COL380 ASSIGNMENT-1

Parallel and Distributed Programming

Chappidi Venkata Vamsidhar Reddy (2021CS10557)
Harsh Vora (2021CS10548)
Pravar Kataria (2021CS10075)

15/02/24

Contents

1 Introduction

Our objective in this assignment is to develop two parallel implementations of LU decomposition with row pivoting, using OpenMP and pthreads.

2 Sequential Algorithm

The sequential algorithm has 4 main steps performed on every column k = 1 to n.

- Find the maximum absolute element in the column k below the diagonal.
- Swap the row containing maximum absolute element and row k in the matrices A and π. Also swap the first k elements of these two rows in matrix L.
- Construct the column k of L and row k of U.
- Gaussian elimination step: Changing the sub-matrix $A\,[k+1:n]\,[k+1:n]$ according to the pseudo code below:

Algorithm 1 Gaussian elimination step

```
1: for i = k + 1 to n do
2:     for j = k + 1 to n do
3:         A_ij = A_ij − L_ik × U_kj
4:     end for
5: end for
```

3 Parallel Algorithms

- The bottleneck in the sequential algorithm lies in the Gaussian elimination step, which has a time complexity of $O\left(n^3\right)$, while the other steps have time complexities of $O(n^2)$. Therefore, it is sufficient to parallelize this particular step.
- Parallelizing the outer loop of this step speeds up the execution.
- We do not parallellize every loop, such as the other $O(n^2)$ loops, because this slows down the execution due to additional overhead introduced by the creation and subsequent deletion of threads. This dominates over the speedup due to parallellization.

3.1 Pthreads

In the pthreads implementation, the outer loop of the Gaussian elimination step is distributed among the threads. Each thread implements $t = \frac{n-k-1}{num\_threads}$ iterations of the outer loop. Thread p performs the following algorithm concurrently with other threads

Algorithm 2 thread p

```
    for i = (k + 1 + pt) to (k + 1 + (p + 1)t) do
2:      for j = k + 1 to n do
            A_ij = A_ij − L_ik × U_kj
4:      end for
    end for
```

3.2 OpenMP

- Here, we parallelize the outer loop of the gaussian elimination step using pragma, so as to lead to a similar complexity as in the pthreads case.
- Execution times for OpenMP codes here are seen to be faster than the corresponding settings in pthreads. This may be due to further optimisation of the OpenMP code by the c++ compiler.

3.3 Time & Space complexity

- Time complexity: $O\left(\frac{n^3}{t} + n^2\right)$
- Space complexity: $O(n^2)$

where n = size of the matrix, t = min(num_threads, num_cores)

4 Memory Structure

- We experimented with different representations of matrices A, L and U, and observed that a double pointer array representation gives the smallest execution times.
- Time reduction is due to spatial locality principles helpful in accessing adjacent elements of L and U from the cache by the same thread.
- Time reduction is also achieved by prevention of false sharing using a row-major representation of the 'A' matrix. Different rows of 'A' which are accessed by different threads are in different cache blocks when the row-major format is used, hence ensuring no false sharing.

5 Graphs

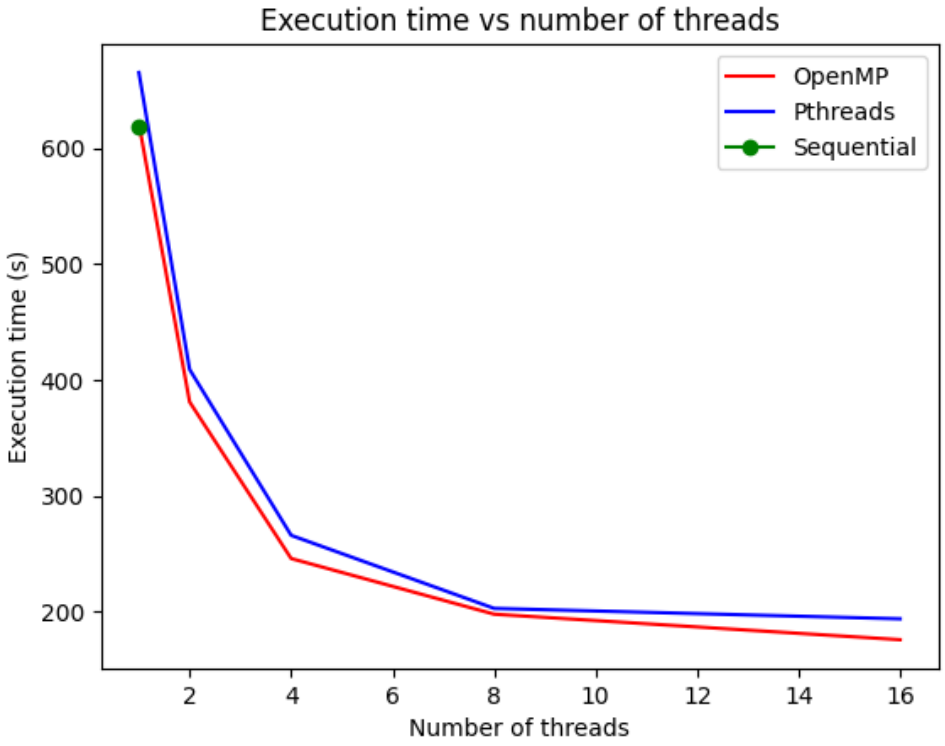

Figure 1: The time taken with OpenMP and pthreads vs number of threads

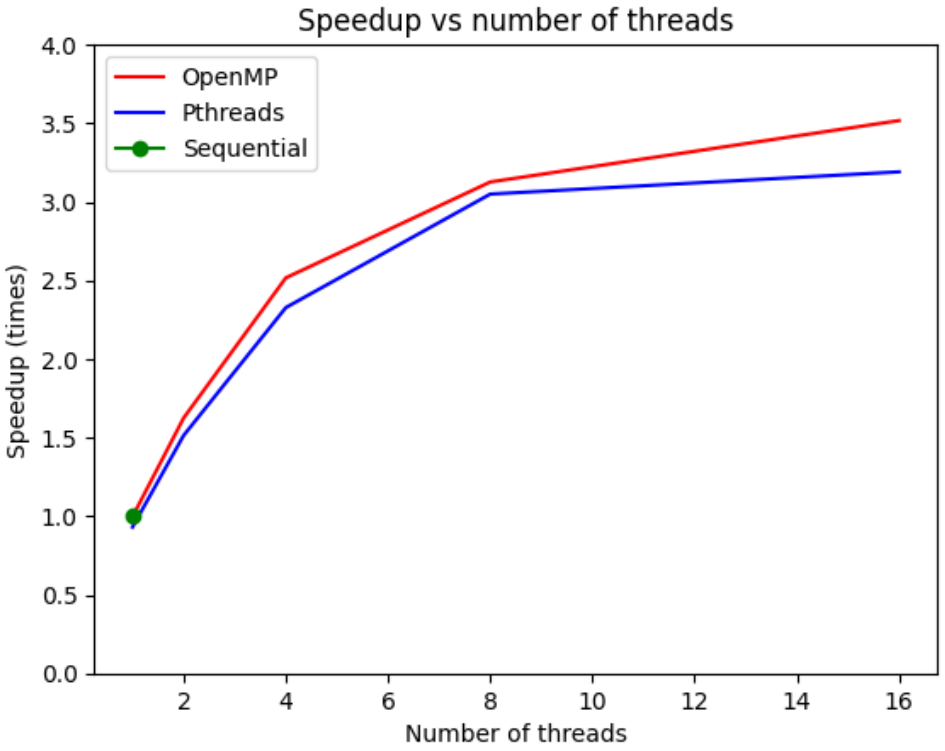

Figure 2: The graph of speedup with OpenMP and pthreads vs number of threads

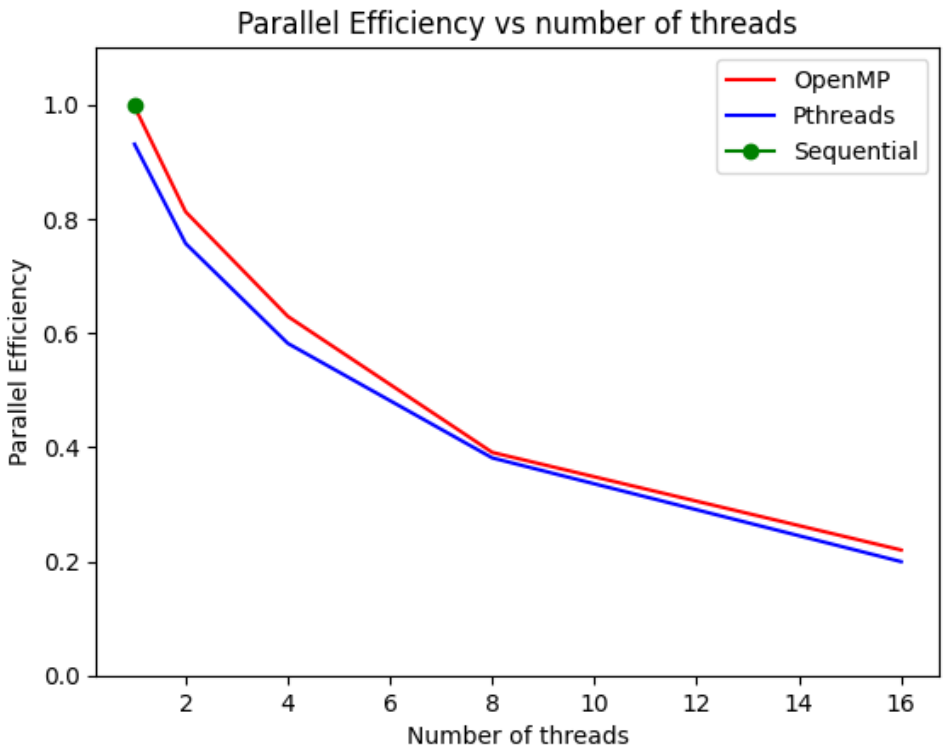

Figure 3: The graph of parallel efficiency using OpenMP and pthreads vs number of threads

6 Data Table

Number of Threads	Time in Parallel (OMP)	Time in Pthreads	Time in Sequential
1	621	665	619
2	381	409	-
4	246	266	-
8	198	203	-
16	176	194	-

Table 1: Execution Times for Different Thread Configurations

- Speedup $= \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$
- Parallel Efficiency $= \frac{\text{Speedup}}{\text{Number of threads (cores)}}$

7 Code structure

7.1 Directory structures

The Zip file contains 3 main files: `omp_array.cpp`, `pth_array.cpp`, and `sequential.cpp`. What is contained in the files is obvious from the names. All three files include `functions.cpp`, which contains all the functions that are used by all three files.

Makefile is made to make the code easier to run. The Makefile structure is provided below:

7.2 Running instructions

We generate random matrix using the `generate.py` file present in the same directory. This takes in the first input argument from the terminal to specify the dimension of the matrix to be generated.

After unzipping the folder, use

```
-- make 'target'
```

to run files. The following targets provide the following functionality:

- `pthreads_array`: This runs the LU decomposition using pthreads. This target needs three inputs `DIM` (dimension of the matrix), `THREADS` (the number of threads), and `CHCK` (if `CHCK = 1` then this also outputs L1,2 norm of $\pi A - LU$, else it just outputs time).
- `omp_array`: This runs the LU decomposition using OPENMP. It requires the arguments similar to `pthreads_array`.
- `sequential_array`: This just runs the LU decomposition on sequential code. This target does not require the `THREADS` argument.
- `all`: This runs all three commands above.

Note that there also exist compiler optimized runs for the same functionality, the respective targets just being suffixed with 'optimized'.
One sample run is as follows:

```
make pthreads_array_optimized DIM=2000 THREADS=4 CHCK=1
```