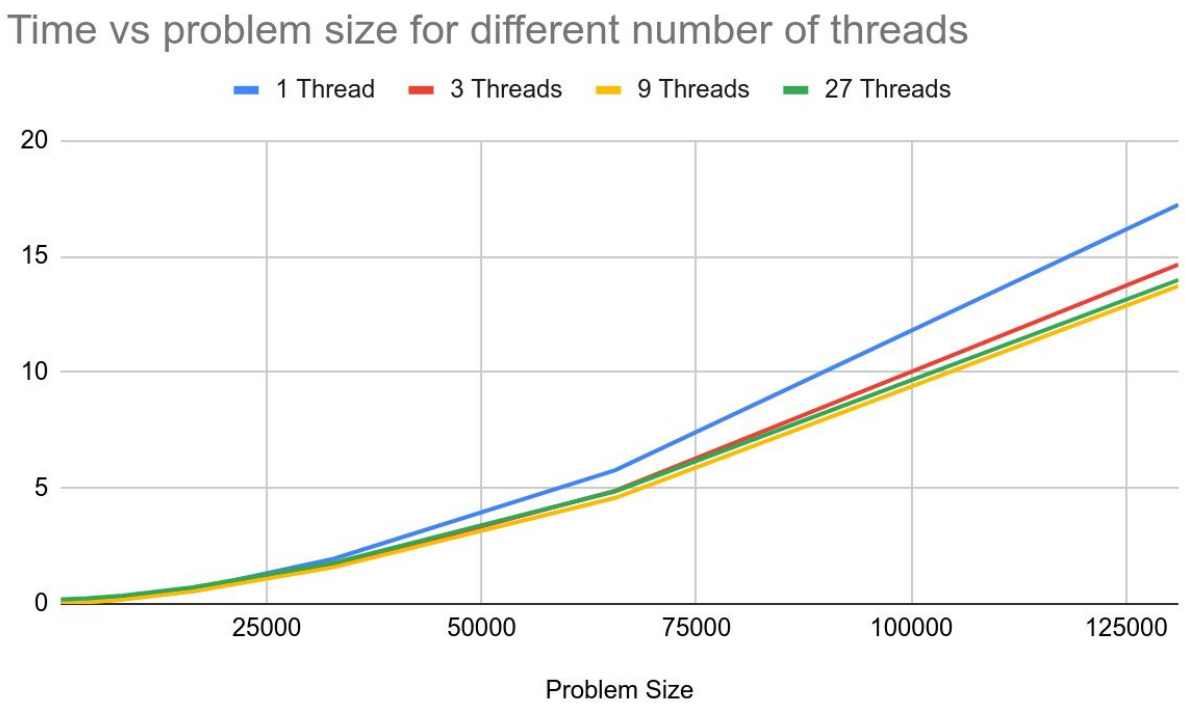
Observations:

Speedup is not coming greater than 1, except for when we are having 4 threads. It is possible as major work as the size of the problem increases is serial.

Karatsuba Algorithm(making two sub processes for each node):

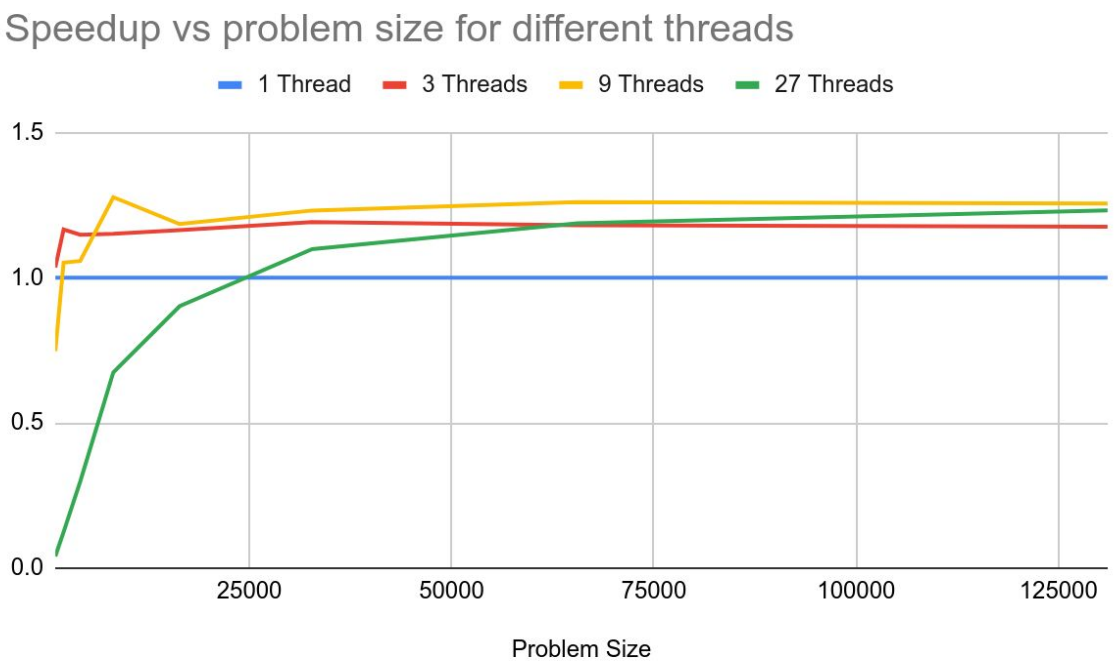
1. **TIme (and speedup) vs Size for different number of threads:**

Problem Size	1 Thread	3 Threads	9 Threads	27 Threads
1024	0.00819988	0.00792061	0.0109689	0.200542
2048	0.0261496	0.0224144	0.0248478	0.209693
4096	0.0712981	0.062092	0.0674171	0.23963
8192	0.238586	0.207194	0.186757	0.353754
16384	0.640377	0.550164	0.540314	0.710019
32768	1.94562	1.63214	1.57951	1.77063
65536	5.77008	4.88484	4.57656	4.85847
131072	17.2362	14.6579	13.7235	13.9874



SpeedUp:

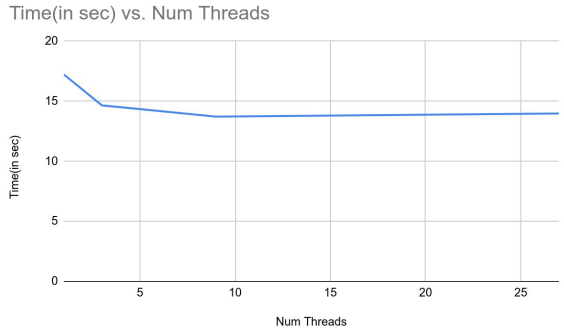
Problem Size	1 Thread	3 Threads	9 Threads	27 Threads
1024	1	1.035258648	0.7475571844	0.04088859192
2048	1	1.166642872	1.052390956	0.1247042104
4096	1	1.148265477	1.057566997	0.2975341151
8192	1	1.151510179	1.277521057	0.6744404304
16384	1	1.163974742	1.185194165	0.9019153009
32768	1	1.192066857	1.231787073	1.09882923
65536	1	1.181221903	1.260789763	1.187633144
131072	1	1.175898321	1.2559624	1.232266182

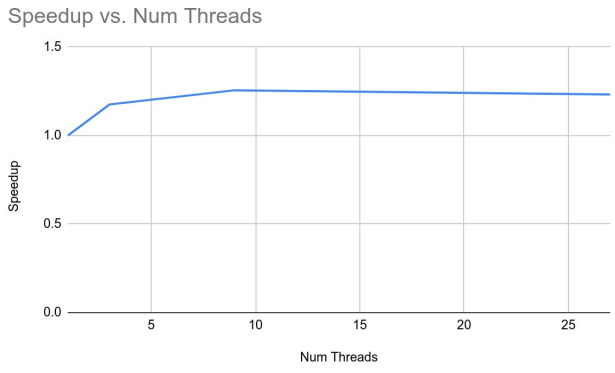


We observe here rather than in previous case that speed up is greater than 1 for larger problem size and hence it is better than the previous parallel algo devised for Karatsuba algorithm.

2. **Time(and speedup) vs number of threads for problem size = 2^17.**

Num Threads	Time(in sec)	Num Threads	Speedup
1	17.2362	1	1
3	14.6579	3	1.175898321
9	13.7235	9	1.2559624
27	13.9874	27	1.232266182



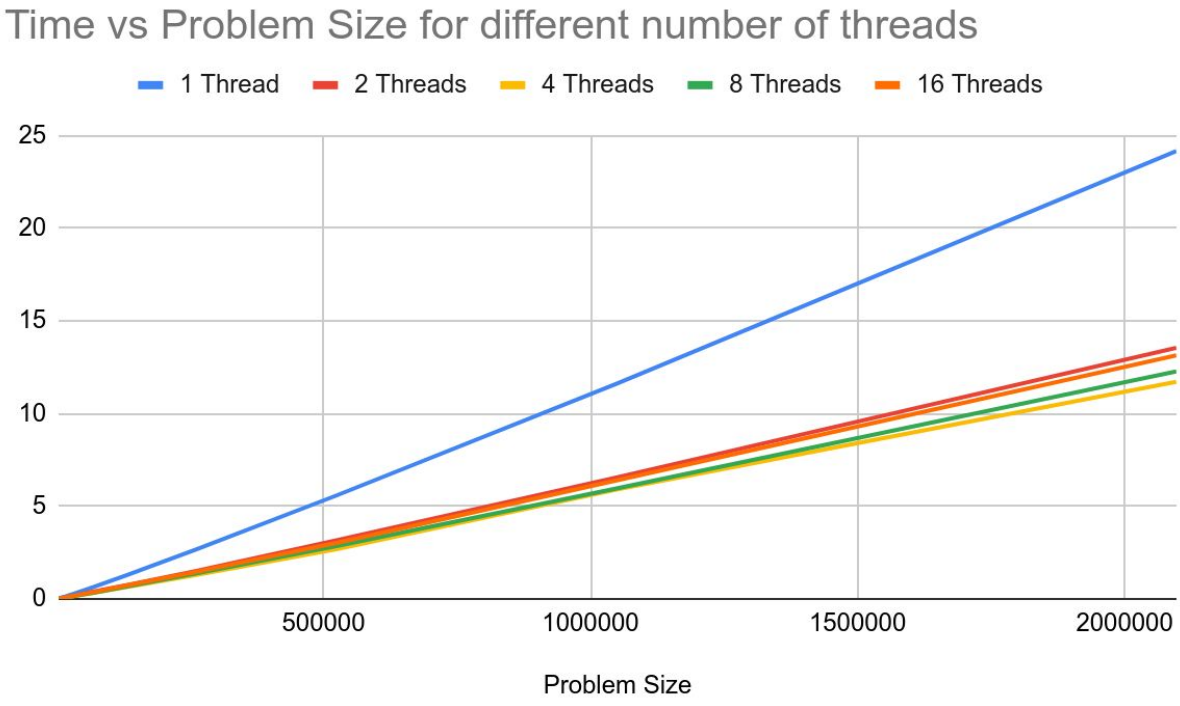


FFT Divide and Conquer Algorithm:

1. **TIme (and speedup) vs Size for different number of threads:**

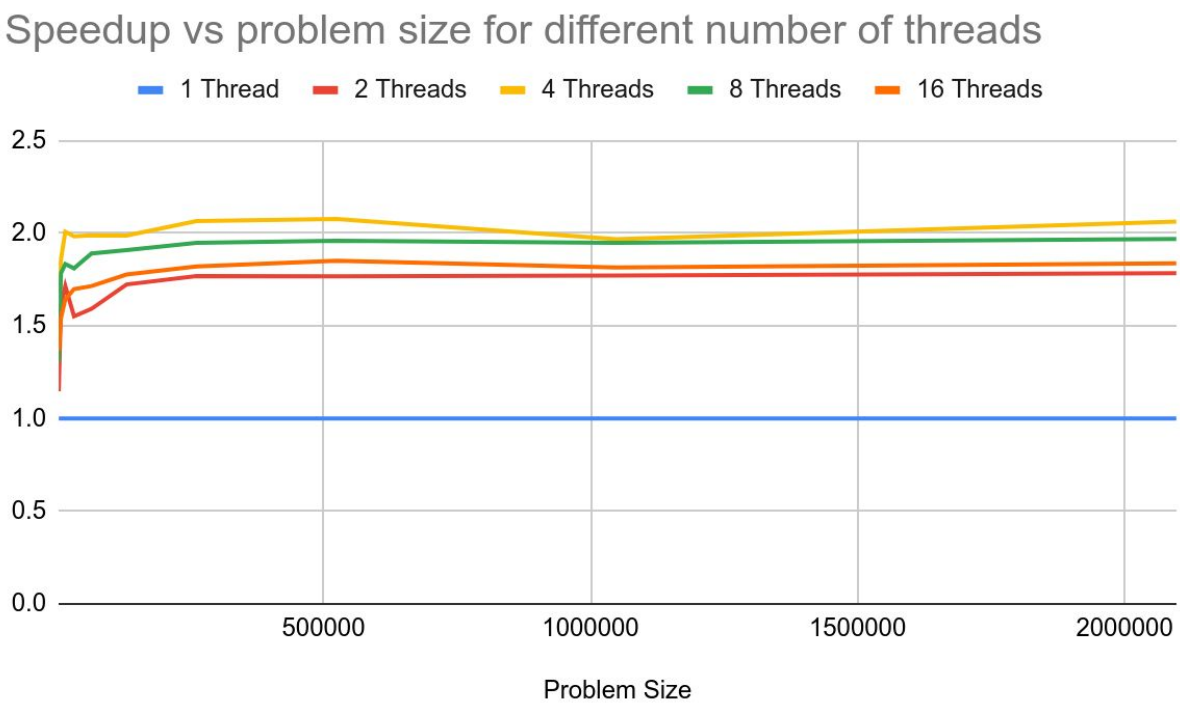
   Time vs Size:

Problem Size	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
4096	0.030531	0.0266122	0.0181622	0.0234059	0.0223404
8192	0.0621086	0.038582	0.0334268	0.0348014	0.0402809
16384	0.129865	0.0756818	0.0646597	0.0707912	0.0789841
32768	0.285922	0.184241	0.144124	0.15786	0.168271
65536	0.603185	0.378726	0.303118	0.318957	0.351591
131072	1.27726	0.740841	0.642522	0.668999	0.718517
262144	2.67435	1.51149	1.29453	1.37276	1.46845
524288	5.5821	3.15774	2.68764	2.84733	3.01494
1048576	11.615	6.55622	5.90016	5.96258	6.39585
2097152	24.1736	13.5464	11.7132	12.2761	13.1473



Speedup vs Size:

Problem Size	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
4096	1	1.14725577	1.681018819	1.304414699	1.366627276
8192	1	1.609781764	1.858048033	1.784658088	1.541887098
16384	1	1.715934346	2.008438022	1.834479427	1.644191679
32768	1	1.551891273	1.98386112	1.811237806	1.699175734
65536	1	1.592668578	1.989934613	1.891116984	1.715587145
131072	1	1.724067647	1.98788524	1.909210627	1.777633654
262144	1	1.769346803	2.065884916	1.948155541	1.821206034
524288	1	1.76775162	2.07695227	1.960468228	1.851479631
1048576	1	1.771600099	1.968590682	1.947982249	1.816021326
2097152	1	1.784503632	2.063791278	1.969159587	1.8386741

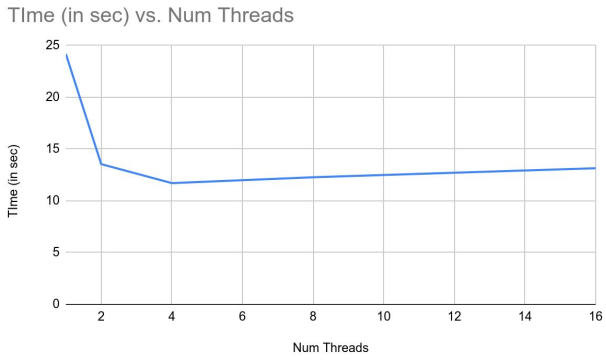


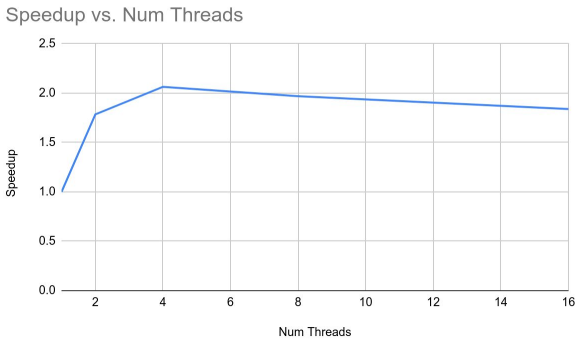
Observations:

We observe max speedup when 4 processes are used. Also, time taken by FFT divide and conquer is lesser than its counterparts. Hence, works well than other counterparts.

2. **Time(and speedup) vs number of threads for problem size = 2^21.**

Num Threads	TIme (in sec)	Num Threads	Speedup
1	24.1736	1	1
2	13.5464	2	1.784503632
4	11.7132	4	2.063791278
8	12.2761	8	1.969159587
16	13.1473	16	1.8386741





TIme Comparison between the 3 algos at different problem size, using 4 threads:

Problem Size	School Method	Karatsurba Algo	FFT Divide & Conquer
4096	0.0334506	0.062092	0.0181622
8192	0.134205	0.207194	0.0334268
16384	0.534705	0.550164	0.0646597
32768	2.14306	1.63214	0.144124
65536	8.57282	4.88484	0.303118

Observations:

We observe that FFT divide and conquer performs best between the three, whereas school method performing the worst.