COL380 Assignment 1 report
LU Decomposition using Pthreads and OpenMP

1 Code overview

Main Function:

1. **Memory Allocation and Initialization**:
   - Memory is dynamically allocated for the matrix `A`, matrices `L` and `U` (for LU decomposition), and an array `A_old` to store a copy of the original matrix `A` for the purpose of calculating residual norm.
   - A permutation vector `pi` is also allocated to keep track of row permutations during LU decomposition.
   - **Pthreads**
     - Here, each thread stores the required parameters, namely the pointers to the Matrices A, L, U and other paramters like starting row, ending row etc in a data structure called ThreadData. This is used for each thread separately.
   - **OpenMP**
     - Here, we do not require the use of the thread data structure, it's implicitly handled by OpenMP.
2. **Matrix Initialization for LU Decomposition**:
   - Matrices `L` and `U` are initialized based on the requirements of LU decomposition.
   - `L` is set to have 1s on the diagonal and zeros elsewhere below the diagonal.
   - `U` is set to have elements of `A` on the diagonal and zeros elsewhere above the diagonal.
3. **LU Decomposition with Parallelization**:
   - The LU decomposition algorithm is executed within a loop iterating over the columns of the matrix.
   - Partial pivoting is performed within this loop, which involves finding the maximum element in the current column and swapping rows accordingly.
   - **Pthreads**
     - The work is divided into chunks to distribute evenly among threads. The number of rows each thread handles depends on the total number of threads and the current iteration of the outer loop.
     - Pthreads are created and joined within the same loop to utilize parallelism effectively.
   - **OpenMP**
     - For OpenMP, instead of explicitly dividing the work in chucks and distributing it evenly among threads, the OpenMP directive directly does the parallelization.
4. **Memory Deallocation**:
   - After the LU decomposition process is completed, memory allocated for matrices `A`, `L`, `U`, `A_old`, and the permutation vector `pi` is deallocated using the `freeMemory` function.

Overall, while both versions achieve parallelization, the OpenMP version simplifies the process by providing a high-level interface for parallelism, whereas the Pthreads version requires more manual management of threads and data structures.

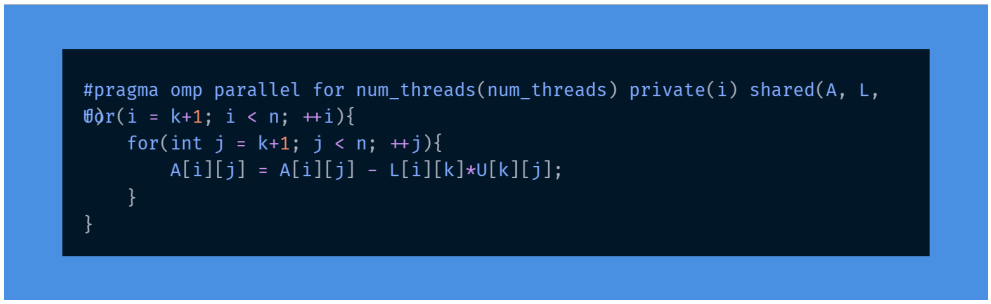
2 Design decisions

- The specific part of the algorithm that is parallelized in this code is the update of the matrix A in the `lu_decomposition` function. This operation involves subtracting the product of elements in the L and U matrices from the corresponding element in the A matrix.
- This operation is independent for each element of the matrix, meaning that the operation for one element does not depend on the result of the operation for any other element. This makes it a perfect candidate for parallelization, as it can be split up into smaller tasks that can be performed simultaneously by different threads.
- Hence we decided to parallelize that section, also considering the fact that it is the main bottleneck during processing.

```
// Divide the work among threads and create them
int chunk_size = (n - k) / num_threads;
for (int i = 0; i < num_threads; i++) {
    thread_data[i].A = A;
    thread_data[i].L = L;
    thread_data[i].U = U;
    thread_data[i].n = n;
    thread_data[i].k = k;
    thread_data[i].start_row = k + i * chunk_size;
    thread_data[i].end_row = k + (i + 1) * chunk_size;
    if (i == num_threads - 1) {
        // Last thread may handle extra rows if n is not divisible by num_threads
        thread_data[i].end_row = n;
    }
    pthread_create(&threads[i], NULL, lu_decomposition, (void
*)&thread_data[i]);

// Join threads
for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
}
// Remaining code ...
```

Figure 1: Parallelized code section for Pthreads



```
#pragma omp parallel for num_threads(num_threads) private(i) shared(A, L,
for(i = k+1; i < n; ++i){
    for(int j = k+1; j < n; ++j){
        A[i][j] = A[i][j] - L[i][k]*U[k][j];
    }
}
```

Figure 2: Parallelized code section for OpenMP

3 Runtime Analysis

n	1 thread	2 threads	4 threads	8 threads	16 Threads
2000	10.259	5.223	2.800	2.389	2.358
4000	80.641	40.625	23.366	20.432	20.939
8000	695.172	350.163	227.030	181.928	185.707

Table 1: Time taken (in seconds) with Pthreads implementation

n	1 thread	2 threads	4 threads	8 threads	16 Threads
2000	7.533	3.914	2.147	1.932	1.943
4000	60.130	34.073	20.221	17.029	17.650
8000	478.16	284.192	183.303	145.766	153.070

Table 2: Time taken (in seconds) with OpenMP implementation

Following are the obtained plots for Parallel efficiency vs number of threads:

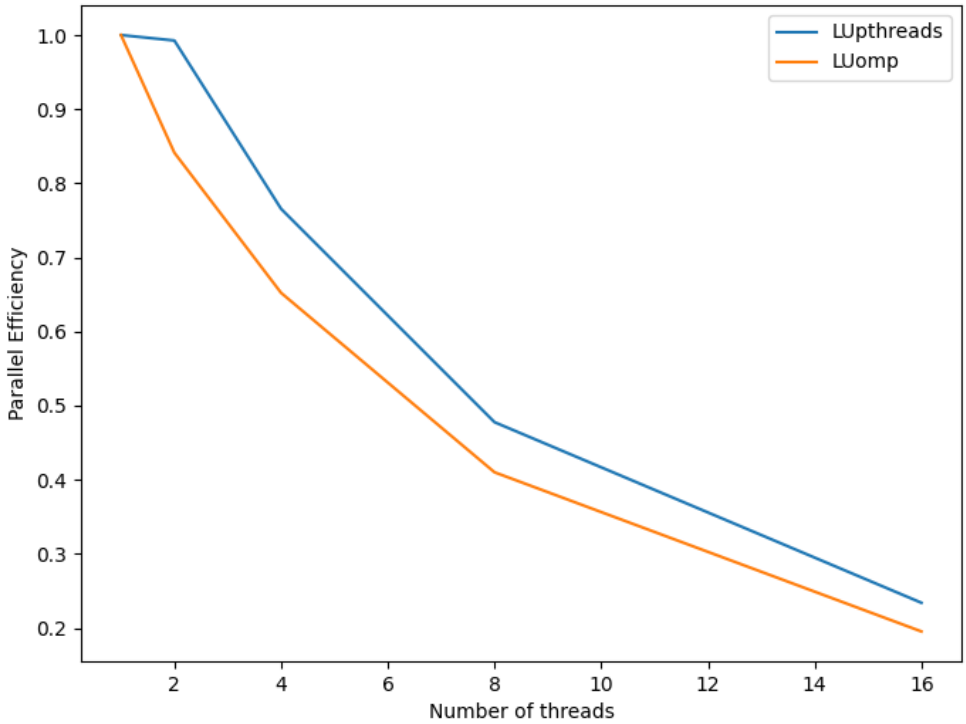


Figure 3: For matrix of size 8000×8000

4 Team info

4.1 Members

	Name	Entry Number
1	Sarthak	2020CS10379
2	Brian Sajeev Katikkat	2021CS50609
3	Aman Hassan	2021CS50607

Table 3

4.2 Device specifications

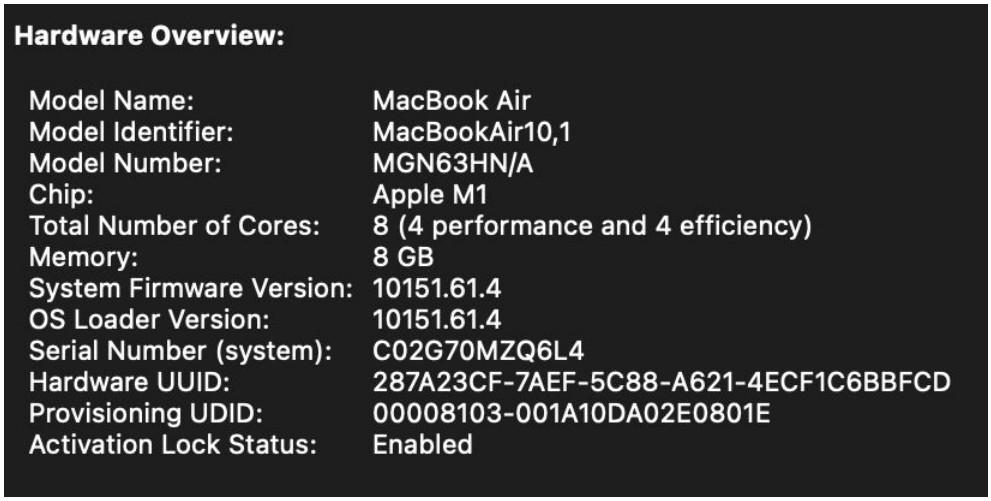


Figure 4: Device specifications