COL730 Assignment 2 - OpenMP and P-threads implementation of LU Decomposition

Starting with the pseudocode for the sequential LU decomposition algorithm with partial row pivoting,

Initialize the matrices a, l, u and vector pi.

```
inputs: a(n,n)
outputs: π(n), l(n,n), and u(n,n)

initialize π as a vector of length n
initialize u as an n x n matrix with 0s below the diagonal
initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal

for i = 1 to n
    π[i] = i
```

This is done with the help of the random functions provided.

Main Loop over columns

Now we start the algorithm, which involves iterating over all the columns as the first step.

```
for k = 1 to n
```

In each iteration, we do the following 3 steps:

1. Find maximum element of that column (`max`) and where it occurs (`k_prime`)

   ```
   max = 0
   for i = k to n
       if max < |a(i,k)|
       max = |a(i,k)|
       k_prime = i
   ```

2. Swap the rows `k` and `k_prime` in the matrix `a` and in `l` (only upto diagonal as `l` is a lower triangular matrix). Update the perumtation matrix (permutation vector `pi` since we are only going to be swapping rows).

   ```
   swap π[k] and π[k_prime]
   swap a(k, :) and a(k_prime, :)
   swap l(k, 1:k-1) and l(k_prime, 1:k-1)
   ```

3. Calculate `l` and `u` and update the remaining `a`

```
u(k,k) = a(k,k)
for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)

for i = k+1 to n
    for j = k+1 to n
        a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

I have implemented this in `lud.c`. (It is equivalent to the provided `ludptr_openmp.c` and `ludptr_pthreads.c`)

A simple profiling of this code[1] will tell us that the most time is taken in the second loop(`for i; for j; a(i,j) = a(i,j) - l(i,k)*u(k,j)`).
So, we should aim to parallelize that as much as possible. In fact, it takes **~95%** of the execution time of the program (excluding verification). Runtime of main (w/o verification) was **20.09 s**; The nested a-update takes **18.9 s (94.08%)**. [It is $\mathcal{O}(n^3)$]

Optimizations

1. Changing data structure / correct loop order

Using an array of pointers (to columns) is better than using a 2-D array (matrix), as we are iterating on the columns. It depends on what order we want to run our loops in. A column-wise array of pointers would be better than row-wise as both finding max/argmax in a column and L and U updates will benefit from continuous column access, whereas only swapping rows would benefit from continuous row access.
So changing from `double [][]` to `double* []` and making it so there are **less cache misses** because of the **order of the loops** in the `a`-update statement.
Now that we have stored it column-wise, we access the matrices using `x[j][i]` for `ith` row and `jth` column.

```
for i = k+1 to n
    for j = k+1 to n
        a(j,i) = a(j,i) - l(k,i)*u(j,k)
```

This is inducing a lot of **cache misses** at the innermost line as `a[j][i]` keeps changing the column inside the `inner j loop`, and so we are not accessing from that column, but changing columns every iteration (i.e. looking at a different memory location). Exchanging the loops,

```
for j = k+1 to n
    for i = k+1 to n
        a(j,i) = a(j,i) - l(k,i)*u(j,k)
```

This is much faster than the above because of **cache hits**, as can also be seen from the graph (omp, run on dim=2000 with 1 thread):

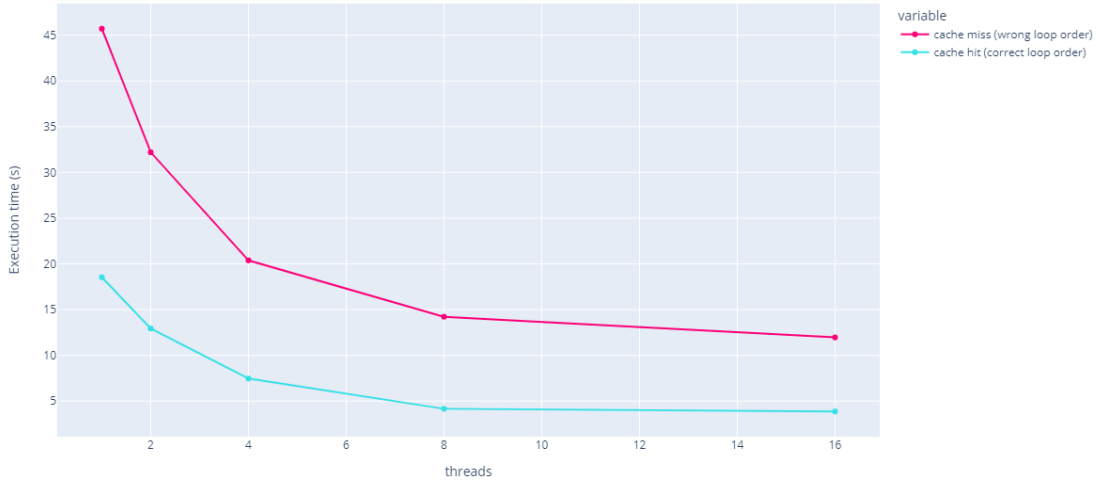

Figure 1: demonstrating the cache miss vs cache hit times

2. Parallelization

1. **Initialization** - The initialization was parallelized using the thread safe/reentrant `drand48_r` function. Since this does not improve performance visibly, we can replace it with single threaded calls to `rand`.
2. **Finding max and argmax** - We can easily find the `max` from multiple threads. However, to find the `k_prime` (argmax), we need to either use an array to store local `max`es and `k_primes`s and then find the corresponding global ones using a loop; or; we can use a critical section at the end of the loops.
3. **Swap rows** - Since we are using a column-wise layout and not row-wise, pointers cannot be directly exchanged.
4. **L & U update** - No loop dependencies.
5. **A update** - The main parallelization, effects runtime the most. Also no loop dependencies.

OpenMP Results

	3	10	500	2k	4k	8k
1 (seq.)	0.0076	0.0067	0.3057	18.5352	143.7408	1133.0058
2	0.0066	0.0068	0.2268	12.9332	99.9348	780.3373
4	-	0.0071	0.1459	7.4575	56.5367	435.3438
8	-	0.0066	0.1027	4.1581	29.6738	231.9506
16	-	-	0.1217	3.8611	24.5281	180.1265

Table 1: Comparing runtime for number of threads (rows) with the dimension of matrix (columns) for OpenMP implementation

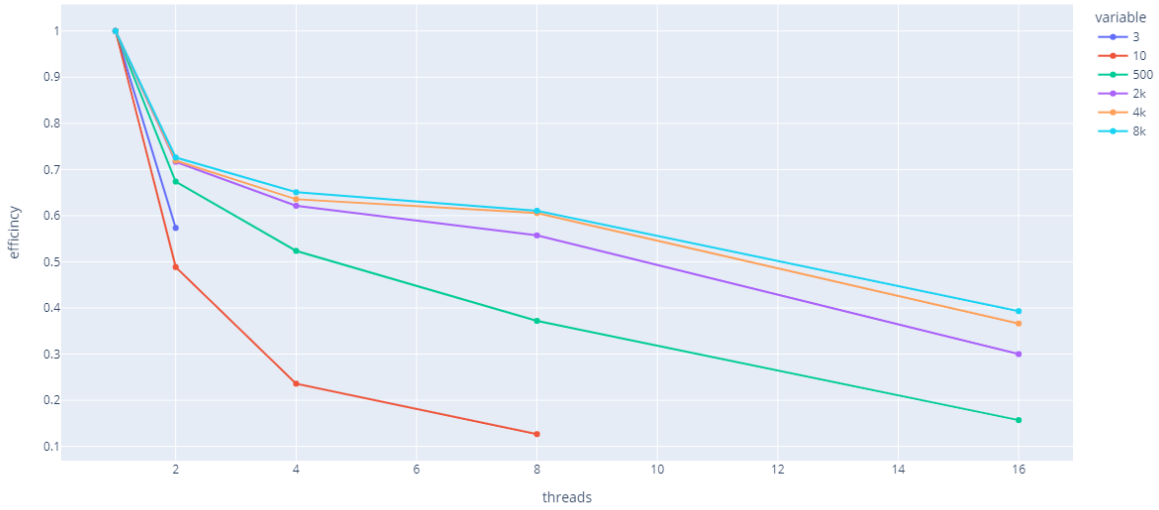

Figure 2: Comparing parallel efficiency with number of threads for various dimensions for OpenMP implementation

pthread Results

	3	10	500	2000	4000	8000
1	0.0115	0.0072	0.3633	19.4330	144.5353	1139.5460
2	0.0072	0.0075	0.2092	9.8756	73.3864	575.1549
4	-	0.0074	0.1561	5.1481	39.2678	297.8582
8	-	0.0075	0.1356	3.3072	21.4759	158.9298
16	-	-	0.1557	3.4417	21.2473	147.3742

Table 2: Comparing runtime for number of threads (rows) with the dimension of matrix (columns) for pthread implementation

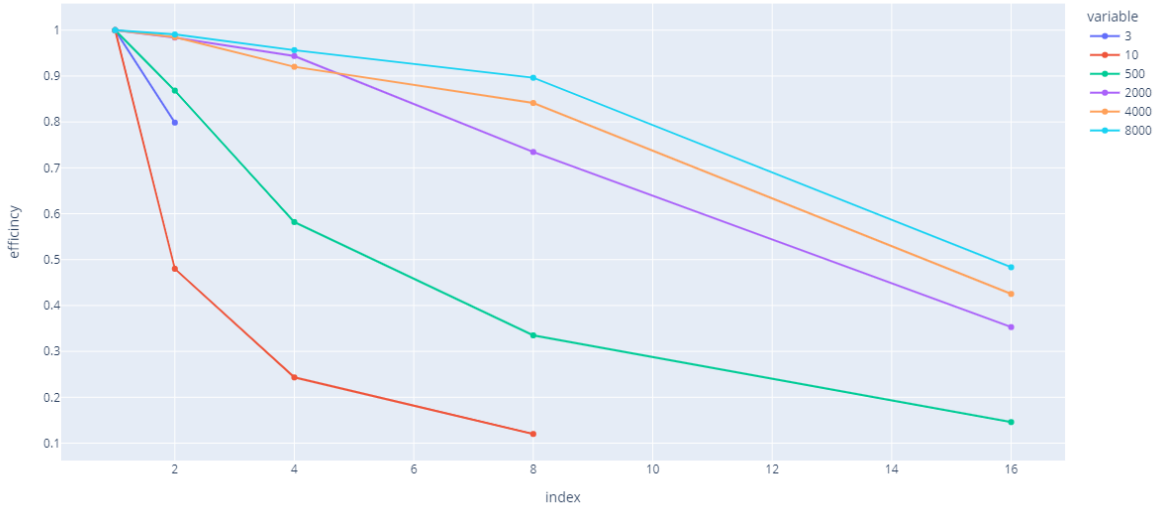

Figure 3: Comparing parallel efficiency with number of threads for various dimensions for pthread implementation

Correctness

The `l` and `u` computed can be checked by calculating the $L_{2,1}$ norm of the matrix $PA - LU$, where P is the permutation matrix should be close to 0 (or exactly 0). This can be verified by uncommenting the sequential verification section at the end of the programs. It uses matrix multiplication so it is slow for high dimensional matrices.

Conclusion

We have parallelized the LU decomposition partial pivoting algorithm, which gives a great speedup and reduction in time for high dimensional matrices.

However, the efficiency goes down, because of the overhead introduced in parallelizing the algorithm.

Speedup and efficiency are defined as:

$$\begin{aligned} S_N &= \frac{\tau_1}{\tau_N} \\ \epsilon_N = \frac{S_N}{N} &= \frac{\tau_1}{N \cdot \tau_N} \end{aligned}$$

All experiments were performed on 16 `Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz` from HPC.

---

1. I have profiled `ludptr.exe` instead of `lud.exe` (for dim=2000) as runtime cannot allocate 2D arrays of that size automatically. `ludptr` uses `malloc` to allocate the required space. ↩