Report
Parallel LU Decomposition

Gayathri Shreeya, CO21BTECH11010

1 Introduction

LU decomposition, also known as LU factorization, is a matrix factorization technique used in numerical linear algebra. It decomposes a square matrix A into the product of a lower triangular matrix L, and an upper triangular matrix U. LU decomposition is a powerful numerical technique employed in efficiently solving systems of linear equations. Additionally, LU decomposition simplifies the calculation of determinants and matrix inverse. The technique is extensively used in numerical simulations, such as finite element analysis, for solving linear systems arising from partial differential equations and is pivotal in image processing for tasks such as compression and recognition.

1.1 Existence of LU factorization

If the matrix A is invertible, then it admits LU factorization if and only if all its leading principal minors are non-zero. If A is a singular matrix of rank k, then it admits LU factorization only if the first k leading principal minors are non-zero.

1.2 Gaussian Elimination

Gaussian elimination is an algorithm used to transform a given system of linear equations into an equivalent triangular system, making it easier to solve. In this process, the operations performed implicitly generate the matrices L and U. The lower triangular matrix L contains the multipliers used in the elimination steps, and the upper triangular matrix U is the result of the elimination process. This is also why LU decomposition is known as the matrix form of Gaussian elimination.

1.3 Pivoting

Pivoting is a technique employed during Gaussian elimination, which is also a foundational step in LU decomposition. The diagonal element used in the elimination is called a pivot. The elimination fails if the pivot is zero. Even if the pivot is not identically zero, a small value can result in big round-off errors. For large matrices, the error propagates and one can lose all accuracy in the solution. Pivoting is crucial for numerical stability, especially when dealing with ill-conditioned matrices. Two common pivoting strategies are partial pivoting and complete pivoting.

- **Partial Pivoting:** Partial pivoting involves exchanging only rows. For increased numerical stability, we choose the largest pivot element by searching the partial column below the diagonal entry. By selecting the largest pivot element on the diagonal, we can minimize the effect of round-off.
  If $|a_{kk}| \neq \max_{k \leq i \leq N} |a_{ik}|$, then swap rows k and i
- **Complete Pivoting:** In complete pivoting or full pivoting, we exchange both rows and columns to select the largest possible element from the sub-matrix. However, column exchange requires changing the order of x_i. Full pivoting is less susceptible to round-off, but the increase in

stability comes at the cost of more complex programming and an increase in work associated with searching and data movement.
If $|a_{ij}| \neq \max_{1 \leq i,j \leq N} |a_{ij}|$, then swap rows i and j and columns i and j

2 Sequential LU Decomposition

2.1 Algorithm Overview

A typical sequential algorithm for computing the LU decomposition of a matrix A resembles Gaussian elimination. In this algorithm, we iterate over columns of U, eliminating elements in column k below the diagonal using row transformations. Specifically, we calculate a multiplier $a_{ik} = a_{ik}/a_{kk}$ for each row $i > k$ and update the elements in row i using this multiplier, $a_{ij} = a_{ij} - a_{ik}a_{kj}$. The pseudo-code for sequential LU factorization is shown below. The final LU factorization is obtained by extracting the lower and upper triangular matrices from the resultant matrix. One way of doing this (which is also used in the code) is Doolittle-like decomposition to get a unit lower triangular matrix L and an upper triangular matrix U. One important point to note is that this implementation is *in-place computation*.

Algorithm 1 Sequential LU Factorization

```
1: for k ← 1 to N − 1 do
2:     for i ← k + 1 to N do
3:         a_ik ← a_ik / a_kk
4:     end for
5:     for j ← k + 1 to N do
6:         for i ← k + 1 to N do
7:             a_ij ← a_ij − a_ik · a_kj
8:         end for
9:     end for
10: end for
```

2.2 Complexity Analysis

From the pseudo-code, we can infer that the time complexity of sequential LU decomposition is $O(N^3)$. Pivoting introduces additional operations, and its impact on time complexity depends on the chosen strategy. Since pivoting is done at every step, partial pivoting requires $O(N^2)$ comparisons and complete pivoting may involve $O(N^3)$ operations to search for the maximum element.

2.3 Challenges

- The cubic time complexity $O(N^3)$ becomes more pronounced for large matrices, limiting the scalability of the algorithm.
- Sequential LU decomposition does not exploit the capabilities of modern parallel architectures.

3 Parallel LU Decomposition

3.1 Pivoting again!

The stability of LU decomposition is guaranteed in the presence of pivoting. In parallel algorithms however, pivoting is a challenging task. When multiple threads are concurrently updating different portions of the matrix, it introduces data dependencies. Coordinating these dependencies across threads

to perform pivoting becomes difficult, and one thread's pivot decision will impact the computation of others. Pivoting decisions made by one thread requires robust communication mechanism which introduces both overhead and synchronization challenges. Efficient inter-thread communication mechanisms are essential, but implementing them without compromising parallel performance is a complex task.
For the sake of simplicity, this code does not perform pivoting. This code is only tested on a special class of matrices that are *strictly column diagonally dominant*. These matrices do not require pivoting and can be directly decomposed. These matrices are generated using the function `generateMatrix()`.

3.2 Task Parallelism

In the sequential LU decomposition, the bottom right sub-matrix of size $(N-k)$x$(N-k)$ is updated after column k has been scaled with a_{kk}. The part of the code where the updates are carried out can be executed in parallel. In our parallel LU decomposition, threads are tasked with updating specific rows of the matrix independently, facilitating efficient parallelization.

3.3 Load Balancing

One of the integral components of parallel computing is deciding how the load is distributed evenly among processors or threads. Here are some of the key points regarding load balancing in this code:

- **Cyclic Allocation of Rows:** In this code, each thread is responsible for updating a set of rows at every step of the algorithm. The allocation of rows adopts a cyclic pattern. For instance, if there are five threads, the first thread handles rows 1, 6, 11, and so forth. However, this allocation strategy is static.
- **Proportional Workload Distribution:** Static allocation of work isn't exactly ideal, especially because some threads could be involved in significantly more computation than others. Despite the static allocation, each thread eventually undertakes roughly the same number of operations. So, the static allocation does a good job of ensuring that each thread has a proportional share of computational work.
- **Addressing Idleness Concerns:** One important point to note is that the cyclic allocation of rows is very strategic. Imagine the case where each thread is assigned a continuous block of rows to update. As the program executes, the size of the sub-matrix to be updated becomes smaller and the only threads that have any work left are the ones that have been assigned the last couple of blocks. And as the program progresses, fewer and fewer threads will have any active work left. Cyclic allocation mitigates the risk, sustaining active engagement of all threads for the most extended duration possible. This is critical for optimizing parallel efficiency, especially in the context of large matrices.
- **Possibility of Dynamic Load Balancing:** While the code presently employs a form of static load balancing through cyclic allocation, exploring the benefits of dynamic load balancing could add depth. We naturally expect dynamic strategies to be more efficient for distributing the load among different threads.

3.4 Shared Memory

In this implementation, the matrix `matrix` and the array `numUpdates` are shared among all threads. Each thread modifies the rows of `matrix` allocated to it. `numUpdates` stores the count of the number of times each row has been updated. All threads read the elements of `numUpdates` but only the thread responsible for row i can update the value of `numUpdates[i]`.

3.5 Synchronization

`numUpdates` is an array of `atomic<int>` of size N. `numUpdates[i]` keeps track of the number of times the i^{th} row has been updated. It serves as a synchronization and communication mechanism between threads. At the k^{th} step of execution, the bottom-right sub-matrix of size $(N-k)$x$(N-k)$ needs to be updated. To do that, all threads have to wait until the k^{th} row is updated i.e., until `numUpdates[k] == k`. In the current implementation, these threads are spinning on this atomic element.

4 Program Design

Algorithm 2 Runner Function

```
1: procedure RUNNER(id)
2:     toCheck ← ∅
3:     for i ← 0 to N − 1 do
4:         if i mod p = id then
5:             append i to toCheck
6:         end if
7:     end for
8:     for k ← 0 to N − 2 do
9:         while numUpdates[k].load() ≠ k do          ▷ Check if the k-th row is updated k times
10:        end while
11:        for all i in toCheck do
12:            if i > k then
13:                matrix[i][k] ← matrix[i][k]/matrix[k][k]
14:                for j ← k + 1 to N − 1 do
15:                    matrix[i][j] ← matrix[i][j] − matrix[i][k] · matrix[k][j]
16:                end for
17:                numUpdates[i].fetch_add(1)
18:            end if
19:        end for
20:    end for
21: end procedure
```

4.1 `runner()`

The pseudo-code of the runner function is shown above. This is the function executed by each thread. The vector `toCheck` contains the list of rows that need to be updated by the corresponding owner thread. The algorithm itself is extremely similar to the sequential counterpart. The main difference is in how the k^{th} column is prepped. In the parallel algorithm, the prepping is done by the corresponding thread right before its own rows are updated. Apart from this, the only other difference is the additional while loop that has been added to ensure synchronization between threads as discussed earlier.

4.2 `sequential_LU()`

This is a simple function that takes the matrix as input, creates a copy of the input matrix, and performs in-place sequential LU factorization on the copied matrix. The modified matrix is then returned. The code follows the exact algorithm for sequential decomposition that has been stated earlier.

4.3 `generateMatrix()`

The function is used for generating strictly column diagonally dominant square matrices of input size N. A column diagonally dominant matrix satisfies one rule: $|a_{jj}| > \sum_{i \neq j} |a_{ij}|$ for all i. For each column, the function fills in random values for all non-diagonal entries and then takes the absolute sum of these random values. The function now fills a random number strictly larger than this sum in the diagonal position, thus generating a column diagonally dominant matrix.

5 Optimizations

- **Dynamic Load Balancing:** Dynamic load balancing can address the problem of workload imbalances and mitigate potential idle times. In this context, we can implement a system in which each thread involves each thread autonomously identifying and updating unprocessed rows. So, the threads are dynamically selecting rows that need to be updated, leading to more efficient resource utilization.
- **Pivoting Considerations:** The current implementation does not perform pivoting due to its complexity and computational cost. The code takes a safer approach by assuming column diagonally dominant matrices, thus avoiding the need for pivoting. However, there is a rationale for extending the code to incorporate pivoting. The major problem faced while implementing pivoting is the data movement, especially for large matrices. A pragmatic solution in this scenario is to swap rows *implicitly*, without directly altering their positions in memory.
- **Block LU Decomposition:** Block LU Decomposition is another popular technique that involves dividing the matrix into smaller contiguous blocks and performing LU decomposition on each block concurrently. The block structure creates a possibility of parallelism. This implementation is known for improved cache utilization. It would be interesting to compare the performance of the current algorithm with the block LU decomposition algorithm.

6 Results

Table 1: Matrix size (N) vs Number of threads (p)

Time taken (in seconds)								
	Sequential	**2**	**4**	**8**	**16**	**32**	**64**	**128**
100	0.006	0.005	0.004	0.025	0.549	1.052	2.28	4.51
500	0.8	0.504	0.324	0.211	1.969	5.051	11.247	23.521
1000	6.18	3.543	2.433	1.35	4.93	10.016	22.488	48.43
1500	26.576	10.441	7.46	4.017	9.093	15.551	33.417	74.992
2000	135.618	39.581	25.142	13.969	18.877	23.996	45.489	97.613
2500	263.831	77.173	47.219	26.559	34.082	34.709	58.265	122.093
3000	472.471	133.147	81.069	45.307	51.563	51.571	74.999	148.052

- As the matrix size increases, the speedup achieved by parallel execution becomes more pronounced. Larger matrices demonstrate more significant speedup.
- However, the speedup gains diminish as the number of threads increases. Beyond a certain point, the overhead of managing multiple threads (context switch) impacts overall performance.
- Despite the additional overhead, it has been observed that for larger matrices, a parallel program utilizing a large number of threads outperforms a sequential program.

- From the results, we can observe that there is an optimal number of threads at which the speedup is maximized. This occurs when there is a balance between parallel processing speedup and associated overhead.
- Larger matrices require more computational effort, resulting in higher execution time for both sequential and parallel programs.

7 References

- https://en.wikipedia.org/wiki/LU_decomposition
- https://courses.engr.illinois.edu/cs357/su2013/lectures/lecture07.pdf
- https://math.stackexchange.com/questions/1334983/gauss-elimination-difference-between-partial-and-complete-pivoting#:~:text=You%20are%20basically,that%20is%20prohibitive.
- https://arxiv.org/pdf/math/0506382v1.pdf
- https://en.wikipedia.org/wiki/Diagonally_dominant_matrix