Mini Project 1: Parallel LU Decomposition

20200703 SOONHO KIM

1. Program Structure

Matrix Allocation:

The matrix is represented as a 2D dynamic array (double**) and allocated via a helper function `allocate_matrix(n)`, which returns an n x n matrix of 64-bit floating-point values.

Random Initialization:

Matrix elements are initialized with uniformly distributed random numbers in [0, 1], using C++11's <random> library. To support parallel generation, each thread uses its own instance of mt19937, seeded with `std::mt19937 rng(time(NULL) + tid);`

This ensures thread-safe and reproducible randomness across threads under OpenMP.

LU Decomposition Function:

The decomposition follows handout pseudocode and includes the following steps:

1. Pivot Selection:
   Find the row with the maximum absolute value in the current column to enhance numerical stability.
2. Row Swapping:
   Swap the current row with the pivot row in both the matrix and the permutation vector.
3. Multiplier Computation:
   Compute values for the lower triangular matrix LLL using:
   $$L(i,k) = \frac{A(i,k)}{U(k,k)}$$
4. Submatrix Update (Schur Complement):
   Update the remaining submatrix entries with
   $$A(i,j) = A(i,j) - L(i,k) * U(k,j)$$

This step reduces the original matrix to upper and lower triangular form in-place.

2. Parallelization Strategy

The LU decomposition is parallelized using OpenMP, primarily through for-based directives to leverage loop-level parallelism.

- **Parallel Directives Used:**
  1. `#pragma omp parallel for`: for standard loop parallelization.
  2. `#pragma omp parallel for collapse(2)`: for nested loop optimization, particularly when updating submatrices.
- **Parallelization Points:**
  1. Matrix Initialization in `main()`: threads generate random values in parallel.
  2. L Calculation Loop in `lu_decomposition()`: each row update is independent.
  3. Schur Complement Update in `lu_decomposition()`: nested loops updating submatrix elements are parallelized.
- **Data Partitioning and Synchronization:**
  Since all parallel regions are loop-based, the workload is implicitly partitioned along i and j loop indices. There are no shared resource conflicts due to row-wise memory access, and false sharing is minimized.

3. Experimental Setup

The program was tested under two hardware environments using the same code:

- Test Platforms:
  - AMD Ryzen 5 5600G (6 cores / 12 threads, 16MB L3 cache)
  - Intel Core i7-1360P (4 performance + 8 efficient cores, 16 threads, 18MB L3 cache)
- Parameters:
  - Matrix sizes: $N \in \{1000, 2000, 4000, 8000\}$
  - Number of threads: $p \in \{1, 2, 4, 8, 12, 16\}$
  - Each configuration was repeated 5 times and averaged.
- Metrics Collected:
  - Execution time $T(p)$
  - Residual $\| PA - LU \|_{2,1}$

4. Performance Evaluation

Before measuring all performance data, I first verified the correctness of the LU decomposition as instructed in the handout.

Specifically, I computed the **L2,1 norm** of the residual matrix, defined as the sum of Euclidean norms of the columns of PA−LU. The result showed **very small values in the range of 10^{-10}~10^{-11}**, which confirmed the numerical correctness.

Based on this, I proceeded to measure the LU decomposition time across various test cases.

MatrixSize	Threads	Type	Time(s)	Residual
1000	1	Serial	0.152936	4.17E-11
1000	1	Serial	0.142234	4.24E-11
1000	1	Serial	0.146328	4.21E-11
1000	1	Serial	0.143175	4.18E-11
1000	1	Parallel	0.181118	4.18E-11
1000	1	Parallel	0.18023	4.13E-11
1000	1	Parallel	0.189719	4.16E-11
1000	1	Parallel	0.188166	4.19E-11
1000	2	Parallel	0.100916	4.15E-11
1000	2	Parallel	0.0952335	4.14E-11
1000	2	Parallel	0.0960171	4.14E-11
1000	2	Parallel	0.0926738	4.15E-11
1000	4	Parallel	0.0818114	4.20E-11
1000	4	Parallel	0.0692526	4.20E-11
1000	4	Parallel	0.0716118	4.20E-11
1000	4	Parallel	0.0797535	4.23E-11
2000	1	Serial	2.5869	2.04E-10
2000	1	Serial	2.55105	2.04E-10
2000	1	Serial	2.22259	2.05E-10
2000	1	Serial	2.46416	2.05E-10
2000	1	Parallel	2.9385	2.03E-10
2000	1	Parallel	2.71701	2.03E-10
2000	1	Parallel	2.59234	2.03E-10
2000	1	Parallel	2.51985	2.02E-10
2000	2	Parallel	1.9152	2.04E-10
2000	2	Parallel	2.17923	2.03E-10
2000	2	Parallel	1.88516	2.04E-10
2000	2	Parallel	1.7937	2.04E-10
2000	4	Parallel	1.48389	2.03E-10
2000	4	Parallel	1.73146	2.03E-10
2000	4	Parallel	1.67667	2.04E-10
2000	4	Parallel	1.63847	2.04E-10

MatrixSize	Threads	Type	Time(s)	Residual
1000	8	Parallel	0.053539	4.19E-11
1000	8	Parallel	0.054411	4.19E-11
1000	8	Parallel	0.055908	4.16E-11
1000	8	Parallel	0.063236	4.16E-11
1000	12	Parallel	0.353676	4.20E-11
1000	12	Parallel	0.124319	4.16E-11
1000	12	Parallel	0.109525	4.19E-11
1000	12	Parallel	0.049384	4.19E-11
2000	8	Parallel	1.26949	2.03E-10
2000	8	Parallel	1.69776	2.03E-10
2000	8	Parallel	1.50775	2.04E-10
2000	8	Parallel	1.66659	2.04E-10
2000	12	Parallel	1.58414	2.03E-10
2000	12	Parallel	2.10247	2.03E-10
2000	12	Parallel	1.93013	2.03E-10
2000	12	Parallel	1.34326	2.04E-10

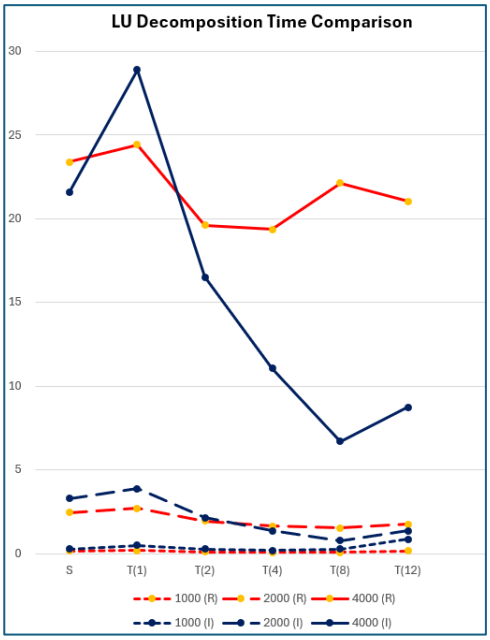
Table1. LU decomposition Efficiency

p		1			2			4			8			12			16		
N	S	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf
AMD Ryzen 5 5600G (specifications: 6P/12Threads, L3 cache 16MB)																			
1000	0.1462	0.1848	0.8	0.79	0.0962	1.5	0.76	0.0756	1.9	0.48	0.0568	2.6	0.32	0.1592	0.9	0.08	-		-
2000	2.4562	2.6919	0.9	0.91	1.9433	1.3	0.63	1.6326	1.5	0.38	1.5354	1.6	0.20	1.7400	1.4	0.12	-		-
4000	23.3884	24.4122	1.0	0.96	19.6104	1.2	0.60	19.3466	1.2	0.30	22.1187	1.1	0.13	21.0237	1.1	0.09	-		-
8000	179.0600	190.4000	0.9	0.94	155.3750	1.2	0.58	158.0830	1.1	0.28	169.5430	1.1	0.13	180.6020	1.0	0.08	-		-
13th Gen Intel(R) Core(TM) i7-1360P (specifications: 4P+8E/16Threads, L3 cache 18MB)																			
1000	0.2747	0.4764	0.6	0.58	0.2603	1.1	0.53	0.1909	1.4	0.36	0.2640	1.0	0.13	0.8486	0.3	0.03	1.2225	0.2	0.01
2000	3.2924	3.8763	0.8	0.85	2.1246	1.5	0.77	1.3644	2.4	0.60	0.7708	4.3	0.53	1.3588	2.4	0.20	1.4993	2.2	0.14
4000	21.5544	28.8771	0.7	0.75	16.5034	1.3	0.65	11.0607	1.9	0.49	6.7043	3.2	0.40	8.7390	2.5	0.21	10.1280	2.1	0.13
8000	174.8510	224.3920	0.8	0.78	127.6360	1.4	0.68	82.8351	2.1	0.53	51.1176	3.4	0.43	67.2453	2.6	0.22	74.0107	2.4	0.15

The data I measured are as follows. In consideration of the number of cores and threads of each CPU, Ryzen measured only up to p=12.

All graphs and data described later are based on the data in Table 1.

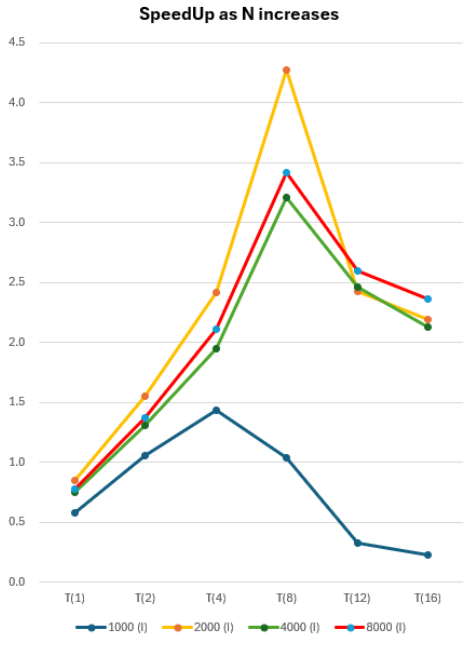
Table2. LU Decomposition Time Comparison (for each matrix size N and thread count p)



	S	T(1)	T(2)	T(4)	T(8)	T(12)
1000 (R)	0.1462	0.1848	0.0962	0.0756	0.0568	0.1592
2000 (R)	2.4562	2.6919	1.9433	1.6326	1.5354	1.7400
4000 (R)	23.3884	24.4122	19.6104	19.3466	22.1187	21.0237
1000 (I)	0.2747	0.4764	0.2603	0.1909	0.2640	0.8486
2000 (I)	3.2924	3.8763	2.1246	1.3644	0.7708	1.3588
4000 (I)	21.5544	28.8771	16.5034	11.0607	6.7043	8.7390

The left column indicates the hardware used: **R (Ryzen)** and **I (Intel)**, along with the corresponding matrix size N. The horizontal entries show **S (Serial time)** and **T(p)** in seconds.
On Ryzen, the time reduction from parallel execution is **less significant**, showing limited performance gain through multi-threading.

Table3. SpeedUp vs. Number of Threads

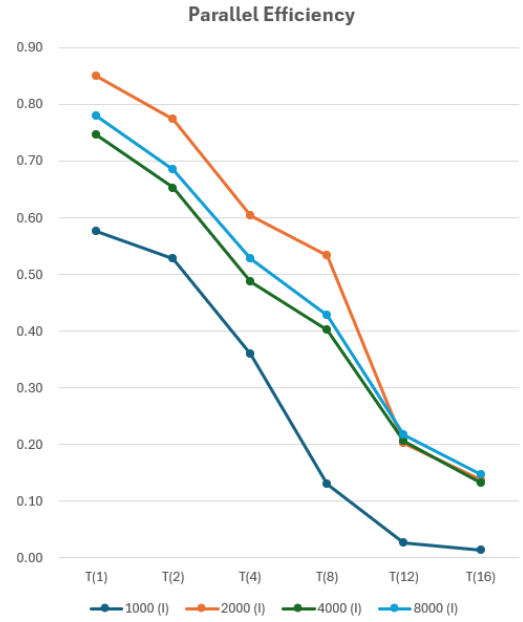


	T(1)	T(2)	T(4)	T(8)	T(12)	T(16)
1000 (R)	0.8	1.5	1.9	2.6	0.9	
2000 (R)	0.9	1.3	1.5	1.6	1.4	
4000 (R)	1.0	1.2	1.2	1.1	1.1	
8000 (R)	0.9	1.2	1.1	1.1	1.0	
1000 (I)	0.6	1.1	1.4	1.0	0.3	0.2
2000 (I)	0.8	1.5	2.4	4.3	2.4	2.2
4000 (I)	0.7	1.3	1.9	3.2	2.5	2.1
8000 (I)	0.8	1.4	2.1	3.4	2.6	2.4

From this point on, I analyze the results based on the Intel CPU.
The data shows that for matrix sizes up to **N = 1000**, the best speedup is achieved when the number of threads is **p = 2**.
When the matrix size exceeds **N = 2000**, the maximum speedup occurs at **p = 8**.
However, increasing the thread count beyond this point actually **degrades performance**, likely due to synchronization overhead and limited parallelism benefits.

Table4. Parallel Efficiency vs. Number of Threads

	T(1)	T(2)	T(4)	T(8)	T(12)	T(16)
1000 (R)	0.79	0.76	0.48	0.32	0.08	
2000 (R)	0.91	0.63	0.38	0.20	0.12	
4000 (R)	0.96	0.60	0.30	0.13	0.09	
8000 (R)	0.94	0.58	0.28	0.13	0.08	
1000 (I)	0.58	0.53	0.36	0.13	0.03	0.01
2000 (I)	0.85	0.77	0.60	0.53	0.20	0.14
4000 (I)	0.75	0.65	0.49	0.40	0.21	0.13
8000 (I)	0.78	0.68	0.53	0.43	0.22	0.15



Parallel efficiency indicates how effectively each thread contributes to performance.
As the number of threads **p increases**, **E(p)** drops sharply, indicating diminishing returns due to overhead.
In particular, when **p exceeds 8**, the efficiency falls below **0.3**, suggesting that the overhead of parallelization becomes significant and outweighs the performance gains.

5. Analysis & Discussion

- The results show that larger matrix sizes generally lead to better parallel performance.
  In my environment, significant speedup was observed when **N > 2000**.
- **Parallel efficiency decreases** as the number of threads **p** increases.
  This is likely due to increased communication overhead and cache contention.
- The hardware architecture has a substantial impact on parallel performance:
  - **AMD Ryzen** uses regular cores, while **Intel i7-1360P** employs a hybrid architecture (Performance + Efficient cores), which seems to provide better scalability in parallel workloads.
- The experiments confirmed a typical parallel trade-off:
  - **Speedup increases** with more threads, but **efficiency tends to decrease** due to overhead, especially beyond a certain thread count.

6. Conclusion

- LU decomposition with OpenMP shows **limited efficiency for small matrix sizes**.
- The parallel implementation is **effective up to 8 threads**.
- **Future optimization directions** include:
  - Block-based decomposition to improve data locality.
  - Cache-aware memory access strategies.
  - **Exploiting NUMA-aware memory allocation** (e.g., using libnuma or numactl) to reduce memory latency and improve performance in multi-socket environments.