COL 380 - A1

Manshi Sagar - 2020CS50429
Rajat Mahaur - 2020CS50534
Richa Yadav - 2020CS50438

15 February 2024

1 Introduction

In this assignment, we developed two parallel implementations of LU decomposition that use Gaussian elimination to factor a dense N x N matrix into an upper and lower-triangular one. In matrix computations, pivoting involves finding the largest magnitude value in a row, column, or both and then interchanging rows and/or columns in the matrix for the next step in the algorithm. The purpose of pivoting is to reduce round-off error, which enhances numerical stability. In our assignment, used row pivoting, a form of pivoting that involves interchanging rows of a trailing submatrix based on the largest value in the current column. To perform LU decomposition with row pivoting, computed a permutation matrix P such that PA = LU.

2 Serial Implementation

In the serial version, we implement LU decomposition with partial pivoting for square matrices in parallel using C++ threads. After LU decomposition, we also compute the L2,1 norm of the difference between the original matrix and the recomputed matrix using LU decomposition.

1. **lu_decomposition() function:** Takes a square matrix a and a permutation vector *pi* as input. Performs parallel LU decomposition with partial pivoting, ie, calculates L and U, and modifies the permutation vector in place. Returns the product of L and U matrix, ie, *luprod* matrix. Includes timing measurement and output.
2. **computeL21Norm() function:** Calculates the L2,1 norm of the input square matrix matrix. Iterates through each column, computes the sum of squares of elements, and takes the square root. Returns the L2,1 norm.
3. **main() function:** Parses command-line arguments for matrix size and number of threads. Creates a random square matrix A_original. Calls *lu_decomposition()* to get the LU product *luprod*. Constructs the recomputed A matrix *A_permuted* using *pi*. Calculates the difference matrix diff and its L2,1 norm. Returns the L2,1 norm.

3 Pthreads

3.1 Implementation

For this, we parallelize the loops that usually don't contain data dependency. Apparently we divide work among threads according to the rank.

1. Initially, shared memory for all threads is defined in the pthreads implementation.
2. Global variables such as *a_global*, *l_global*, *u_global*, and *pi_global* are defined to represent the matrices A, L, U, and the permutation vector *pi* respectively. Additionally, a global variable *k_global* is defined to represent the current column iteration in the *lu_decomposition* function. These matrices are used in all target functions of pthreads, so making these global matrices helps these functions to access the matrices.

3. In *lu_decomposition* function, we parallelized loops that didn't contain data dependencies using pthreads and made separate functions for each loop. We divided the work equally among each thread using thread ranks. For example, if 4 threads are running a for-loop of 100 iterations, each thread performs 25 iterations parallely.
4. In the first parallelized loop, each thread calculates the local maximum value in its assigned chunk of columns. Mutex is used to compare and update the global maximum value max_val and its index k_prime.
5. Subsequent loops involve swapping rows, which are divided among threads based on their ranks. Since threads operate on different memory areas, mutex is not required here.
6. The assignment of values to the entries in L and U matrices is also parallelized. Threads are assigned chunks of columns (for L) and rows (for U), and they update their respective parts independently before joining back.
7. The assignment of the input matrix A involves **optimizing cache utilization** by restructuring nested for-loops into a single for-loop. This makes work division among threads easy. Each thread performs work in row-major form and cache also works efficiently.
8. Finally, the L21 norm is calculated, and the results are returned.

3.2 Experiments and Observations

1. We observed that increasing the number of threads did not always result in less execution time. In almost all cases (n=10 to n=4500), the program took minimum time with 2 threads.
2. An interesting observation is that for n around 4000-5000, 16 threads took the minimum time. This is probably because the input size is very high, and since each thread does totalWork/t amount of work, for large n, if there are more number of threads, the load on each thread is decreased.
3. We split the following for-loop into 2 for-loops (2.1 and 2.2) as there was no dependency among the 2 statements in this loop

```
1: for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)
v/s
1.1: for i = k+1 to n
      l(i,k) = a(i,k)/u(k,k)
1.2: for i = k+1 to n
      u(k,i) = a(k,i)
```

4. Also, instead of joining threads of the first part of the for loop before starting the second part, we tried joining threads of both parts together. This showed better results as one barrier from the code was removed.

```
v1:
    start threads(1.1)
    join threads(1.1)
    start threads(1.2)
    join threads(1.2)
    v/s
v2:
    start threads(1.1)
    start threads(1.2)
    join threads(1.1)
    join threads(1.2)
```

For example, v1(1400, 2) = 10671 msec , v2(1400, 2) = 7009 msec
v1(700, 2) = 1579 msec , v2(700, 2) = 1168 msec

4 OpenMP

4.1 Implementation

1. The timing trend was: **serial>pthreads>OpenMP**

2. Optimizing cache utilization by restructuring nested loops into a single loop. This significantly increased the cache hits and thus reduced the time.

3. We divided the assignment of l and u matrices into two different areas and thus helped with loop parallelization and cache use. Using two different memory locations will unnecessarily clutter the cache and cause cache misses. This also increased the speed and simplicity of work diving into threads.

4. We join threads for these two loops parallelized in the assignment of l and u together as they are independent in the data structure, and there is no data dependency.

5. Divide work equally according to thread ID increase speed.

6. Mutex is used in only one loop where we find k_prime as elsewhere there is just reading and no data dependency while writing.

7. We also free thread handles after their work is done for no segmentation faults or risks.

4.2 Experiments and Observations

1. We first run openmp using **Static scheduling** with chunk size of ($size/num\_of\_threads$). This implementation worked better than most of the other scheduling methods.

2. We also tried **Dynamic scheduling** with chunk size of ($size/2 * num\_of\_threads$) but its computation was quite slow for less no. of threads with comparison to static scheduling and a little slow for higher no. of threads. For eg, Dynamic(1400, 2)= 5283 milisec Static(1400, 2)= 3969 milisec ; Dynamic(1400, 16)= 2077 milisec Static(1400, 16)=1812 milisec.

3. We tried **Runtime scheduling** and it was very slow for all values. For eg, Static(1400, 16)= 1812 milisec Runtime(1400, 16)= 5615 milisec.

4. For **Auto scheduling**, most of the results were comparable with the static implementation but a little slow. So, we tried implementing combination of static and auto scheduling by using static mode for loops which had iterations from 0 to n and auto for loops which had iterations from k to n or similar, but the implementaion was slow with comparison to static scheduling. For eg, Static+Auto(3500, 4)= 71479 milisec Static(3500, 4)= 70078 milisec ; Static+Auto(4900, 4)= 208691 milisec Static(4900, 4)= 182820 milisec.

5. We then **finally** implemented **Static** scheduling with a chunk size of min(max((size/num_of_threads), 2), 100), so we limited the chunk size between (2, 100) so that if $size/num\_of\_threads$ is too large then smalls chunks are divided between threads and if it is too small like in the case of (size=10, threads=12) then we assign 2 threads to divide the work. We got fastest results for this implementation. For eg, Static(with varying chunk size)(2100, 16)= 4913 milisec Static(2100, 16)= 6377 milisec ; Static(with varying chunk size)(5600, 6)= 162 milisec Static(5600, 6)= 199510 milisec.

5 Tables and Graph

Note: The * values were not collected due to laptops overheating, causing extended processing times.

Table 1: Serial Timing wrt Size of the matrix (for Pthreads)

Size:	Time (in msec):
n=10	0
n=100	9
n=400	269
n=700	1346
n=1400	10493
n=2100	35782
n=2800	81963
n=3500	156386
n=4200	356440
n=4900	508152
n=5600	694226
n=6300	*
n=7000	*

Table 2: Pthread Timing with respect to the Size of the Matrix

Size	Time (in msec) with respect to the number of threads							
	2	4	6	8	10	12	16	best
$n = 10$	4	13	19	26	16	21	32	4 (t=2)
$n = 100$	41	73	105	116	163	175	238	41 (t=2)
$n = 400$	348	460	560	703	971	897	1098	348 (t=2)
$n = 700$	1168	1568	1688	1964	2134	2424	2756	1168 (t=2)
$n = 1400$	7009	9791	9358	10582	10609	11280	12860	7009 (t=2)
$n = 2100$	22788	33988	31196	33961	37205	36396	37564	22788 (t=2)
$n = 2800$	55704	81185	83956	92326	81162	81635	81227	55704 (t=2)
$n = 3500$	111203	166737	152007	148342	156413	149234	157200	111203 (t=2)
$n = 4200$	210702	260222	250267	250588	251043	254512	223455	210702 (t=2, 16)
$n = 4900$	361304	439130	468533	425705	393719	375040	334502	334502 (t=16)
$n = 5600$	524399	523459	533400	513459	513705	508985	490879	490879(t=16)
$n = 6300$								
$n = 7000$								

Table 3: Serial Timing wrt Size of the matrix

Size:	Time (in msec):
n=10	0
n=100	1
n=400	111
n=700	565
n=1400	4640
n=2100	15838
n=2800	38369
n=3500	110692
n=4200	128834
n=4900	207996
n=5600	544776
n=6300	784587
n=7000	1129738

Table 4: OpenMP Timing wrt Size of the matrix

Size:	Time (in msec) wrt to number of threads							
	threads=2	threads=4	threads=6	threads=8	threads=10	threads=12	threads=16	Best
n=10	1	5	10	8	9	3	10	t=2
n=100	84	34	41	45	62	62	53	t=4
n=400	256	425	199	207	211	193	248	t=12
n=700	411	275	379	314	280	282	353	t=4
n=1400	3510	2068	2525	2000	1735	1538	1536	t=16
n=2100	11322	7336	8175	6399	5625	5274	4913	t=16
n=2800	28163	21026	20252	18626	13095	12326	11200	t=16
n=3500	58597	37589	41009	37330	33000	30378	27309	t=16
n=4200	105084	72530	70435	87524	57252	51258	45458	t=16
n=4900	157341	109797	110425	97038	90643	79245	72445	t=16
n=5600	232519	168668	162638	141116	132372	117925	110174	t=16
n=6300	646389	407982	320475	271733	329370	284288	160501	t=16
n=7000	956439	541540	526734	491015	508956	457822	295368	t=16

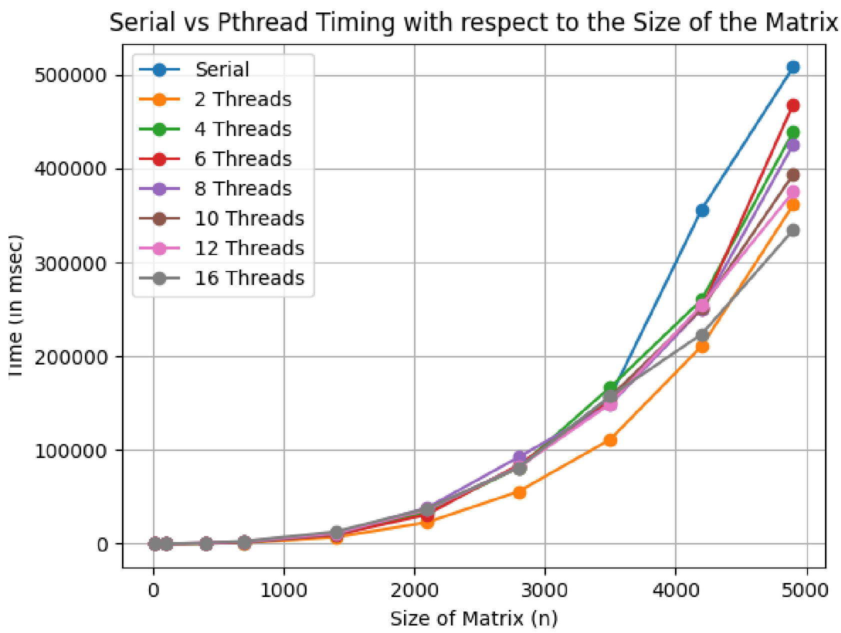

Figure 1: Serial vs Pthreads

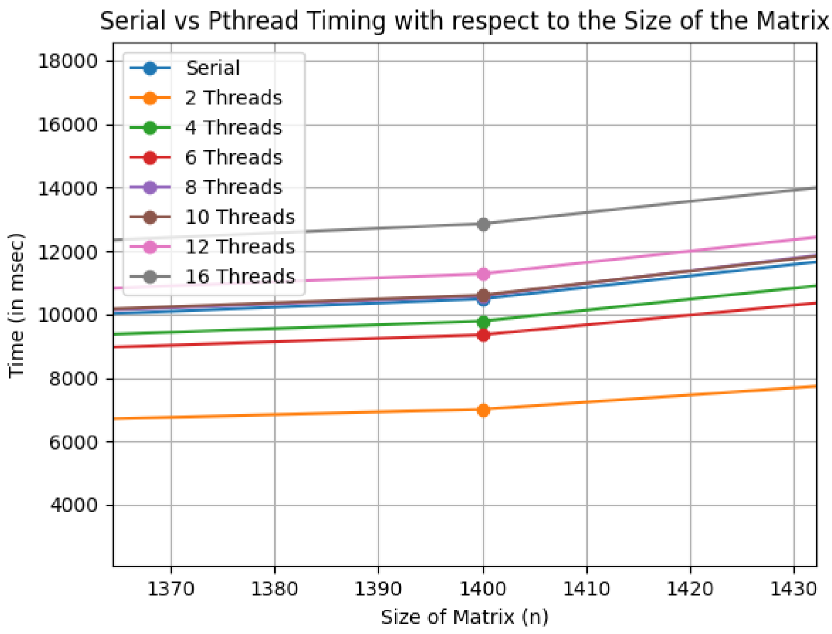

Figure 2: Serial vs Pthreads deviation

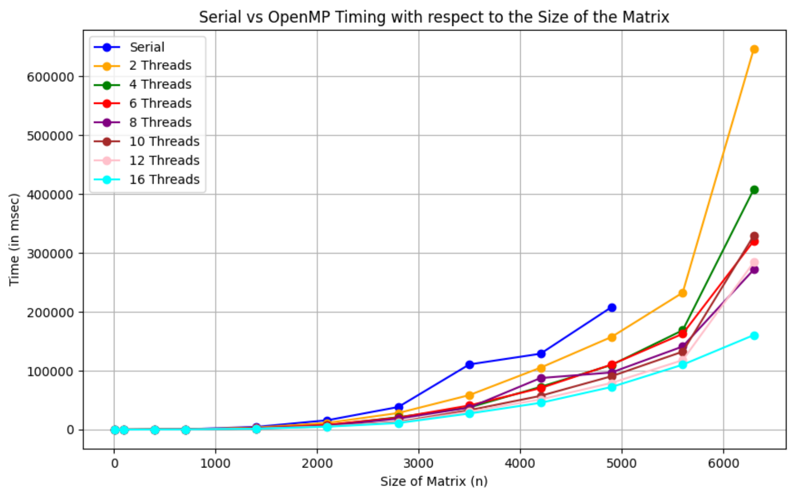

Figure 3: Serial vs OpenMP

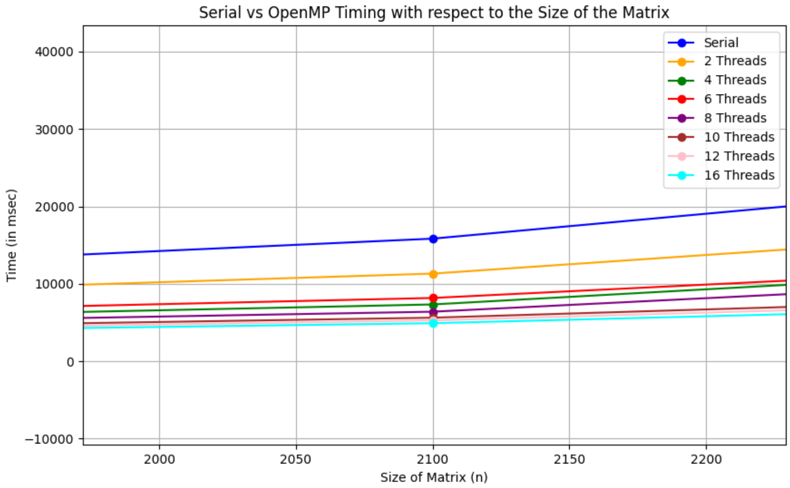

Figure 4: Serial vs OpenMP deviation

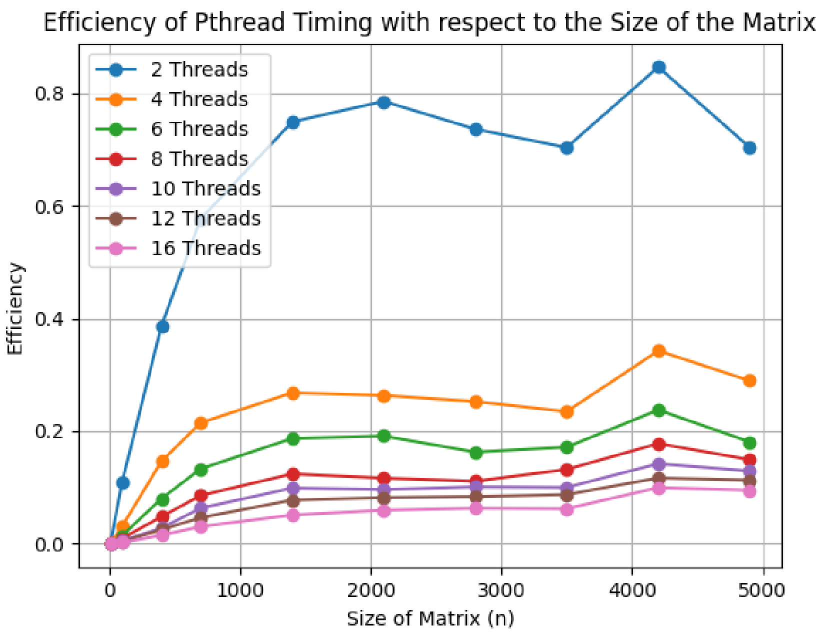

Figure 5: Serial vs Pthreads Efficiency

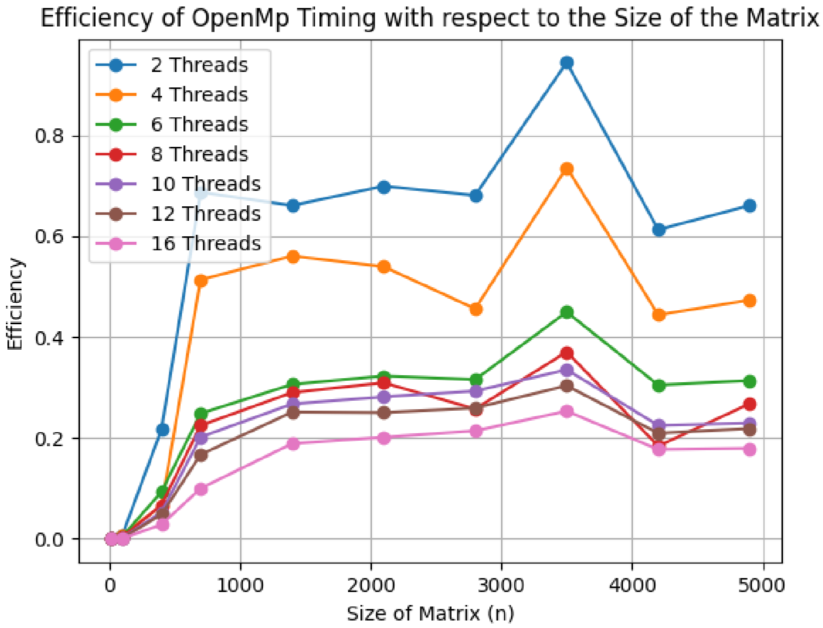

Figure 6: Serial vs OpenMP Efficiency

Figure 7: Serial vs Best Pthreads

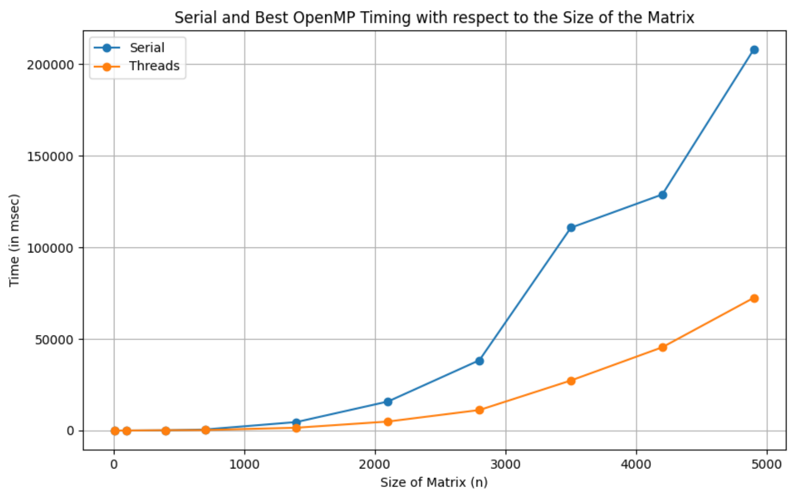

Figure 8: Serial vs Best OpenMP

Figure 9: Serial vs Best Pthreads Efficiency

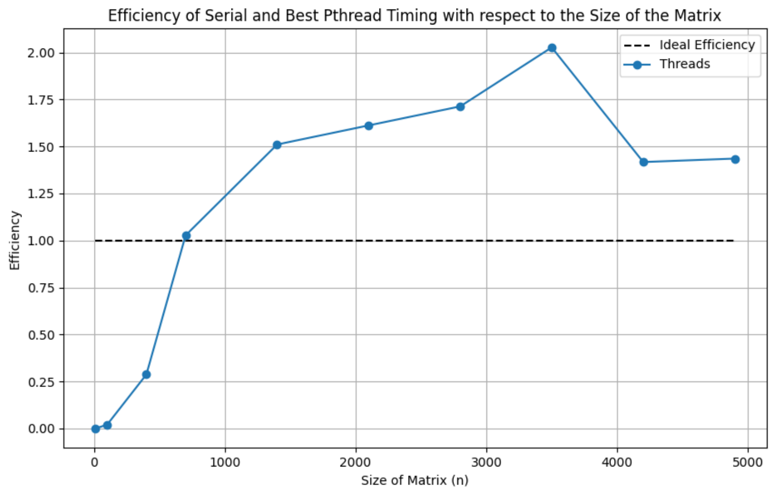

Figure 10: Serial vs Best OpenMP Efficiency

6 Conclusion

1. Efficiency comparison: $OpenMP(2.0) > Pthread(0.9)$
2. Time comparison: $OpenMP < Pthread < Serial$
3. OpenMP code offers easier implementation and achieves the fastest execution.