Assignment 1 : LU Decomposition

Aryan Dua
2020CS50475

Divyansh Mittal
2020CS10342

Harihar S
2020CS10878

1 Implementation

1.1 Sequential Base Code

We have chosen to represent π as an array of size n, and the matrices A, L, U by a contiguous array of size n^2. Note that using a vector of n pointers to n data elements would only help fasten up the `swap`$(A[k\ :], A[k'\ :])$ step. However, accessing elements through pointers introduces an additional level of indirection, which can slightly degrade performance, especially for random access patterns. Moreover, there is an increased memory overhead for storing the pointers. Since the above `swap` operation is relatively inexpensive, we have chosen to use the contiguous array representation.

1.2 Parallelization

Most of the parallelism in both our implementations arise due to the fact that most operations performed inside loops are independent of each other. The only step where this is not the case is the computation of the largest element in a column. To do this, we compute the maximum of a sequence of blocks using separate threads and finally combined their results.

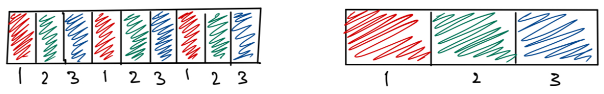

Figure 1: Memory Partition among Threads

We allocate regions of the memory to the threads in such a way that each thread has a contiguous block of memory and that these blocks are disjoint. This way of partition is particularly helpful in terms of cache locality since each thread would have higher cache hit rates. On the other hand, interleaving the threads would not exploit this spatial locality and also leads to false sharing since the threads modify data that reside on the same cache line. This division is done implicitly in OpenMP while we did it explicitly for pthreads.

In the pthread implementation, the decompose function can broadly be divided into 3 sub functions, which need to have barriers in between them due to sharing of data. These are:

1. **find_max():** This required keeping a global maximum variable guarded by a mutex, so that only one thread updates the global maximum with its local maximum at one time.
2. **swap_rows():** The n iterations of the loop were divided equally among the threads. The threads were all synchronised after the swap completed.
3. **update_matrices():** For each k, the $n - k + 1$ iterations of the loop were divided equally among the threads. The threads were all synchronised after the matrices were updated.

2 Analysis

An overview of the runtimes for $n = 8000$ is reported in the following table. As a baseline, the running time of the sequential implementation is 132.9 sec on $n = 8000$.

Statistics		
# Threads	OpenMP	pthreads
1	132.8 sec	133.1 sec
2	71.34 sec	76.79 sec
4	47.89 sec	49.22 sec
8	32.29 sec	49.07 sec
16	26.57 sec	47.30 sec

The running time of our sequential implementation in seconds were as follows

n	1000	2000	3000	4000	5000	6000	7000	8000
T_S	0.16	1.28	5.26	14.49	30.83	55.21	87.39	132.9

2.1 OpenMP

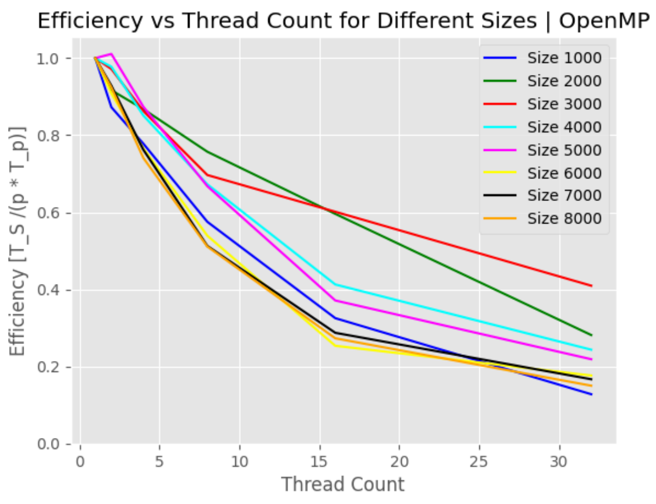

Figure 2: OpenMP Efficiency

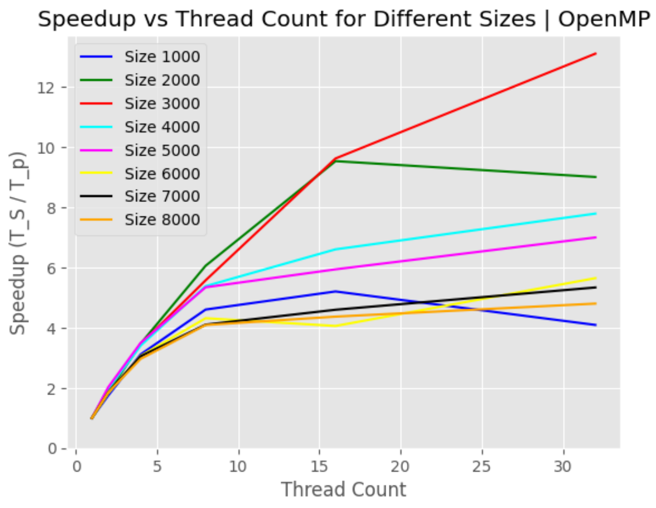

Figure 3: OpenMP Speedup

2.2 Pthreads

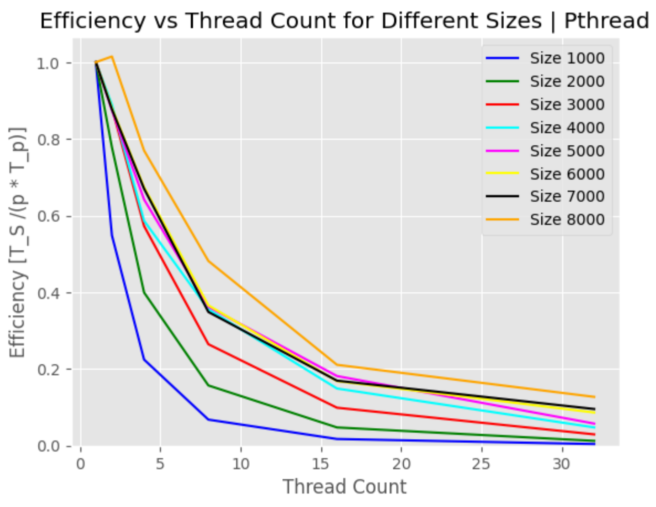

Figure 4: Pthreads Efficiency

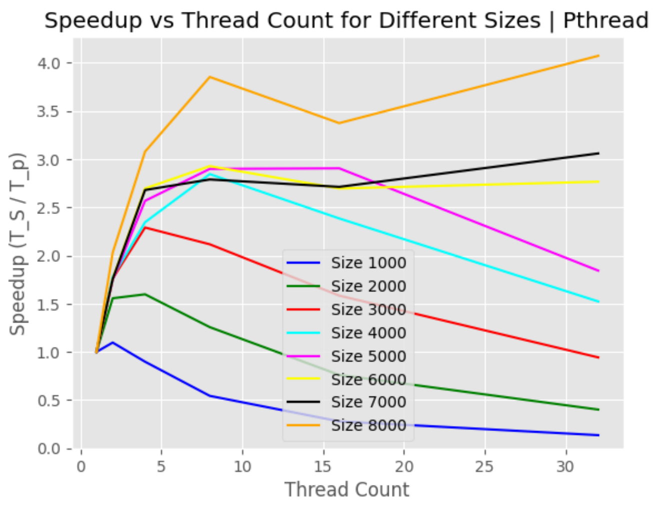

Figure 5: Pthreads Speedup

For pthreads, for smaller inputs, we get that initially the speedup increases as we increase the number of threads, but it quickly falls down as the number of threads get utilised. On the other hand, efficiency is consistently decreasing.

We know that initially spreading the work among the different threads makes us have a better speedup as the parallel execution time decreases. But with the increase in number of threads, the thread creation, synchronisation and communication overheads outweigh the benefit of parallelisation of our code for smaller sizes of n. With the increase in the size of n, and having more work, we get that the benefit of more threads is apparent as now we get a higher speedup.

3 Statistics

Table 1: Time(in s) for Different Thread Counts and Input Sizes OpenMP

Thread Count Size	1	2	4	8	16	32
1000	0.17	0.09	0.05	0.04	0.03	0.04
2000	1.29	0.70	0.37	0.21	0.14	0.14
3000	5.43	2.79	1.57	0.97	0.56	0.41
4000	14.68	7.50	4.31	2.73	2.22	1.88
5000	32.52	16.08	9.30	6.09	5.48	4.65
6000	54.57	30.04	17.82	12.65	13.45	9.67
7000	88.24	47.51	28.94	21.51	19.20	16.54
8000	133.46	72.23	45.02	32.64	30.56	27.80

Table 2: Time(in s) for Different Thread Counts and Input Sizes Pthreads

Thread Count Size	1	2	4	8	16	32
1000	0.24	0.22	0.27	0.45	0.87	1.80
2000	1.47	0.94	0.92	1.17	1.93	3.65
3000	5.61	3.18	2.45	2.65	3.54	5.95
4000	15.09	8.48	6.43	5.30	6.33	9.90
5000	31.00	17.72	12.07	10.69	10.67	16.81
6000	55.33	31.41	20.52	18.90	20.52	20.00
7000	88.46	50.42	33.01	31.71	32.61	28.91
8000	158.94	78.31	51.63	41.25	47.10	39.03