1. Objective:

– The programs each read a matrix from an input file (`inp.txt`), and calculate the square of the matrix by distributing the workload among multiple threads.

– The result is written to an output file , including the time taken for the computation.

2. Key Components:

– `multiplyMatrices`: Function to perform matrix multiplication given certain constraints.

– `main`:

  – Reads input matrix from file.

  – Splits the work among `K` threads.

  – Measures the time taken for the computation.

  – Writes the result and time to an output file.

3. Thread Usage:

– Threads are created to parallelize the matrix multiplication task.

– All 3 programs divide the workload between K threads in different ways

- Program 1: Chunk– The C matrix is divided into chunks of size p = N/K. Then thread1 will compute the values of the rows 1 to p of C; thread2 will compute the values of the rows (p + 1) to (2p); thread3 will compute the values of the rows (2p + 1) to (3p) and so on. Thus, the thread th_i will be responsible for computing the rows corresponding to chunk i.

- Program 2: Mixed– Here, the rows of the C matrix are evenly distributed among the threads. Thread1 will be responsible for the following rows of the C matrix: 1, (k+1), (2k+1), ... Similarly, thread2 will be responsible for the following rows of the C matrix: 2, (k+2), (2k+2), ...This pattern continues for all the threads.

- Program 3: Mixed Chunk– Here, the matrix is first divided into chunks of size p = N/(K/2). Thus, we have K/2 chunks, as opposed to the K chunks we would have had in program 1. Each chunk is then assigned 2 threads which function the way

threads did in program 2. The two threads will calculate alternating lines within their own chunk. Hence, it combines the algorithms from both programs.

4. File Handling:

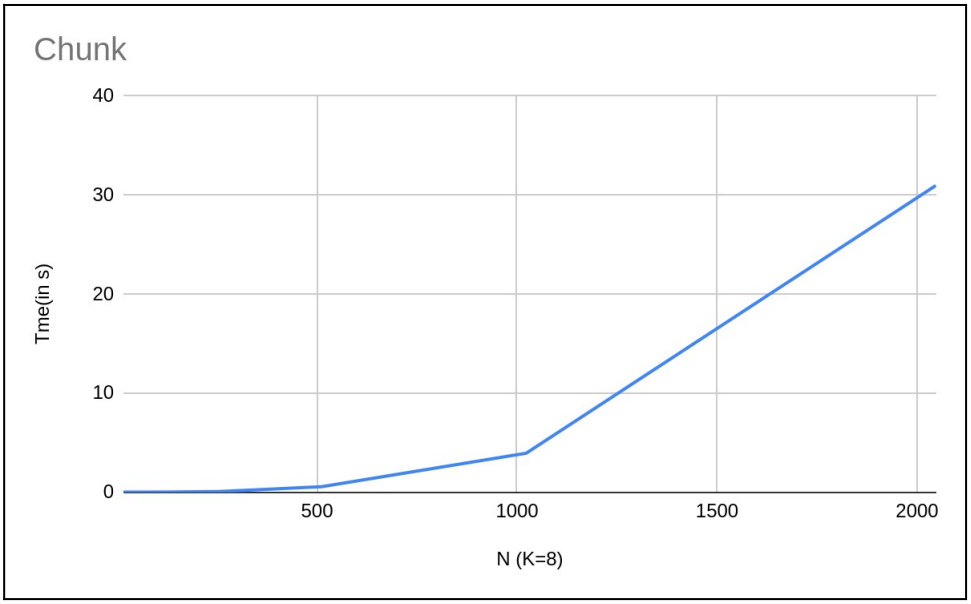
– The program uses file I/O to read the input matrix from `inp.txt` and write the result to the respective output file

5. Data Collected (All times are in microseconds)

➔ Fixing number of threads (K=8)

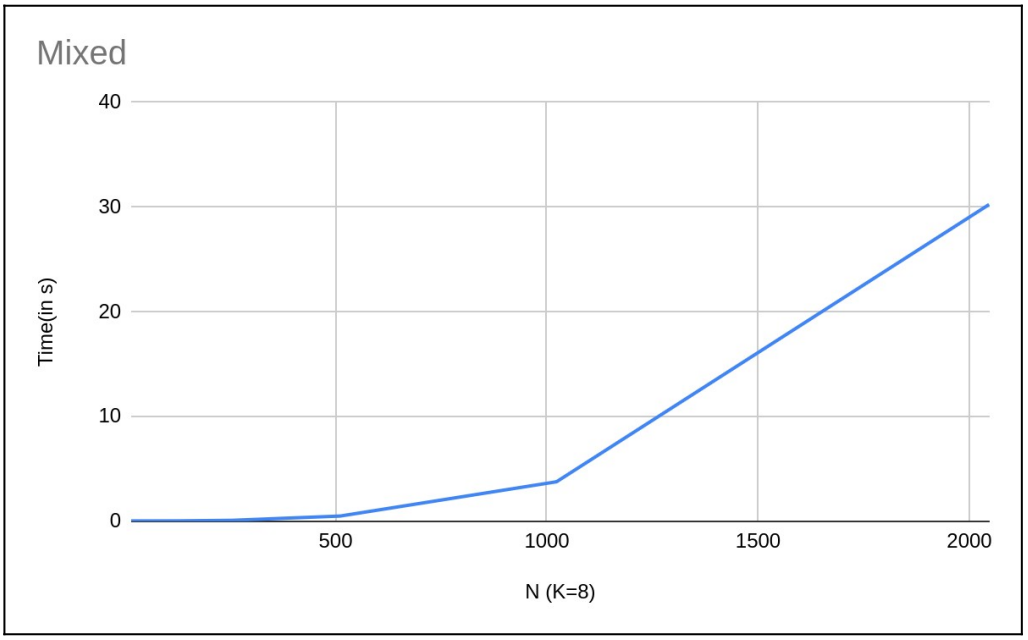
1) Chunk:

N (K=8)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
16	705	888	966	788	1050	879.4
32	1105	898	848	634	1315	960
64	1566	1491	1390	1431	1434	1462.4
128	6830	8391	8557	7063	7047	7577.6
256	57844	60218	58176	59011	56566	58363
512	517572	558294	523772	535037	568769	540688.8
1024	3838873	3973775	3798553	3895114	4107340	3922731
2048	30022977	30127635	32567837	29998765	31987123	30940867.4



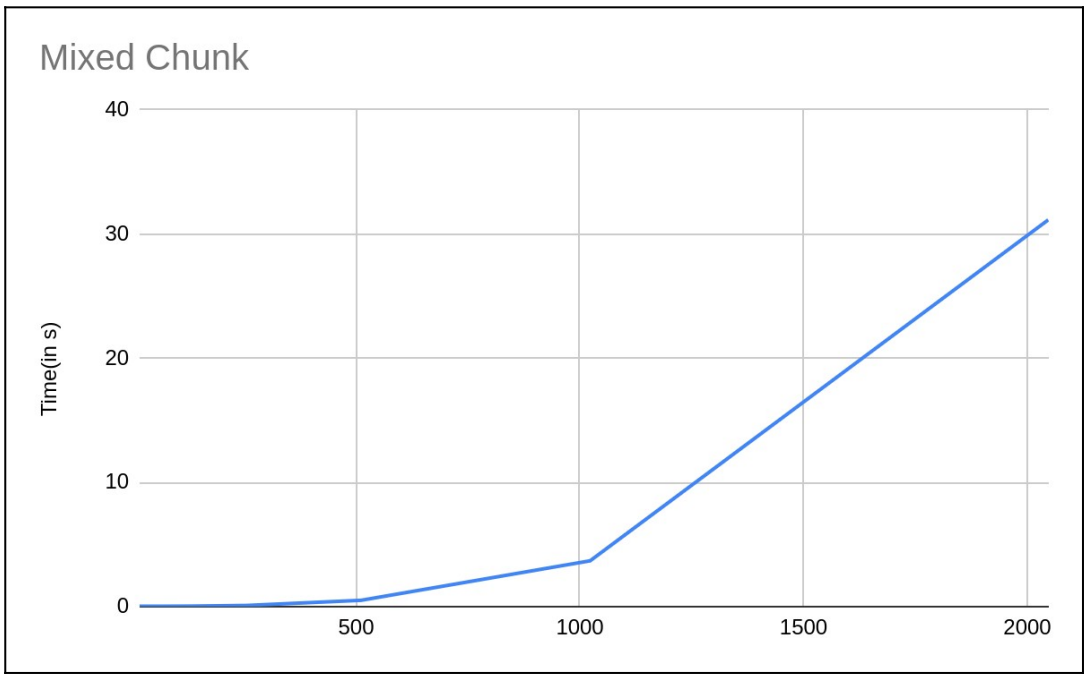
2) Mixed:

N (K=8)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
16	976	853	724	865	1068	897.2
32	1545	761	825	833	1210	1034.8
64	1369	1073	1935	1305	1212	1378.8
128	7197	7536	5340	7457	8342	7174.4
256	59211	57089	55772	55550	63862	58296.8
512	466655	459698	450461	496326	466412	467910.4
1024	3679861	3679500	3794327	3683017	3854222	3738185.4
2048	30081471	30068987	31092536	29807632	30001657	30210456.6



3)Mixed Chunk:

N (K=8)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
16	1226	1152	980	1225	1167	1150
32	1470	783	924	1038	953	1033.6
64	1792	1537	1477	1716	1518	1608
128	9011	7865	7854	8068	8244	8208.4
256	59149	61958	68195	64888	61339	63105.8
512	467377	498588	468666	498263	465145	479607.8
1024	3579357	3858617	3687559	3557339	3603928	3657360
2048	30148507	30017849	31457856	33970924	29971654	31113358

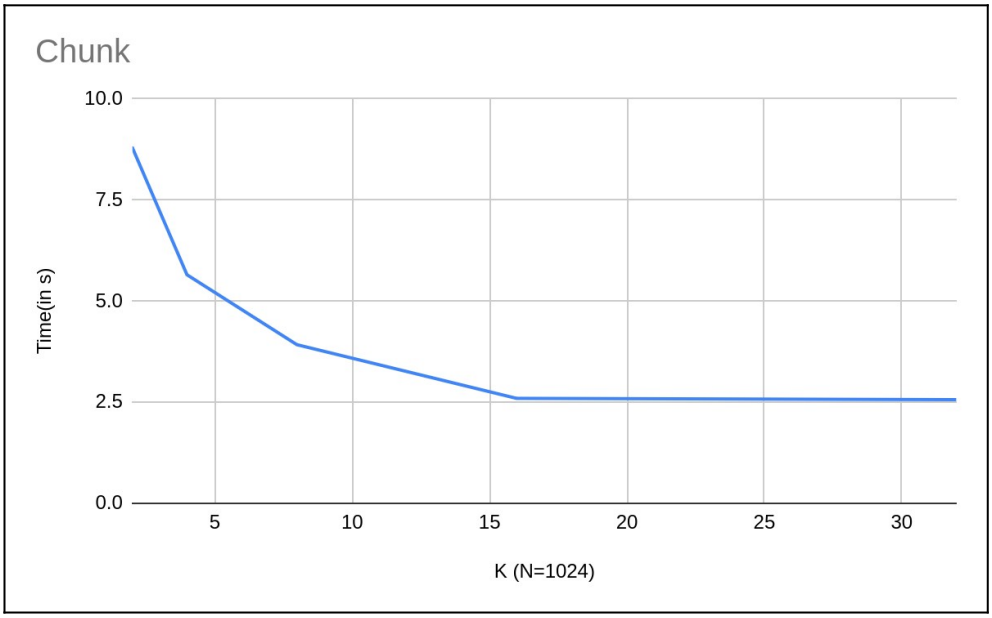


Analysis: All three methods lead to very similar results, which is not surprising since in each case, the distribution of workload is fairly even. The behaviour of the grpah makes sense too, since larger matrices will obviously take more time to evaluate, which is exactly what the data shows as well. Matrix multiplication is an $O(n^3)$ algorithm, which is also visible by the fact that the graphs aren't increasing linearly but with the slope constantly increasing with an increase in the size of N.

➔ Fixing size of matrix (N=1024)

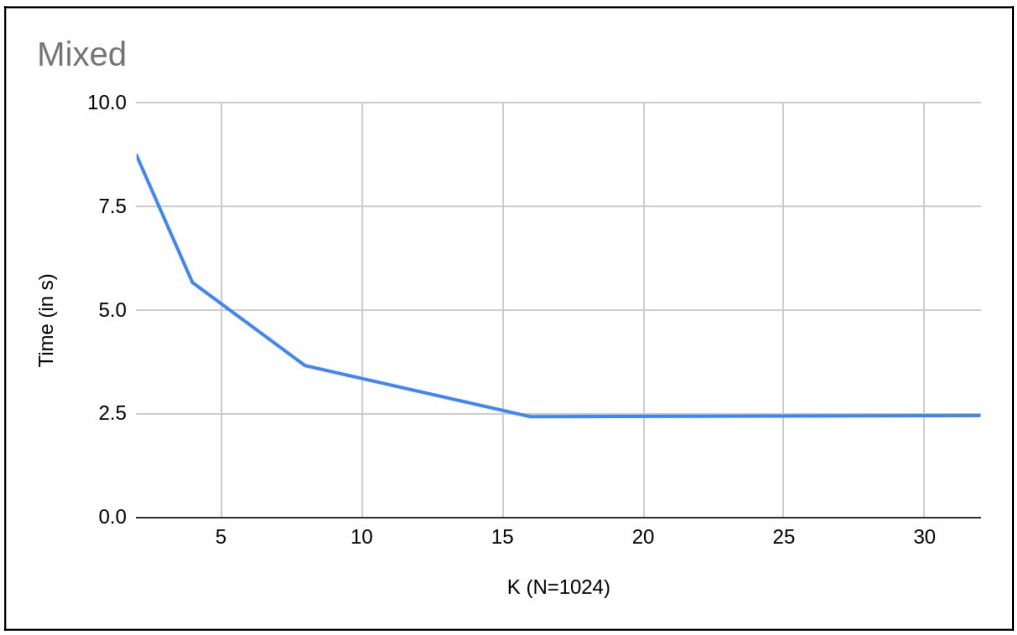
1) Chunk:

K (N=1024)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
2	8764263	8749976	8909765	8425629	9172674	8804461.4
4	5884499	5768116	5627864	5666287	5260909	5641535
8	3874642	4213256	3747880	3748158	3978938	3912574.8
16	2556028	2535784	2671991	2567308	2587616	2583745.4
32	2474076	2555873	2597930	2659841	2474928	2552529.6



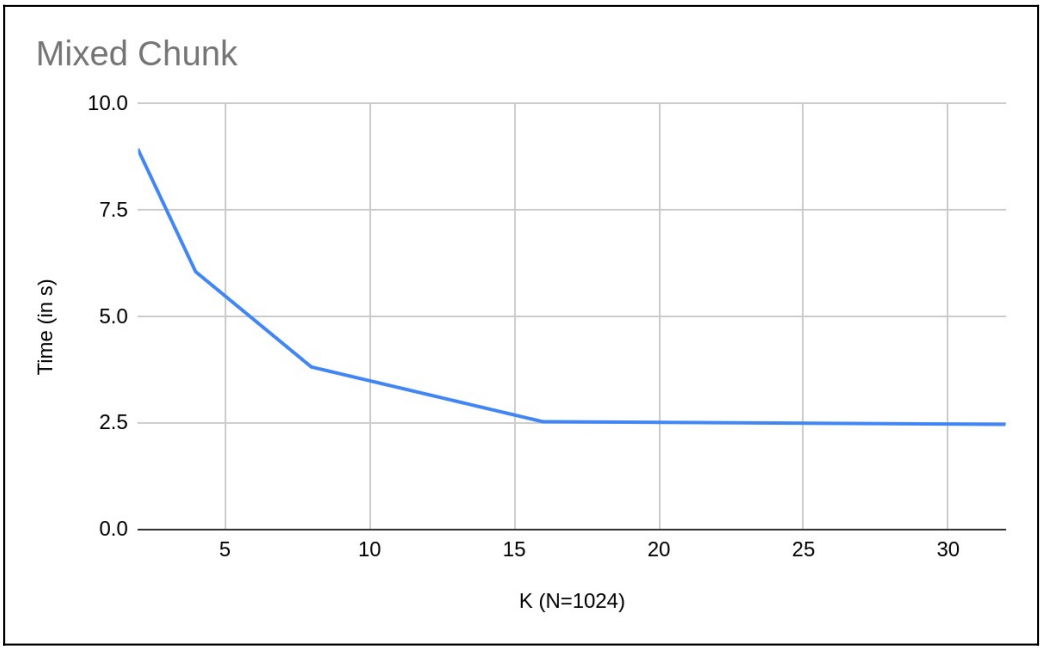
2) Mixed:

K (N=1024)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
2	8634810	8871467	8961828	9007156	8301896	8755431.4
4	5647069	5606797	5378926	5699735	5998163	5666138
8	3622909	3699026	3597884	3716815	3663406	3660008
16	2439326	2427856	2583060	2309339	2378450	2427606.2
32	2334317	2449228	2434243	2573553	2480546	2454377.4



3) Mixed Chunk:

K (N=1024)	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average
2	9126532	8756789	8589257	9171638	9000274	8928898
4	6030914	6117179	5997189	6209836	5871638	6045351.2
8	3810892	3721361	3837584	3854899	3807687	3806484.6
16	2540219	2621231	2416575	2487862	2545871	2522351.6
32	2592070	2446317	2388296	2515231	2355043	2459391.4



Analysis: Again, like the case with fixed threads, all three graphs show very similar nature which follows from the even workload distribution. The behaviour of the graph however is more interesting. Clearly, for a while, increasing the number of threads increases performance, which tracks from the fact that parallelism is taking place. However, it's clearly visible that the performance is hitting a limit, as we can see from the fact that it's more or less equal for K=16 and K=32. This is due to the fact that while increasing number of threads parallelizes the program, it also keeps increasing the overhead in creating and rejoining the threads. After a point, that overhead essentially neutralizes any performance benefits and thus, it hits a plateau.