COL380 Assignment 1 — Report

Parallel programming implementation of LU decomposition

The algorithm

First of all, I went with an algorithm—namely the Doolittle algorithm—that differs slightly from the given pseudo-code, although in principle the two are the same. My implementation makes changes in the original matrix that is to be decomposed, and in the end separates out the L and U matrices from it, whereas the original implementation updates L and U at each step of the computation.

Both implementations have the same big-O flops complexity, when it comes to the exact number of flops considering the constant terms in the complexity equation, my implementation turns out to be faster.

The parallel implementations

I have tried out two different memory layouts for the arrays

- A contiguous 1d array of doubles, being used as a 2d one by manually calculating the indices.
- An array of pointers, where each pointer points to an array of doubles, representing a row of the matrix.

I implemented the first one only using Pthreads, whereas I implemented the second one using both Pthreads and OpenMP. So, in total, I tried out three implementation.

Pthreads::

Threadpool paradigm

First of all, the outermost loop of the algorithm can not be parallelised, as at each iteration the matrix gets rearranged and the next iteration depends on the previous iteration. Therefore only the inner loops can be parallelised.

The program uses a threadpool paradigm, creates a pool of threads at the start of the program and uses it throughout for the different chunks of tasks. As I was parallelising only the inner loop, by using a threadpool I could avoid creation and destruction of threads in each iteration of the outermost loop.

Signalling using semaphores

Arguments are passed to the threads using structs, one for each of the threads. The main thread updates the values in the struct, and then signals the threads to start working on those values.

I implemented this signalling using a boolean flag in the argument struct itself—the worker threads would keep checking if the flag was set or not, and when it was set it would start working, at the end of which it would set the flag back to false. But this approach was costly, as the threads were continuously running and checking the flag even when there wasn't any work.

Then I implemented it the proper way, using a semaphore. Each thread has a personal semaphore, and waits on the semaphore until there's work to be done. This way, the threads sleeps when the semaphore is not set.

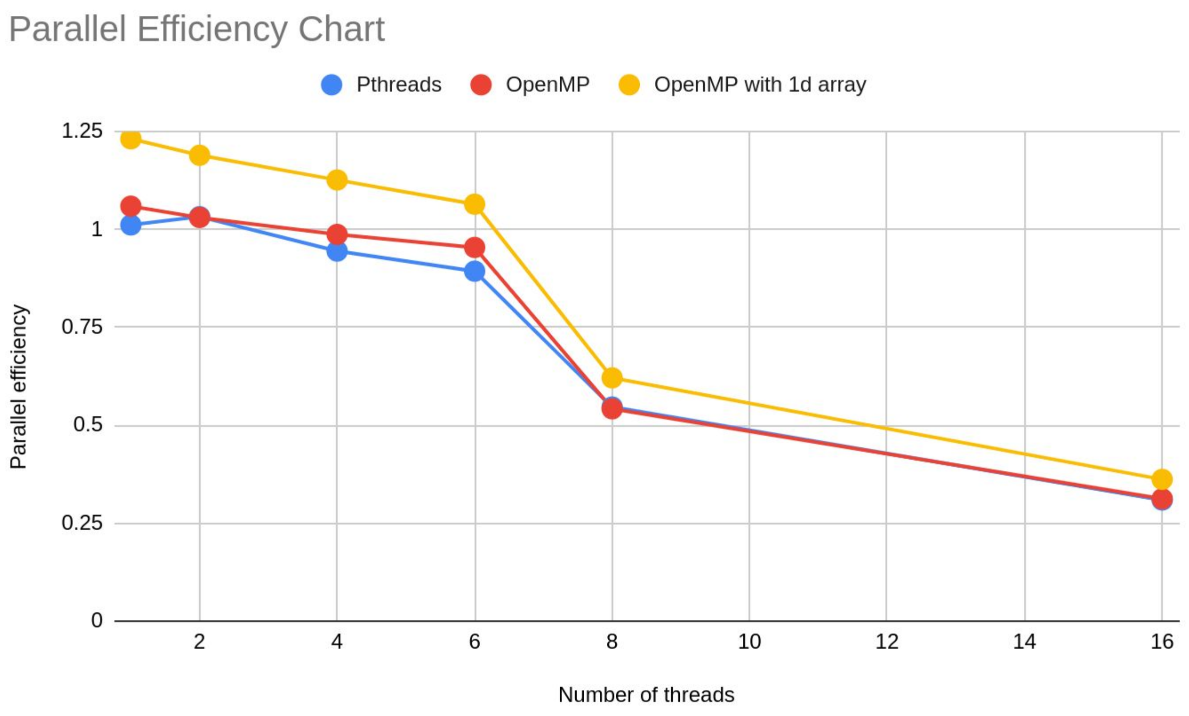
Synchronization using barrier

After each iteration of the outermost loop, before the program can proceed to the next iteration it needs to make sure that each worker thread is finished with the task. To accomplish this synchronization, I used a global barrier, `pthread_barrier_t`. Each thread calls `pthread_barrier_wait()` after it's finished with its task, and the program would only proceed to the next iteration when all the threads have called `pthread_barrier_wait()`.

OpenMP::

For the OpenMP implementations, I simply parallelised the for loops using the `#pragma omp parallel for schedule(static)` directive. The `schedule(static)` argument makes sure that each thread gets equal parts of the loop, because I know that each iteration of the loops take equal time to complete. I also added `collapse(2)` argument to the directive when the loop to be parallelized had an inner loop, that way OpenMP could collapse the two loops and parallelise the whole equally.

Parallel efficiency chart



Observations

- We can see that the parallel efficiency decreases with number of increasing number of threads. This is expected because the overhead associated with creating threads and distributing work among them increases with increasing number of threads.
- There is a sharp drop in the parallel efficiency after 6 threads. This is because the machine I used to run these programs on and make the graph, it has a 6-core, 6-thread CPU, AMD Ryzen 5 3500. As long as the total number of threads is less than the physical threads the CPU supports, the overhead will be much less. That’s why there’s a sharp drop after 6 threads in the graph.