COMP 534: Project 2: Parallel LU Decomposition

Dejan Grubisic

April 2, 2020

Abstract

This project implements the parallel algorithm for finding the lower and the upper triangular matrix whose product is equal to starting matrix - LU decomposition. The experiments are done for 1, 2, 4, 8, 16 and 32 cores for matrix size 4000x4000 and 8000x8000.

1. **The problem statement and a short motivation for why solving this problem is useful.**

   Linear algebra is commonly used in scientific simulations. Data is usually represented by a vector or matrix. Computation on large data-sets could be computationally intensive and it is important to optimize matrix/vector operation. One of the most basic operation is LU decomposition. The naive implementation is based on simple Gauss elimination which complexity is O(N^3), where N is matrix side and for large matrix, this could be unacceptably long, so it need to be parallelized. In order to improve the stability and precision of the result, we use a pivoting technique that searches for the row with the highest first coefficient in each step of Gauss elimination and divides all other rows with that value.

2. **The details of your approach and an explanation of how/why this approach solves your problem.**

   Basically, whole LU decomposition with pivoting could be represented in 3 steps for each element on the main diagonal (Figure 1). The first step is to find maximal elements in the column under the current element on the main diagonal and swap its row with a row of the current element on the main diagonal. To make this step cheap memory is arranged in row-major fashion with the array of pointers to rows, which is particularly beneficial as we don't have to copy whole rows when we do a swap, but rather just to swap the value of pointers to the rows.

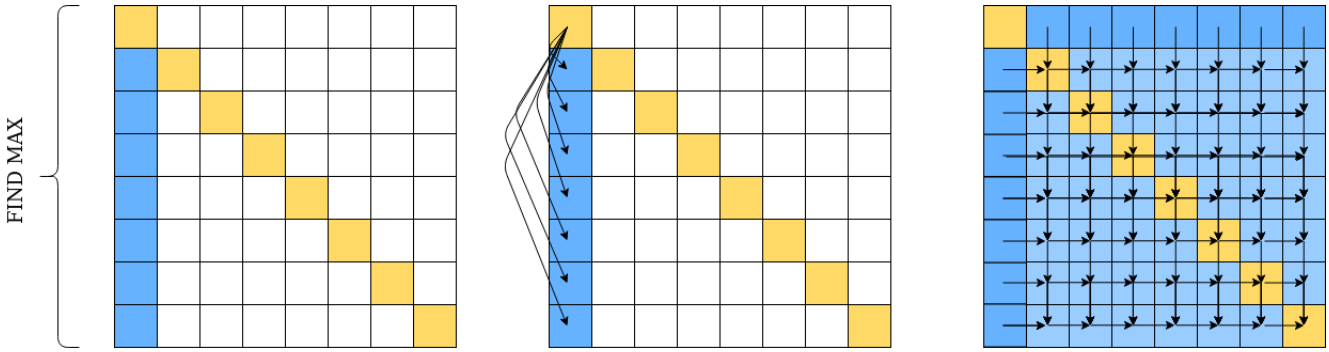

Figure 1: The 3 step algorithm for LU decomposition

The second step is to divide all values under the diagonal with that max element and the third step is to multiply left and upper edged elements colored in darker blue and subtract it from the element in their intersection (Figure 1). This approach implements the LU decomposition in-place and once the

algorithm terminates, the values under the main diagonal plus '1's on the main diagonal represent the lower matrix, while the elements on the main diagonal and upper represents the upper matrix.

In order to make parallel implementation efficient, the key thing was to balance workload and to avoid cache misses as much as possible. For load balancing, it is important that all threads get approximately the same amount of work in every iteration to finish in about the same time before they hit the barrier. On the other hand, to avoid cache misses all threads should read and write from the same memory locations, separate for each thread. In this project, there are considered two memory layouts: interleaved and basic (Figure 2, 3). Both of these layouts take care of two specified conditions under the assumption that there is enough space to store all data.

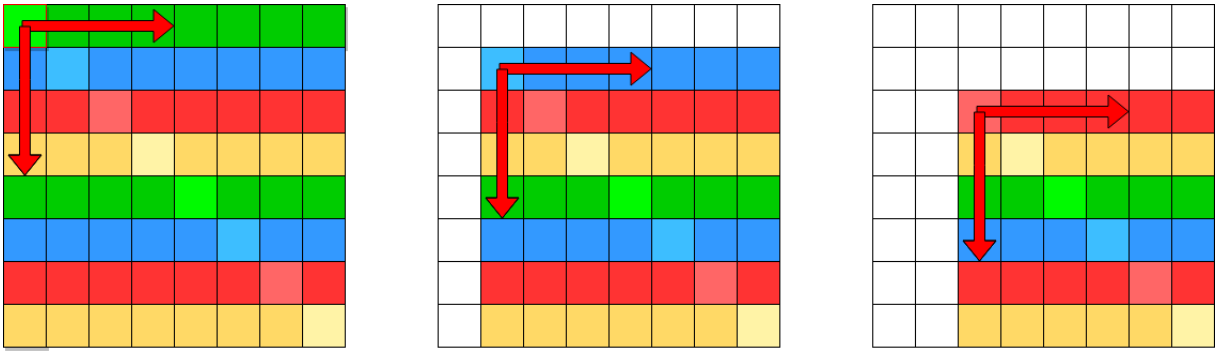

Figure 2: Memory layout Interleaved

By using interleaved layout, in each iteration, one thread gets rid of one row in circular order, which keeps about the same amount of work per thread. Additionally, all threads preserve the computations on rows that they previously have. In the best case, if no swapping is performed, this is the optimal memory placement, as there would be no cache misses (if rows managed by a thread can fit into caches). On the other hand, with every swap, there is the chance of 1 - 1/num_threads that some 2 threads will get a new row to process, while the amount of work remains the same. This is actually not that bad, because once the swap is done threads keep calculation on that rows and there is a maximum 2 cache misses per iteration.

However, the main question here is whether all rows fit inside local threads memory. In Table 1 there are cache sizes together with number of rows that could be stored in them, depending on the row size. From that we can see that only a limited number of rows could be stored. Even worse, these rows are not consecutive in memory and it is not possible to use data locality between them. Another option is to use regular, basic memory placement and to allocate consecutive rows to each thread.

	Cache Sizes(kB)	Rows 8k	Rows 4k	Rows 1k
L1	32	0.5	1	4
L2	256	4	8	32
L3	2048	32	64	256

Table 1: Cache sizes

Unlike interleaved placement, regular placement would group consecutive rows lower than the current element in each iteration and arrange it to threads (Figure 2). This could enable us to collect consecutive rows together, but the problem here is that the majority of the threads will have one

different row in each iteration which could lead to cache misses. Both of these solutions are tested for OpenMP implementation as well as for pthreads, but there was no significant difference in their performance, that could mean that in both cases it is impossible to avoid misses and for better performance, and it is necessary to design more scalable algorithm.

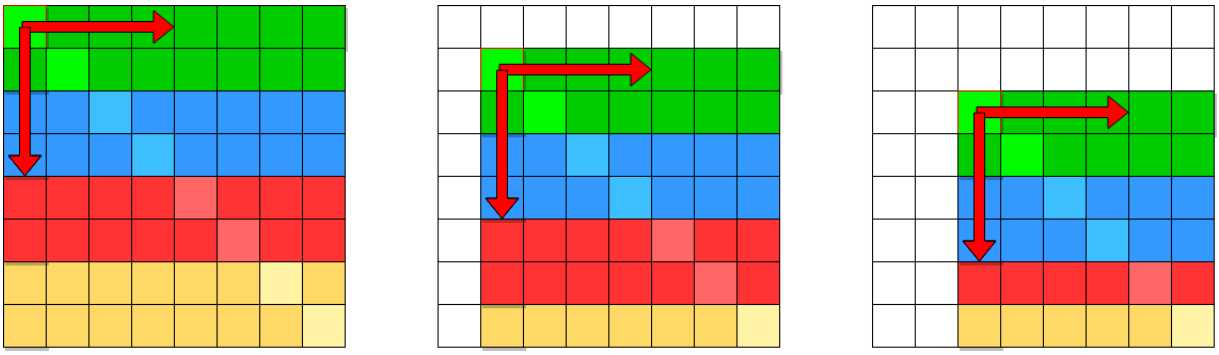

Figure 3: Memory layout Basic

The row-oriented memory ordering also made cheap the step of finding a max element in the current column as all threads could just find the maximum on their rows and let one thread find the global maximum from calculated regional maximums. This step is implemented in both parallel and sequential fashion and no large difference in execution time could be reported. The reason for this is that the largest portion of time is spend in the innermost-loop which hides other parts of the code. Even though, parallel execution could be faster, that step is not particularly interesting for speeding up the application.

The most important part of this application is mapping the third step of the algorithm to particular hardware units that we use and implementing fast synchronizations. For OpenMP implementation, mapping the critical computation to the vector unit could be done by directive *pragma omp simd*. Unfortunately, in this case, this didn't make a change, so we can conclude that there is no simd unit, or the compiler do that by default.

Synchronization for both implementation is implemented as two barriers at the end of each iteration and at the finding maximum in parallel (Figure **??**). For OpenMP that is implemented with directive *#pragma omp barrier*, while finding the global maximum and swapping, after finding regional maximums in parallel are serialized by *#pragma omp single*. Pthreads implementation for the finding regional max in parallel, synchronized with the barrier, after which master thread finds global max sequentially. In meanwhile, if other threads get processor's time they would go to stay in conditional wait until the global max is found, and wake up on signal *max_found_cv*. While loop is enclosing pthread_cond_wait for the case of spurious wakeups.

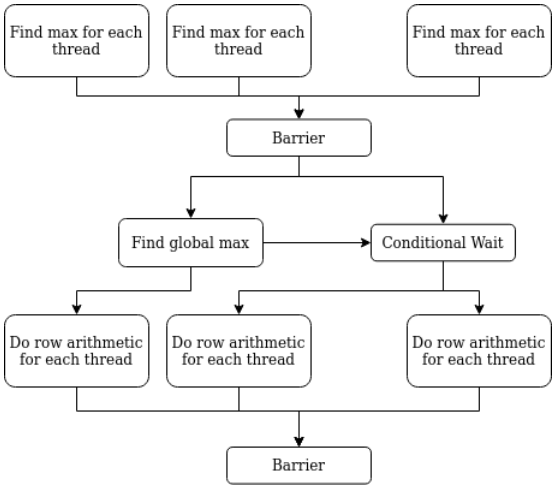

Figure 4: Pthreads iteration control flow

Barriers are implemented with a sense switching approach, combined with conditional wait for threads who are waiting on the barrier. As an alternative to conditional wait, I tried simple spinning in a while loop, but such an approach just spent cycles on waiting threads. For each thread, there is a local_sense flag that alternates on every barrier and once when barrier_counter reaches 0, current thread resets the counter and sends the signal to other threads to continue (Listing 1).

Listing 1: Barrier implementation

```
void barrier(bool &local_sense, int nthreads) {

    local_sense = !local_sense;

    if (std::atomic_fetch_add(&barrier_count, -1) == 1) {

        barrier_count.store(nthreads);
        work_available = false;

        pthread_mutex_lock(&barrier_mutex);
        sense = local_sense;
        pthread_cond_broadcast(&barrier_cv);
        pthread_mutex_unlock(&barrier_mutex);

    } else {
        pthread_mutex_lock(&barrier_mutex);
        while (local_sense != sense) {
            pthread_cond_wait(&barrier_cv, &barrier_mutex);
        }
        pthread_mutex_unlock(&barrier_mutex);
    }
}
```

3. **A description of the experiments you conducted. Give the quantitative and qualitative analysis of your results.**

Implemented parallel algorithm is tested on Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz, with cache sizes L1 - 32kB, L2 - 256kB, L3 - 2048kB. Experiments had been conducted on 1, 2, 4, 8, 16, 32 cores, on the matrix size 4000 and 8000. The results are shown in Figure 5. From these charts, we can conclude that there is no significant difference between omp and pthreads implementation for both matrix size. Similar graphs could be obtained for parallel or serial finding of column maximum or for interleaved or basic memory layout.

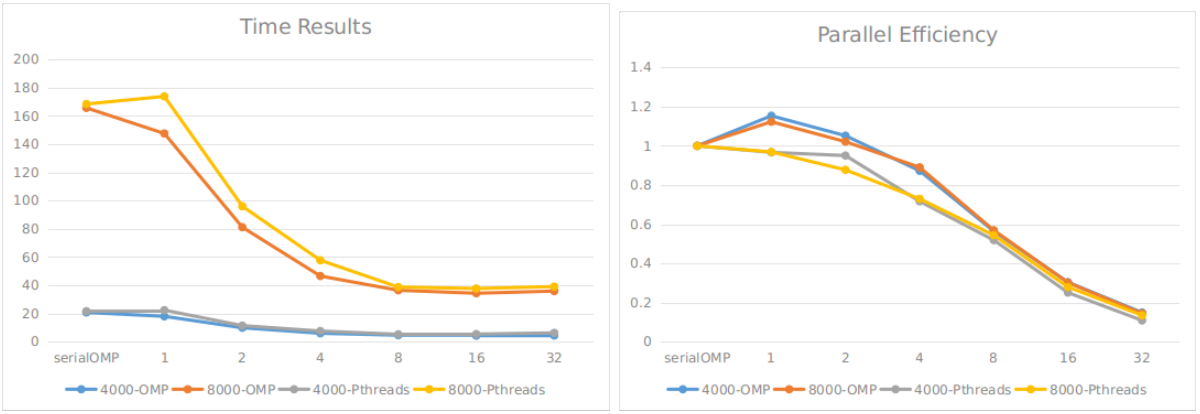

Figure 5: Execution time and parallel Efficiency

Both omp and pthreads implementation results in an error of less than 10^{-8}, with no data races reported by the intel inspector tool.

HPC Viewer results

From the HPC Viewer tool, we can see that the majority of the time is spent in the most inner loop, which was expected. For OpenMP, it is 89.9% while for pthreads is 90.5%. Additionally, omp spends 6.3% on the barrier, while pthreads spends about 1% inside barrier function and conditional wait all-together.

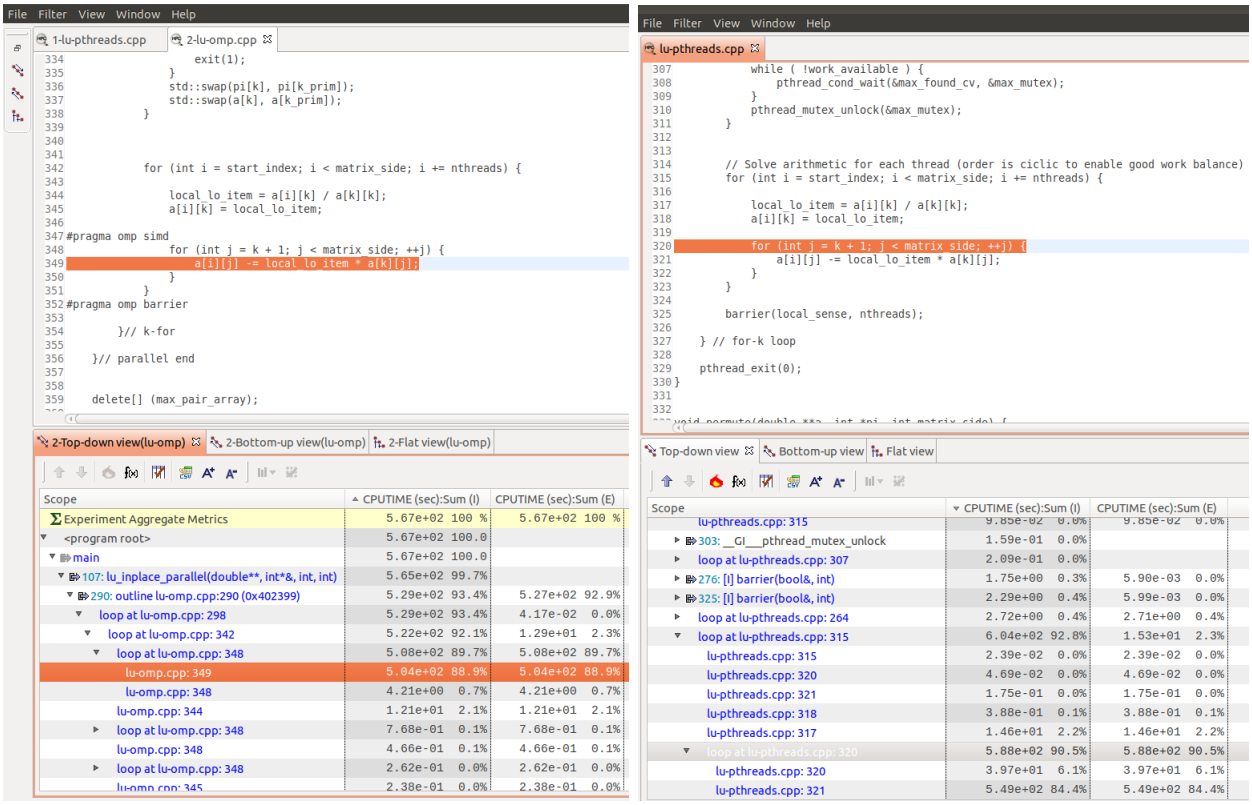

Figure 6: HPCToolkit Viewer results (omp and pthreads)

In Figure 7 we can see that the synchronization pattern matches the result from HPCtoolkit viewer. Based on Statistic from HPCtraceveiwer the omp version spends 89.65% in the parallel region, while for pthreads that is 91.37%.

Nevertheless, strange thing is that we cannot see clear barrier point from where all threads starts together.

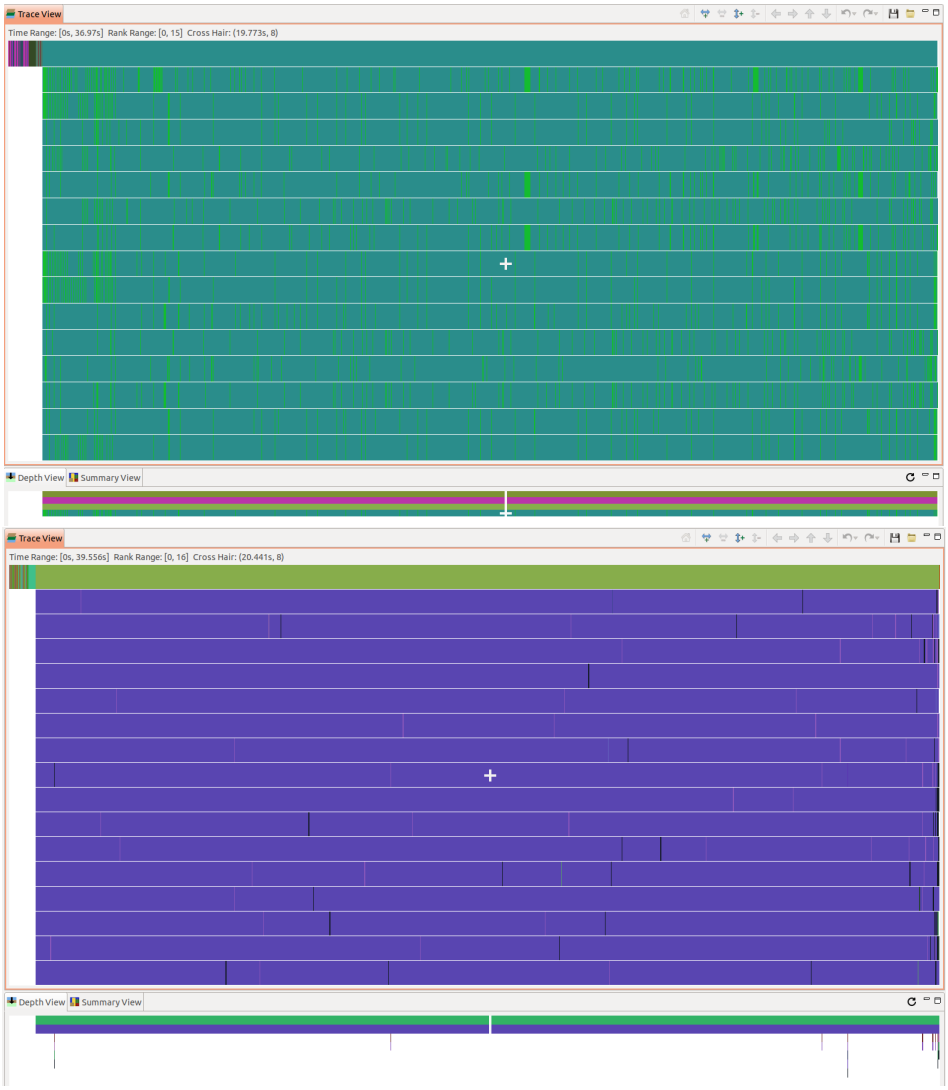

Figure 7: HPCToolkit TraceViewer results (omp and pthreads)

4. **Conclusion** This project implements and analyses the parallel version of LU decomposition. In this report, there are shown the idea and the variants behind the parallel algorithm together with results from HPCtoolkit.

   From shown results, we can conclude that the algorithm chosen for LU decomposition is not particularly scalable and about 8 cores are enough to get a maximal speedup. This report also analyzed different strategies for memory layout and based on different experiments that are done, it is concluded that there is no great difference between interleaved and basic layout, which could mean that there is no enough cache size to take the benefits from first or the second approach. I also tried to use a simd unit for OpenMP, but that didn't result in significant improvement. Finally, we can conclude that a different algorithm should be used to get better scalability, or some special mapping to given architecture should be used to maximize the benefit.