Homework 2 LU Decomposition Writeup

Gyuheon Oh

March 2024

1 Introduction

The traditional approach to LU Decomposition employs Gaussian elimination, a process that methodically reduces a matrix to its upper-triangular form, while tracking the transformation steps in a lower-triangular matrix. This method, though effective, can be computationally demanding for large matrices, posing significant challenges in terms of execution time and resource utilization. In this assignment, we attempt to harness the power of multicore processing to efficiently achieve the goal of matrix decomposition

2 Code Design

At a high level, the decomposition function follows these steps:

1. **Initialization**: The permutation vector P, and matrices L and U are initialized. P keeps track of row swaps due to pivoting, L is initialized to have 1s on its diagonal and 0s elsewhere, and U is initially empty.

2. **Pivoting and Decomposition**: The algorithm enters a parallel region, iterating through columns of the matrix A to perform the decomposition. For each column, it identifies the pivot (the maximum absolute value in

the column) using a parallel reduction to ensure that the pivot selection is efficient and correct in a parallel context. After identifying the pivot, it swaps rows in P, L, and A as needed to perform the row pivoting.

3. **Updating L and U**: Once the pivot is established for the current column, the L and U matrices are updated accordingly in a parallel loop. Each thread works on a portion of the matrix, updating L with the multipliers used to eliminate elements below the diagonal in A and constructing U by copying the modified A.

4. **Barrier Synchronization**: The implementation makes use of OpenMP barriers to synchronize threads at critical points, ensuring that all threads complete their work on the current column before moving to the next column. This synchronization is crucial for maintaining the correctness of the LU decomposition process across multiple threads.

5. **Row Swapping and Copying**: Within the parallel region, row swaps are performed based on the pivot decisions, and necessary adjustments are made to A, L, and U to reflect these swaps, ensuring that the decomposition progresses correctly even as rows are permuted.

2.1 Reduction

```
struct Pivot {
    double val;
    int index;
};
// Comparison function
inline Pivot max_pivot(const Pivot& a, const Pivot& b) {
    // Return the pivot with a the greater value
}
```

```
#pragma omp declare reduction(maximum : struct Pivot : omp_out =
    max_pivot(omp_in, omp_out))
```

In the given code snippet, a custom reduction operation for OpenMP is defined to find the maximum value and its index from a set of `Pivot` structures across parallel threads. The `Pivot` structure contains two members: `val`, a `double` representing the value, and `index`, an `int` representing the index of this value.

We define a `max_pivot` function, which compares two instances of `Pivot`, `a` and `b`, returning the one with the larger value (`val`).

This custom reduction allows us to merge these local maxima from each thread into a global maximum in a thread-safe manner. After the pivot is determined, row exchanges and updates to the L and U matrices proceed, based on the global pivot identified through the reduction operation.

2.2 Pivot Initialization

```
    #pragma omp parallel shared(L, U, A)
{
    #pragma omp for schedule(static) reduction(maximum:max)
    for (int i = 0; i < n; i++) {
        P[i] = i;
        std::fill(U[i].begin(), U[i].end(), 0);
        std::fill(L[i].begin(), L[i].end(), 0);
        L[i][i] = 1;

        if(std::abs(A[i][0]) > max.val) {
            max_pivot.val = std::abs(A[i][0]);
            max_pivot.index = i;
        }
    }
}
```

In this parallel section, each thread initializes the diagonal of L to 1, zeroes out the rest of L and all of U, and fills the permutation vector P with the natural order. It also identifies the pivot (the element with the maximum absolute value) for the first column of A, which is used for partial pivoting.

2.3 LU Decomposition

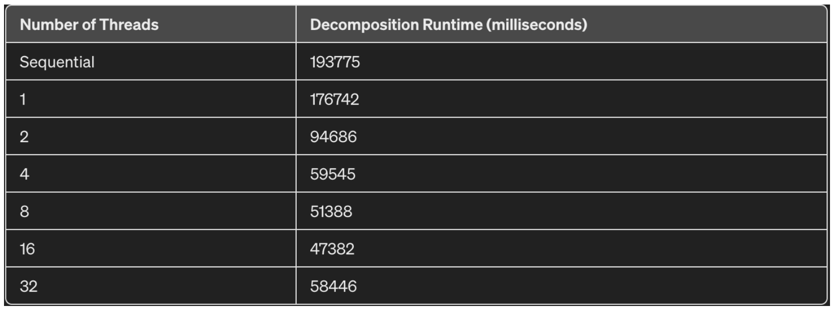
```
    #pragma omp parallel shared(L, U, A)
{
    for (int k = 0; k < n; k++) {
        #pragma omp master
        {
            // Code for swapping rows and preparing the pivot element
        }

        #pragma omp barrier
        #pragma omp for schedule(static) reduction(maximum:max)
        for (int i = k + 1; i < n; i++) {
            // Code for computing the elements of L and U
        }
    }
}
```

This section performs the core of the LU decomposition. It is a loop over columns k of matrix A, where for each column, it:

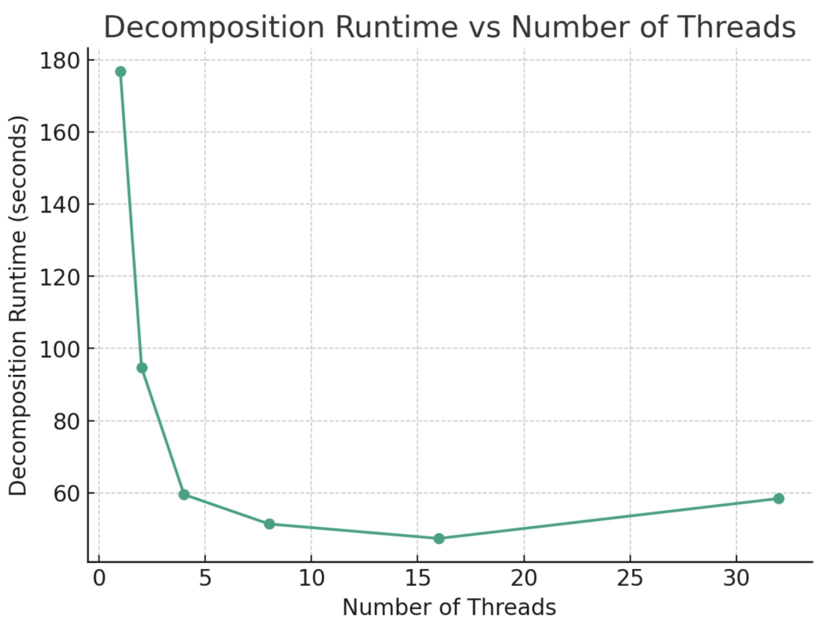
- Uses the master thread to swap the k-th row with the row of the identified pivot to maintain numerical stability.

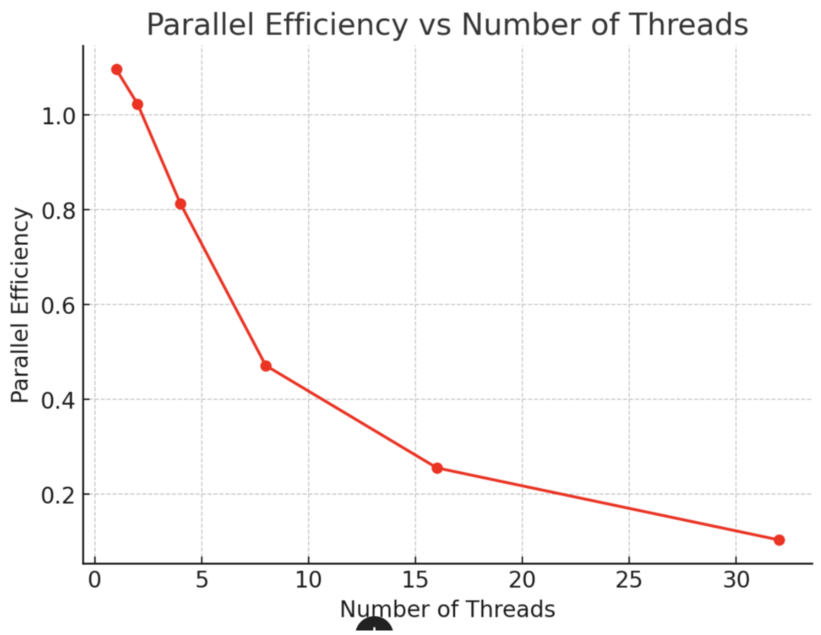
- Performs a barrier synchronization to ensure that row swapping is completed before proceeding.
- In parallel, updates the L and U matrices. For L, it calculates the multipliers that zero out the elements below the diagonal. For U, it updates the upper triangular part including the diagonal elements.
- Identifies the next pivot element for partial pivoting in the next column.

3 Result Analysis



Number of Threads	Decomposition Runtime (milliseconds)
Sequential	193775
1	176742
2	94686
4	59545
8	51388
16	47382
32	58446





This is a table of the runtime for different number of threads for the decomposition of a 8000×8000 randomly generated matrix and also a graph that visualizes how the runtime changes as we increase the number of threads in our parallel program. We can also see how our parallel efficiency changes as we add cores.

We can notice that our program does not scale well horizontally as we add more cores. I remembered from class that placing large matrices all over the place in memory with different processors accessing different parts of the row could lead to longer wait times. I also remembered that when different processors accessing the same cache line could cause performance degradation. I decided to tweak my implementation, allocating each row in the matrix specific to a thread, then processing the matrix by chunks of memory closest to the processor that is processing that memory.

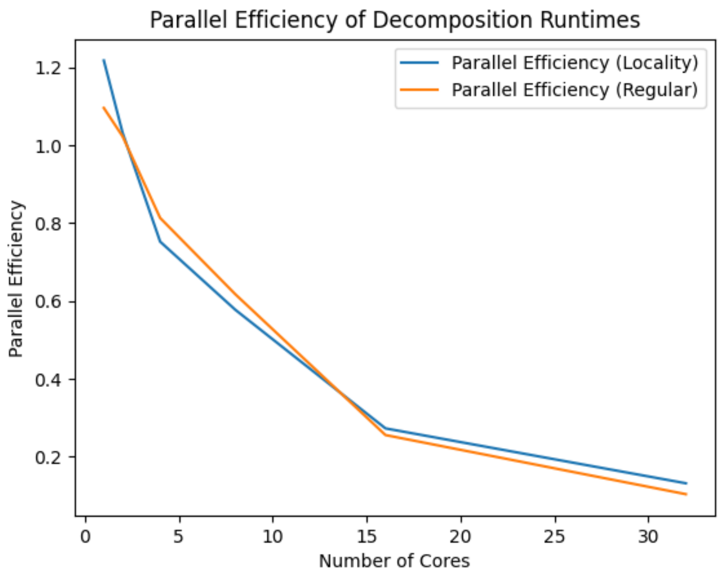
In order to achieve this, I first did this:

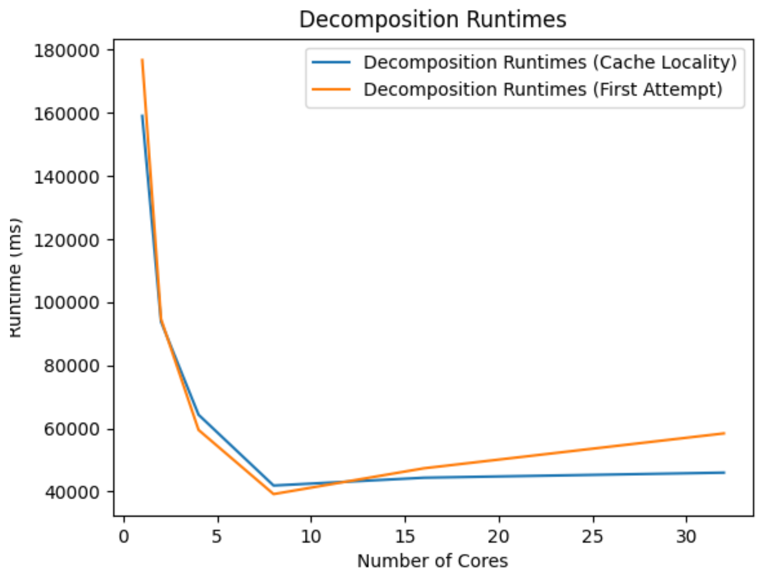
```
void generate_rand_matrix(...) {
    // Allocate memory for the array of pointers using
        numa_alloc_local
    A = reinterpret_cast<double**>(numa_alloc_local(n *
        sizeof(double*)));
    A_copy = reinterpret_cast<double**>(numa_alloc_local(n *
        sizeof(double*)));
    // Allocate and initialize each row of the matrices
    #pragma omp parallel for num_threads(num_threads)
    for (int i = 0; i < n; i++) {
        A[i] = reinterpret_cast<double*>(numa_alloc_local(n *
            sizeof(double)));
        A_copy[i] = reinterpret_cast<double*>(numa_alloc_local(n *
            sizeof(double)));
}
```

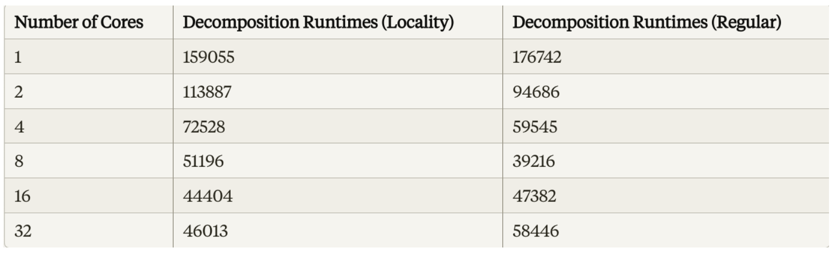
Then, I changed my omp for loop annotation to include a ‘chunkSize‘ parameter as such

```
#pragma omp for schedule(static, chunkSize)
       for (int i = 0; i < n; i++) {
           ...
```

This assigns chunkSize amount of rows to each thread, where the chunk is the same chunk of rows allocated in the memory of the node using numa_alloc_local when we first initialized the matrix. By doing this, we effectively store the parts of the matrix closest to the processor that will process it, improving cache locality, and also making sure that false sharing is minimzied; there is less of a chance of multiple processors querying the same cache line. With these changes we saw these results.







Number of Cores	Decomposition Runtimes (Locality)	Decomposition Runtimes (Regular)
1	159055	176742
2	113887	94686
4	72528	59545
8	51196	39216
16	44404	47382
32	46013	58446

I was happy to see that this change led to improvements in runtime and parallel efficiency at high number of cores (16 and 32). I predict that this is because with more cores, having the matrix stored all over the place leads to more sub-optimal cache queries and memory accesses, along with a higher chance of false sharing of cache lines.

However, we can still see that the parallel efficency of the program drops as we add cores. This means that we are seeing diminishing returns for each processing core that we add. The phenomenon we're experiencing may be a classic illustration of Amdahl's Law, which posits that the overall performance improvement gained by optimizing a particular portion of a task is limited by that portion's contribution to the entire task. If a significant part of our program must execute more must sync and wait for all threads to complete, the benefits of adding more cores will inevitably taper off. This is because the serial portion of the workload does not benefit from additional cores, acting as a constant overhead that caps the overall speedup. In addition to Amdahl's Law, parallel overhead is a critical factor that can degrade the efficiency of ours as the number of cores increases. This overhead includes the time spent on communication between cores, synchronization efforts (e.g., ensuring that all threads reach a certain point before proceeding), and the logistics of distributing and gathering data among processors. Each of these activities consumes resources and time, which do not contribute directly to solving the problem at hand but are necessary for parallel execution. As the number of cores increases, these overheads can escalate, sometimes exponentially, because more coordination and data exchange are required. This increase in overhead can erode the time saved through parallel execution, leading to a situation where adding more cores results in smaller incremental improvements.

Also, in order to support thread id bound chunks, I could not use a singular Pivot struct, and had to create an array of these for each thread processing a chunk of the matrix. This feels like a sub-optimal hack and this likely contributed to some computational overhead.

In conclusion, it was slightly disappointing to see that the implementations accounting for data locality did not significantly outperform the first implementation. There may have been other inefficiencies in these implementations that negated the effects of data locality. It's possible that accessing far away memory was not a bottle neck. In the future, I would like to explore different matrix representation. Indexing arrays is a very fast operation; what if we represented the matrix in one sigular array and divided it up into chunks, indexing into it using i * j + c, instead of i,j. However, would this approach lead to deadlocks and shared access to a singular reference, since all threads are accessing the pointer to the array then + i * j + c? These are all question that I think would be worth asking in future exploration.