Mini Project 2 Report

OpenMP vs Pthreads on Parallel LU Decomposition

Name: SoonHo KIM
Student ID: 20200703
Due Date: May 4, 2025

1. Implementation Details

1.1 OpenMP Implementation

The OpenMP-based implementation is encapsulated in the **lu_decomposition_omp()** function. One of the main advantages of OpenMP is that it enables parallelization with minimal changes to the original sequential code structure. The following steps describe how parallelism is introduced in the implementation:

1. For each step k, the pivot row is selected and rows are swapped if necessary.

   (This step is executed sequentially due to its inherent data dependency.)

2. The computation of $L[i][k]$ is parallelized using **#pragma omp parallel for**, distributing the row-wise computation across threads.

3. The update of $A[i][j]$ is parallelized using **#pragma omp parallel for collapse(2)** to flatten the nested loop structure and improve parallel utilization.

4. Lastly, the diagonal initialization $L[i][i] = 1$ is performed using a simple **#pragma omp parallel for**.

OpenMP simplifies parallel programming by handling thread creation, synchronization, and data sharing through compiler directives. This abstraction makes the code more concise and easier to maintain. However, it may limit fine-grained control and hardware-aware optimizations in specific environments.

1.2 Pthread Implementation

The Pthread-based implementation follows a lower-level approach where threads are explicitly created using **pthread_create()** and synchronized using **pthread_barrier_t**. This gives the programmer full control over thread behavior and synchronization points. The key features of this implementation are:

1. **Thread Worker Dispatching**: Each thread is assigned a unique ID and is responsible for processing a specific chunk of rows, determined by dividing the range $[k+1, n)$ evenly across threads.

2. **Synchronization**: At each iteration k, three barriers (**pthread_barrier_wait()**) are used to enforce strict execution order between the main thread and the worker threads:

   - After the main thread finishes pivoting and computing $U[k][j]$
   - After worker threads compute $L[i][k]$
   - After all threads complete the update of $A[i][j]$

3. **Blocking Optimization**: The function **get_cache_line_size()** is used to determine the cache line size at runtime. The j-loop is then block-partitioned accordingly, optimizing memory access patterns and potentially reducing cache misses.

While this approach introduces complexity in both thread management and synchronization, it allows for fine-tuned control that aligns more closely with underlying hardware architecture. In some test environments, this led to a slightly better speedup than the OpenMP implementation.

1.3 Performance and Scalability Comparison

OpenMP offers a straightforward and intuitive way to implement parallelism, often yielding satisfactory performance improvements with minimal development effort. However, Pthread provides explicit control over workload distribution and memory access, which can result in superior performance if tuned appropriately.

From a scalability perspective, OpenMP automatically handles loop distribution as the number of threads increases, but it is susceptible to parallel overhead and performance degradation due to issues like false sharing. In contrast, the Pthread implementation allows explicit control of chunk and block sizes, enabling more balanced workload distribution among threads. This distinction becomes critical when optimizing LU decomposition for large-scale matrices.

2. Experimental Setup

All experiments were conducted on **Ubuntu 22.04 LTS** using the **g++ 11.4.0** compiler to ensure compatibility with the grading environment.

Test Platforms:

Two hardware configurations were used to evaluate performance under different architectural conditions:

- **AMD Ryzen 5 5600G**: 6 cores / 12 threads, 16MB L3 cache
- **Intel Core i7-1360P**: 4 performance cores + 8 efficient cores (total 16 threads), 18MB L3 cache

Experiment Settings:

- The matrix size n was chosen to satisfy the condition n > 1000, and the following sizes were tested: **1000, 2000, 4000, and 8000**.
- The **random seed number** was fixed at r = 1 for reproducibility.
- The number of threads t tested includes **1, 2, and 4**, as required, and was extended to **8 and 16** for additional insight into scalability.

To streamline batch testing, a custom **run_all.sh script** was written to execute all combinations and log timing results, rather than manually invoking Makefile targets. This helped ensure consistent and systematic benchmarking across configurations.

3. Performance Comparison

There are several structural differences between the OpenMP-based implementations from Assignment 1 and Assignment 2. The two most notable distinctions are as follows:

- **First, the choice of data structure**.
  Assignment 1 uses C-style 2D arrays (**double****), whereas Assignment 2 utilizes the C++ STL **std::vector<std::vector<double>>**. While STL provides safer and more flexible memory management, it introduces an additional level of abstraction, which can result in slightly less efficient memory access compared to raw pointers.

- **Second, the separation and copying of matrices**.
  In Assignment 2, the input matrix A is copied internally to preserve the original data, and the L and U matrices are managed separately. In contrast, Assignment 1 stores both L and U in the same matrix, which saves memory and can improve cache performance due to reduced memory overhead.

Time Measurement Results

Assn1	S	1	2	4	8
n	AMD Ryzen 5 5600G				
1000	0.15	0.18	0.10	0.08	0.06
2000	2.46	2.69	1.94	1.63	1.54
4000	23.39	24.41	19.61	19.35	22.12
8000	179.06	190.40	155.38	158.08	169.54

Increasing →

Time Measurement Results

Assn2	S	1	2	4	8
n	AMD Ryzen 5 5600G				
1000	0.15	0.21	0.14	0.07	0.07
2000	2.47	2.54	1.75	1.67	1.51
4000	25.38	29.28	20.30	20.32	19.85
8000	192.93	243.00	164.93	160.41	175.41

Time Measurement Results

Assn1	S	1	2	4	8
n	13th Gen Intel(R) Core(TM) i7-1360P				
1000	0.27	0.48	0.26	0.19	0.26
2000	3.29	3.88	2.12	1.36	0.77
4000	21.55	28.88	16.50	11.06	6.70
8000	174.85	224.39	127.64	82.84	51.12

Decreasing →

Time Measurement Results

Assn2	S	1	2	4	8
n	13th Gen Intel(R) Core(TM) i7-1360P				
1000	0.12	0.22	0.13	0.14	0.22
2000	1.52	2.01	1.55	1.39	3.23
4000	14.52	18.19	12.90	8.75	9.09
8000	119.75	152.23	101.28	72.79	57.61

Table 1-4. Assn1 and Assn2 execution time results

These differences led to varying performance outcomes depending on the CPU architecture. Interestingly, the same OpenMP-based LU decomposition code showed different execution times depending on the hardware. This discrepancy can be attributed to a combination of the following factors:

- **Access pattern optimizations between STL vectors and C arrays**:
  Modern Intel CPUs tend to have better branch prediction and memory prefetching capabilities, which may reduce the performance penalty of using STL vectors.

- **Cache hierarchy and size differences**:
  For instance, the Intel Core i7-1360P has a larger and shared L3 cache compared to the AMD Ryzen 5 5600G. This may lead to improved cache hit rates even when using separate L and U matrices.

- **OpenMP runtime scheduling strategies**:
  The OpenMP runtime on Intel CPUs may aggressively distribute workloads across threads, yielding better parallel efficiency.

- **Compiler-level STL optimizations**:
  The compiler targeting Intel hardware may generate more optimized code when working with STL containers.

Through this investigation, we were able to reaffirm an important lesson:

"Code performance is not solely determined by the algorithm, but is also deeply influenced by hardware architecture and system-level interactions."

3.1 Timing Table (LU Decomposition Time)

We now focus on comparing the performance of the Pthread and OpenMP implementations based on experimental results obtained from an Intel-based CPU. Although both implementations are based on the same LU decomposition algorithm, their performance characteristics differ due to the parallelization strategy employed.

	p		1			2			4			8			16		
	N	S	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf	T(p)	spdUp	pEf
13th Gen Intel(R) Core(TM) i7-1360P (specifications: 4P+8E/16Threads, L3 cache 18MB)																	
OpenMP	1000	0.1219	0.2230	0.5	0.55	0.1311	0.9	0.46	0.1362	0.9	0.22	0.2204	0.6	0.07	1.0295	0.1	0.01
	2000	1.5172	2.0065	0.8	0.76	1.5474	1.0	0.49	1.3923	1.1	0.27	3.2311	0.5	0.06	7.3431	0.2	0.01
	4000	14.5238	18.1946	0.8	0.80	12.8957	1.1	0.56	8.7481	1.7	0.42	9.0888	1.6	0.20	12.6877	1.1	0.07
	8000	119.7500	152.2340	0.8	0.79	101.2770	1.2	0.59	72.7943	1.6	0.41	57.6089	2.1	0.26	63.1303	1.9	0.12
Pthread	1000	0.1354	0.3305	0.4	0.41	0.1821	0.7	0.37	0.2039	0.7	0.17	0.2465	0.5	0.07	0.4137	0.3	0.02
	2000	1.4985	1.8994	0.8	0.79	1.5451	1.0	0.48	1.3692	1.1	0.27	1.3515	1.1	0.14	2.0908	0.7	0.04
	4000	14.9057	16.4679	0.9	0.91	11.9760	1.2	0.62	8.1727	1.8	0.46	7.3782	2.0	0.25	8.3439	1.8	0.11
	8000	118.0570	130.1580	0.9	0.91	93.7224	1.3	0.63	61.5142	1.9	0.48	50.5047	2.3	0.29	57.4054	2.1	0.13

Table 5. LU decomposition time and analysis

The measured results are summarized in the following timing table, and the subsequent plots are constructed based on these measurements.

All plots are designed in accordance with the guidelines specified in the handout. The x-axis represents the number of threads, while the y-axis displays the performance metric — either speedup or parallel efficiency.

The plots are organized into three categories:

1. A standalone graph showing only the OpenMP results for a fixed matrix size n.
2. A corresponding graph showing only the Pthread results under the same conditions.
3. A combined comparison graph that overlays both OpenMP and Pthread results.

Given that the result values are distributed within a similar range, yet subtle performance differences are critical for analysis, the combined plot (3) is useful for observing overall trends at a glance. However, the individual plots (1) and (2) are also included to enable a more detailed examination of performance differences. These separate graphs complement the combined view and help clarify the distinctions between the two implementations.

3.2 Speedup Plot

Speedup is calculated as T_1/T_n, where T_1 is the execution time using a single thread and T_n is the time using n threads. Ideally, speedup should increase linearly with the number of threads. However, in practice, it tends to level off due to the overhead introduced by parallelization.

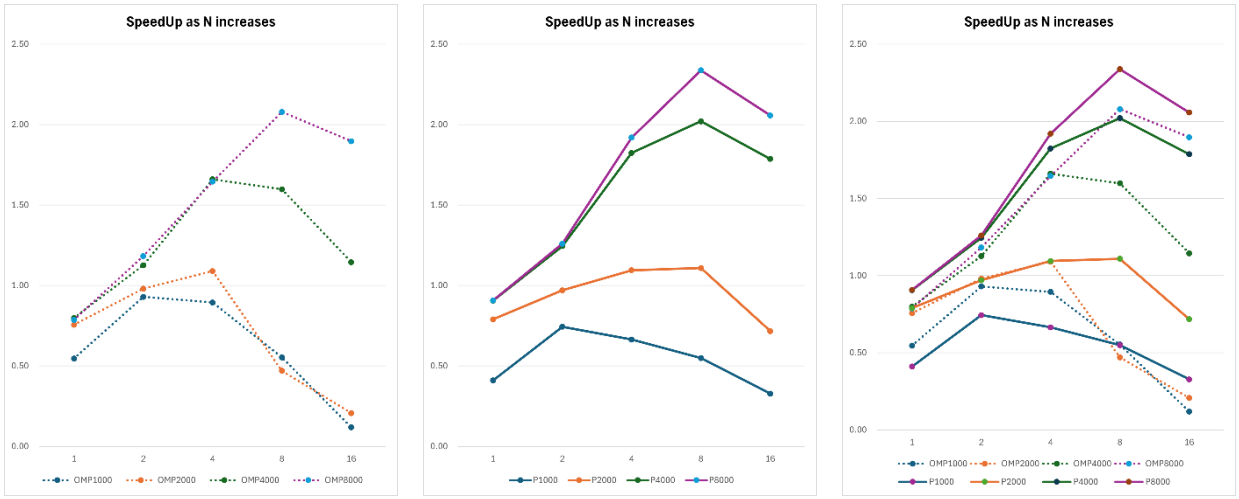

Figure 1-3. speedup plots by three categories

According to the results, both OpenMP and Pthread implementations exhibited similar speedup curves. Nevertheless, in most cases, Pthread achieved slightly higher speedup under the same conditions. This can be attributed to the lower-level control provided by Pthreads, which enables more fine-tuned parallel execution and potentially reduces overhead, resulting in better performance.

3.3 Parallel Efficiency Plot

Parallel efficiency is computed as $T_1 / (T_n \times n)$, reflecting how effectively each thread contributes to the overall speedup. The overall trend is similar to the one observed in the Assn1 experiment, but with some noticeable differences.

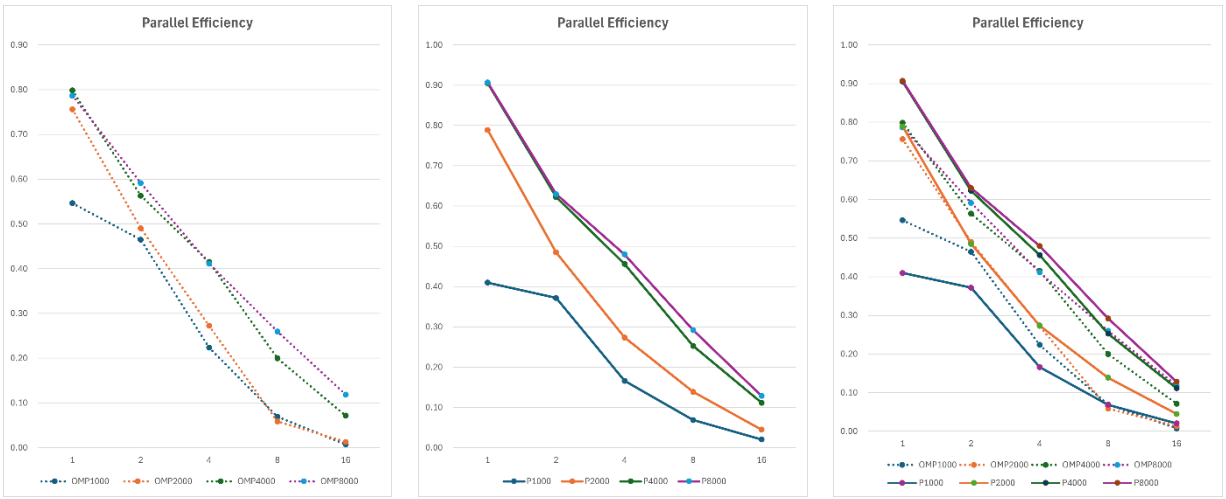

Figure 4-6. Parallel efficiency plots for three categories

In particular, when the number of threads exceeds 4, efficiency drops sharply below 0.3. This suggests that the overhead associated with parallel execution begins to outweigh the performance gains. The inefficiencies can be attributed to various system-level factors, including synchronization overhead, increased cache misses, and memory access bottlenecks.

4. Analysis & Discussion

Experimental results showed that the Pthread-based implementation slightly outperformed the OpenMP version. This performance gain is likely due to the Pthread implementation being tailored specifically for the LU decomposition algorithm. In particular, explicitly assigning row ranges to each thread and employing a blocking strategy for the inner j-loop based on cache line size contributed to improved memory access efficiency.

In contrast, OpenMP offers simplicity and automated loop scheduling through compiler directives. However, it is not free from issues such as false sharing, load imbalance, and cache misses. Moreover, the presence of inherently sequential operations like pivoting significantly reduces parallel efficiency, especially as the number of threads increases, becoming a bottleneck in scalability.

The use of **pthread_barrier_t** for synchronization allowed for clear and structured coordination between computation phases. However, excessive synchronization points also limited the degree of achievable parallelism. While the blocking technique helped optimize cache utilization to some extent, its effectiveness varied depending on the matrix size and the underlying hardware architecture.

5. Summary

In this project, we compared two parallelization approaches for LU decomposition: OpenMP and Pthreads. While OpenMP offered ease of implementation and reasonable scalability, it had limitations in terms of fine-grained optimization for specific computational patterns. On the other hand, Pthreads enabled more precise control over cache behavior and data access, resulting in slightly better performance under certain conditions.

Both implementations achieved meaningful speedup, but parallel efficiency decreased as the number of threads increased. This was mainly due to inherent challenges in parallelizing LU decomposition, such as pivoting overhead and memory locality issues. The results highlight the importance of choosing appropriate parallelization strategies depending on both the algorithmic structure and the target hardware environment.