Operating System -2 Programming Assignment 1

Name : Ahmik Virani
Roll Number : ES22BTECH11001

1. *Mixed technique for computing the C matrix in parallel*

Low Level Design :

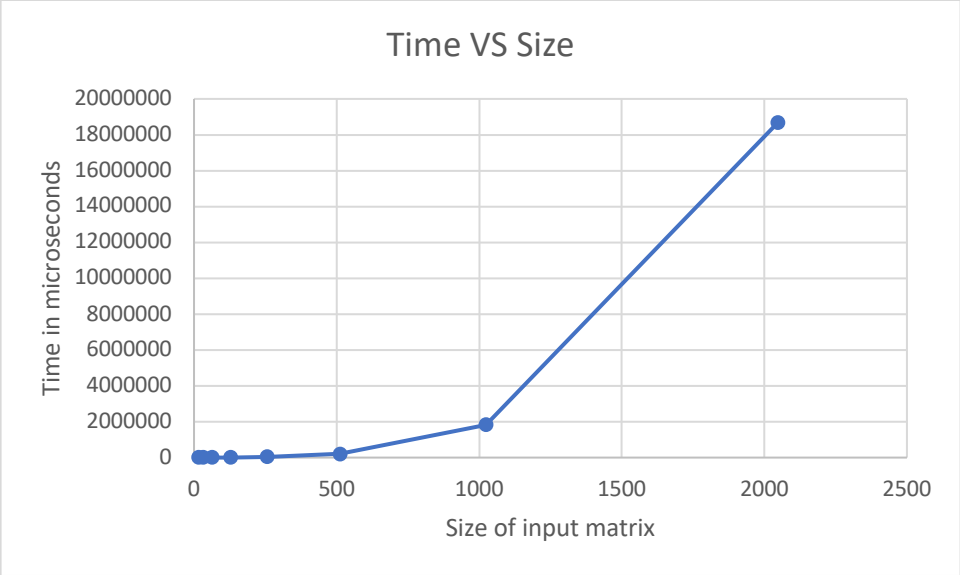
```
void *runner(void *param)
{
    int k = (*(int *)param);

    for(int i = k; i < N; i += K)
    {
        multiply(A, N, i);
    }

    pthread_exit(0);
}
```

The parameters N and K are as given in the problem statement.
k is the thread number I have passed in as the parameter
Then starting from k, I have calculated the k , k + K, k + 2K so on rows of the C matrix.

Performance of Algorithm :

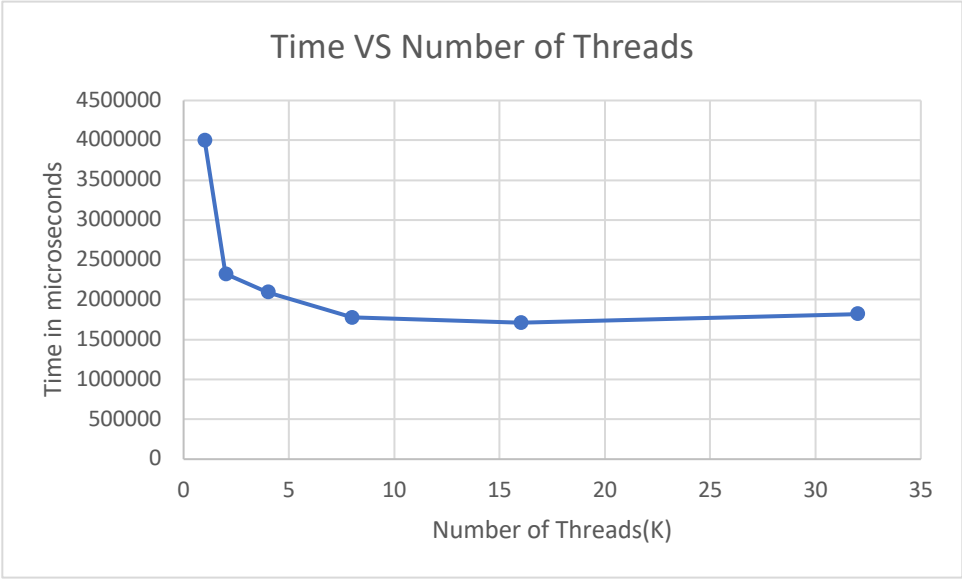
Time VS Size

N	Reading	1	2	3	4	5		avg
16		923	597	665	612	622		683.8
32		1226	2316	1033	1307	815		1339.4
64		1984	2007	3042	2699	2764		2499.2
128		7665	7013	7101	10879	7068		7945.2
256		36532	35527	33829	33750	48912		37710
512		196888	223971	231671	179013	213516		209011.8
1024		2021963	1638448	1741640	1890904	1808450		1820281
2048		18984143	21358775	15837642	21056420	16213352		18690066.4



Time VS Number of Threads

K	Reading	1	2	3	4	5		avg
1		3946955	4029186	4110636	3900746	4012811		4000066.8
2		2178692	2295322	2820395	2179880	2146151		2324088
4		1996122	1881865	1906481	1709239	2979137		2094568.8
8		1832810	1646584	1977891	1684439	1743751		1777095
16		1749594	1576512	1742618	1688927	1803741		1712278.4
32		1883251	1901955	1709023	1785786	1818108		1819624.6



2. *Chunks technique for computing the C matrix in parallel*

Low Level Design

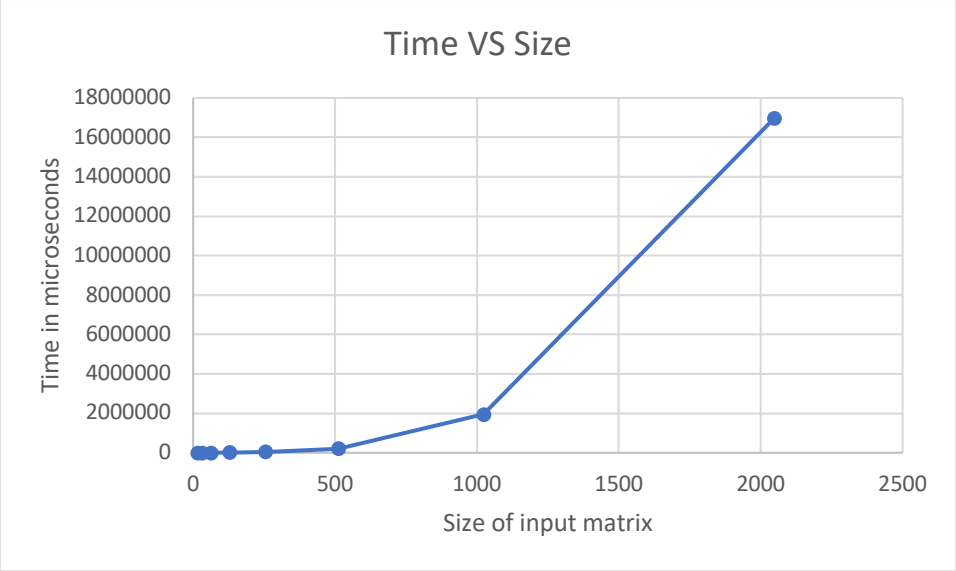
```
void *runner(void *param)
{
    int p;
    if(N < K)
    {
        p = 1;
    }

    else
    {
        p = N/K;
    }

    int k = (*(int *)param);

    if(k == K - 1)
    {
        for(int i = k * p ; i < N ; i++)
        {
            multiply(A, N, i);
        }
    }

    else
    {
        for(int i = k * p ; i < p * (k + 1) && i < N ; i++)
        {
            multiply(A, N, i);
        }
    }

    pthread_exit(0);
}
```

Here I am creating chunks of sizes N/K and solving for them in the result matrix.
The difference here is the use of the variable p.
If number of threads are more than number of rows in input matrix, then I am assigning each of the thread one row to compute in the resultant matrix.
Otherwise, as hinted in the problem statement, I am assigning each chunks of N/K, expect the last one, which I am assigning all the remaining rows.

Performance of Algorithm :

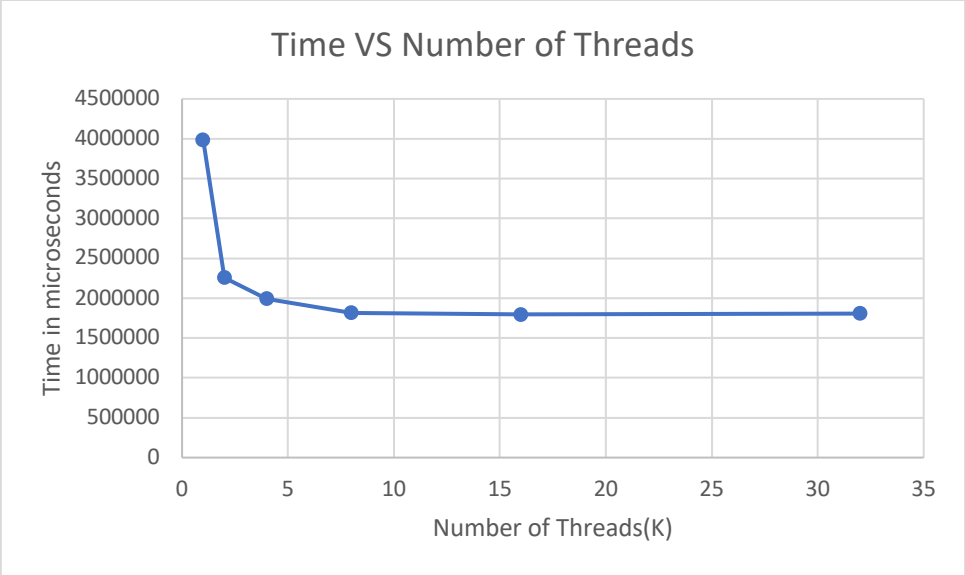
Time VS Size

N	Reading	1	2	3	4	5		avg
16		984	630	446	461	457		595.6
32		1094	1012	1679	1065	924		1154.8
64		2014	2828	2186	3993	2020		2608.2
128		7276	7430	9254	9925	9565		8690
256		36431	32867	52636	37545	46133		41122.4
512		258880	181540	171804	178076	238320		205724
1024		1903887	1840849	2209814	1919733	1878419		1950540.4
2048		15039703	16834188	16397321	21039368	15556827		16973481.4



Time VS Number of Threads

K	Reading	1	2	3	4	5		avg
1		4270995	3904939	3904192	3930246	3903887		3982851.8
2		2397824	2193162	2340883	2201299	2155058		2257645.2
4		2114744	2014275	2028670	1772413	2041578		1994336
8		1820352	1810721	1746898	1886777	1820586		1817066.8
16		1820102	1674873	1840289	1842873	1806058		1796839
32		1918226	1729032	1730907	1837217	1827133		1808503



3. *Extra Credit Idea*

Low Level Design

What my idea was that, instead of dividing work into rows, I am giving each thread elements over the matrix.
For example, consider the below matrix

1 2 3
4 5 6
7 8 9

Here if we have 4 threads, then thread 1 will compute elements labelled 1, 5, 9… thread 2 will compute elements labelled 2, 6 and so on.

```
void *runner(void *param)
{
    int dummyRow = 0;


    int k = (*(int *)param);

    for(int i = k; dummyRow < N ; i += K)
    {
        if(i >= N)
        {
            i = i % N;
            dummyRow++;
        }

        if(dummyRow == N)
```

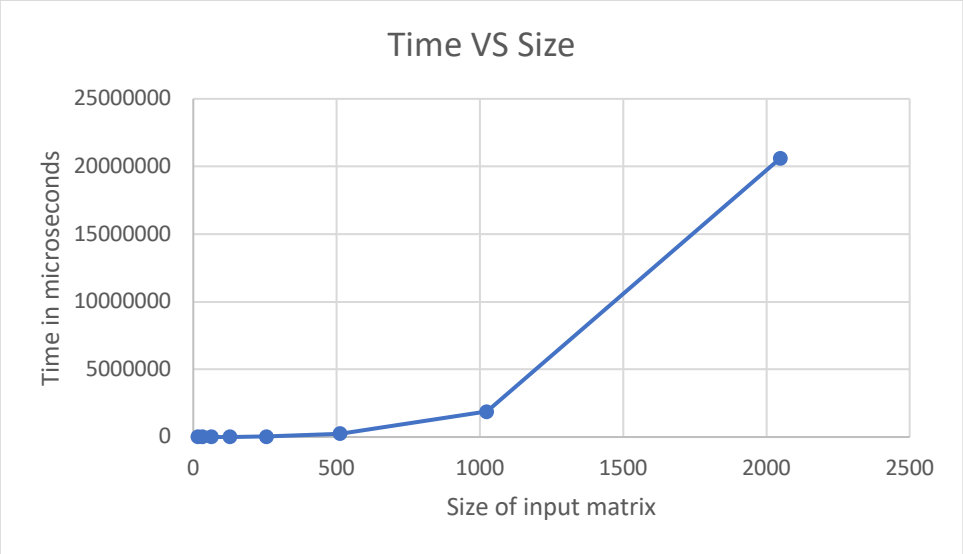
```
            break;

        multiply(dummyRow, i);
    }

    pthread_exit(0);
}
```

For the above mentioned reason, I have created a variable dummyRow. Basically it starts from first row (i.e. row = 0 in C++). As labelled in the above matrix, we start from say the first element 1. Now after this we add the number of threads (K), i.e. 1 + 4 = 5. Now we need to compute this, but first we need to locate where this is, so to do so we first check if 5 > N (number of rows), i.e. 5 > 3. If so, we just find the remainder when 5 divided by 3 and increase the current row by 1. Which means now we are in the second row and second element.

Performance of Algorithm :

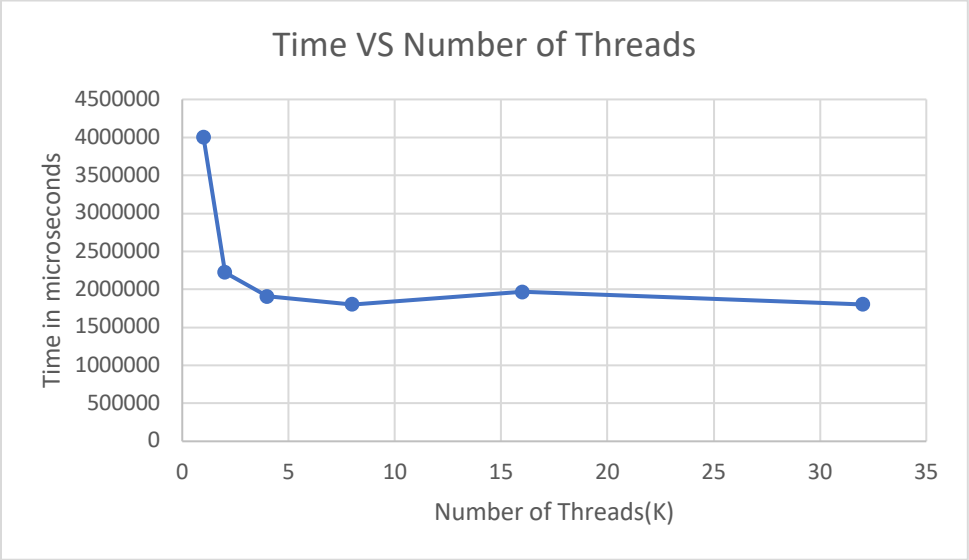
Time VS Size

N	Reading	1	2	3	4	5		avg
16		678	888	516	731	532		669
32		782	775	1086	1009	1217		973.8
64		2137	1982	2446	2086	3472		2424.6
128		9047	7285	7200	7153	9426		8022.2
256		33354	49531	37647	34251	37332		38423
512		184915	177748	326280	340915	221314		250234.4
1024		1827372	1931246	1785027	2012216	1870485		1885269.2
2048		18405507	19432452	21181412	21855437	22132170		20601395.6



Time VS Number of Threads

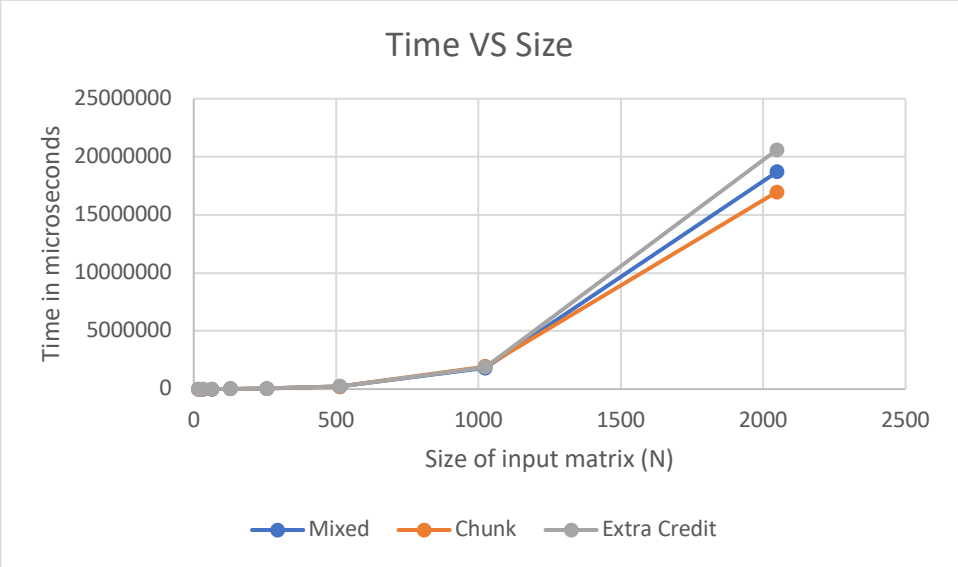
K	Reading	1	2	3	4	5		avg
1		3909797	3943921	3890941	4368524	3892980		4001232.6
2		2194144	2349396	2150024	2291920	2140550		2225206.8
4		2044478	1860299	1901582	1835608	1887807		1905954.8
8		1991142	1725853	1762444	1771453	1769923		1804163
16		1758443	1838882	2190643	1726115	2317046		1966225.8
32		1861522	1795540	1804716	1749941	1796510		1801645.8



Conclusion: As the input size increases, the time taken for the matrix multiplication also increases, whereas as the number of threads increases, the time taken first decreases first, then remains almost constant after 8 threads (The reason for this may be because my laptop has 8 cores, so is not optimal in providing 16 or 32 threads).

Comparison of algorithms :

1. In terms of Size :
    a. For smaller sizes all perfom almost similar
    b. however as size increases the performance is as follows : chunks < mixed < my ides



2. In terms of Number of Threads :
    a. All programs run in almost similar times overall