Uppsala University

High Performance and Parallel Computing

1TD064

June 2024

Block LU-factorization

Author:
Seyedehmoniba Ravan

Introduction

The investigation of LU factorization marks a significant milestone in numerical linear algebra, offering a powerful method for solving systems of linear equations. LU factorization decomposes a matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U), enabling efficient solutions to linear systems and facilitating various numerical computations.

One of the advantage of LU decomposition lies in its ability to factorize matrix A only once, regardless of the number of different vectors b you need to solve for. This means that if you have multiple sets of b values but the same matrix A, you can reuse the LU decomposition of A to solve for each set of b efficiently. Understanding its origins and applications sheds light on its importance in modern computational mathematics.

As computational demands grow, handling large datasets becomes increasingly challenging. Matrices serve as a foundational structure in representing data, and manipulating them efficiently is crucial for computational tasks. However, traditional LU factorization methods face limitations when confronted with large matrices due to memory constraints and computational overhead.

To address these challenges, we delve into implementing block LU factorization in C using OpenMP. Enter block LU factorization, an extension of LU factorization, it splits matrices into smaller blocks, enabling parallel computation and efficient memory utilization.

Problem Description

The primary difference with block LU factorization is that it operates on blocks of the matrix rather than individual elements. In block LU factorization, instead of decomposing the entire matrix into L and U, the matrix is partitioned into blocks, and each block is decomposed into lower and upper triangular components. This decomposition is performed in a way that preserves the structure of the original matrix.

The original matrix A is partitioned into square blocks. The size of the blocks is chosen based on the problem's characteristics and computational considerations:

$$A = \begin{bmatrix} A_{00} & A_{01} & \dots & \dots \\ A_{10} & A_{11} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

The blocks for lower triangular matrix L and blocks for upper triangular matrix U are defined as follows:

$$L = \begin{bmatrix} L_{00} & 0 & 0 & \dots \\ L_{10} & L_{11} & 0 & \dots \\ L_{20} & L_{21} & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

$$U = \begin{bmatrix} U_{00} & U_{01} & U_{02} & \dots \\ 0 & U_{11} & U_{12} & \dots \\ 0 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

The block-wise LU factorization proceeds as follows:

1. **Factorization**: In the i-th iteration, the A_{ii} block is factorized into a lower triangular block L_{ii} and an upper triangular block U_{ii}: for exmaple for i=1, $A_{11} = L_{11} \cdot U_{11}$.

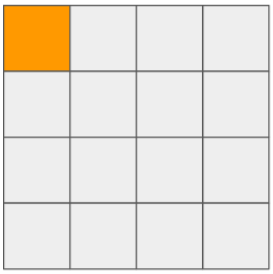

Figure 1: Factorization

2. **Solve linear equations**: We compute the lower triangular blocks L_{ji} and upper triangular blocks U_{ij} in a linear equation system:

$$L_{ii} \cdot U_{ij} = A_{ij} \quad \text{for all block in the same row of i}$$

$$L_{ji} \cdot U_{ii} = A_{ji} \quad \text{for all blocks in the same column of i}$$

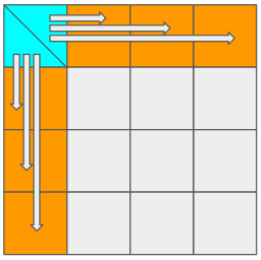

Figure 2: Solve linear equations

3. **Updating other blocks in** A: As we know

$$A_{i+1,i+1} = L_{i+1,i} \cdot U_{i,i+1} + L_{i+1,i+1} \cdot U_{i+1,i+1}$$

therefore we can update blocks $A'_{i+1,i+1}, \ldots$ as follows:

$$A'_{i+1,i+1} = A_{i+1,i+1} - L_{i+1,i} \cdot U_{i,i+1}$$

And so on.

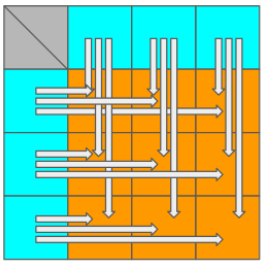

Figure 3: Updating other blocks in A

4. **Repeat**: Now that we need to find $L_{i+1,i+1} \cdot U_{i+1,i+1}$, we can recursively apply this process to the orange block, as illustrated in Figure 3.

However, implementing block LU factorization requires careful consideration of factors such as block size selection and load balancing to ensure optimal performance. Additionally, the computational benefits may vary depending on the specific characteristics of the matrix and the hardware architecture being utilized.

Solution Method

In the first step, Task parallelism is implemented in the `block_lu` function, which orchestrates the blocking of the matrix, the LU factorization of each block, and the subsequent forward and backward substitutions.

OpenMP directives are used to manage variable states across parallel tasks, enabling efficient parallel execution without compromising the correctness of the results.

```python
def block_lu(N, block_size, A):
    """
    Perform LU decomposition on a matrix A using a block-wise approach.

    Parameters:
    - N: int, size of the square matrix
    - block_size: int, size of each block
    - num_block: int, number of blocks
    - A: numpy.ndarray, input matrix to be decomposed

    Returns:
    None
    """
    for idx in range(0, N, block_size):
        block_kk = A[idx: idx + block_size, idx: idx + block_size]
        lu(block_kk)

        for i in range(idx + block_size, N, block_size):
            block_ik = A[i: i + block_size, idx: idx + block_size]
            back_substitution(block_kk, block_ik)

        for j in range(idx + block_size, N, block_size):
            block_kj = A[idx: idx + block_size, j: j + block_size]
            forward_substitution(block_kk, block_kj)

        for i in range(idx + block_size, N, block_size):
            for j in range(idx + block_size, N, block_size):
                block_ij = A[i: i + block_size, j: j + block_size]
                block_ik = A[i: i + block_size, idx: idx + block_size]
                block_kj = A[idx: idx + block_size, j: j + block_size]
                block_ij -= block_ik @ block_kj
```

Figure 4: Block LU driver code

We apply the serial LU factorization algorithm to decompose each block of the matrix. Although the time complexity of the algorithm is $O(n^3)$, applying it to smaller blocks significantly reduces the computation time, and we leverage parallelization to further enhance performance. The following figure illustrates the `lu` function, which performs LU factorization on blocks of a matrix using OpenMP for parallelization.

```
void lu(int idx, double* A, int n, int block_size) {
    /*
    Perform LU decomposition on a block of the matrix A,
    where the block is defined by the starting index `idx` and
    the block size `block_size`. The matrix A is overwritten with
    the LU decomposition result.

    Parameters:
        idx : int
            Starting index for the block in the matrix A.
        A : double pointer
            Pointer to the matrix A (size n x n).
        n : int
            Size of the matrix A.
        block_size : int
            Size of the block for LU decomposition.
    */
    for (int k = 0; k < block_size; k++) {
        // Vectorized division step
        #pragma omp simd
        for (int i = k + 1; i < block_size; i++) {
            if (A[(k + idx) * n + (k + idx)] == 0.0 )
                A[(i + idx) * n + (k + idx)] = 0.0;
            else
                A[(i + idx) * n + (k + idx)] /= A[(k + idx) * n + (k + idx)];
        }
        #pragma omp for schedule(dynamic)
        for (int i = k + 1; i < block_size; i++) {
            for (int j = k + 1; j < block_size; j++) {
                A[(i + idx) * n + (j + idx)] -= A[(i + idx) * n + (k + idx)] * A[(k + idx) * n + (j + idx)];
            }
        }
    }
}
```

Figure 5: LU factorization code

In the second step, we solve the equations using backward substitution since U_{ii} is upper triangular. Backward substitution is applied to solve the linear equations for the corresponding row, as depicted in Figure 6.

```
void back_substitution(int idx_i, int idx_j, double* A, int n, int block_size) {
    /*
    Perform back substitution to solve XU = Y for X,
    where U is an upper triangular matrix and Y is the
    matrix to be updated. The block of U and Y is defined by
    the indices `idx_i` and `idx_j`, respectively.

    Parameters:
        idx_i : int
            Starting row index of the block in matrix U.
        idx_j : int
            Starting column index of the block in matrix Y.
        A : double pointer
            Pointer to the matrix A, which contains the upper triangular matrix U.
        n : int
            Size of the matrix A.
        block_size : int
            Size of the block for back substitution.
    */
    #pragma omp parallel for
    for (int i = 0; i < block_size; i++) {
        for (int j = 0; j < block_size; j++) {
            double y = A[(i + idx_j) * n + (j + idx_i)];
            for (int k = 0; k < j; k++) {
                y -= A[(i + idx_j) * n + (k + idx_i)] * A[(k + idx_i) * n + (j + idx_i)];
            }
            if (A[(j + idx_i) * n + (j + idx_i)] == 0)
                A[(i + idx_j) * n + (j + idx_i)] = 0;
            else
                A[(i + idx_j) * n + (j + idx_i)] = y / A[(j + idx_i) * n + (j + idx_i)];
        }
    }
}
```

Figure 6: Backward substitution code

Next, we solve the equations for L_{ii}, which is lower triangular, using forward substitution, as shown in Figure 7. Forward and backward substitution can solve an equation system $Ax = b$ with a time complexity of $O(n^2)$, where n is the size of A. However, in this context, we are solving $AX = B$, where X and B are matrices rather than vectors, resulting in a time complexity of $O(n^3)$.

```
void forward_substitution(int idx_i, int idx_j, double* A, int n, int block_size) {
    /*
    Perform forward substitution to solve LX = Y for X,
    where L is a lower triangular matrix and Y is the matrix
    to be updated. The block of L and Y is defined by the indices
    `idx_i` and `idx_j`, respectively.

    Parameters:
        idx_i : int
            Starting row index of the block in matrix L.
        idx_j : int
            Starting column index of the block in matrix Y.
        A : double pointer
            Pointer to the matrix A, which contains the lower triangular matrix L.
        n : int
            Size of the matrix A.
        block_size : int
            Size of the block for forward substitution.
    */
    #pragma omp parallel for
    for (int j = 0; j < block_size; j++) {
        for (int i = 0; i < block_size; i++) {
            double y = A[(i + idx_i) * n + (j + idx_j)];
            for (int k = 0; k < i; k++) {
                y -= A[(i + idx_i) * n + (k + idx_i)] * A[(k + idx_i) * n + (j + idx_j)];
            }
            if (A[(i + idx_i) * n + (i + idx_i)] == 0.0)
                A[(i + idx_i) * n + (j + idx_j)] = 0.0;
            else
                A[(i + idx_i) * n + (j + idx_j)] = y;
        }
    }
}
```

Figure 7: Forward substitution code

In the subsequent step, we address the bottleneck of updating the rest of the matrix, which is the most computationally intensive part. This update is performed block by block, as illustrated in Figure 8. To optimize performance, the matrix multiplication function, which is the most time-critical part of the algorithm, was vectorized.

```
#pragma omp for collapse(2) schedule(dynamic)
for (int i = idx + block_size; i < N; i += block_size) {
    for (int j = idx + block_size; j < N; j += block_size) {
        //


        double* temp = (double*)malloc(block_size * block_size * sizeof(double));
        matrix_multiply(idx, i, j, A, temp, N, block_size);


        #pragma omp simd collapse(2)
        for (int ii = 0; ii < block_size; ii++) {
            for (int jj = 0; jj < block_size; jj++) {
                // double* block_ij = &A[i * N + j];
                A[(ii + i)* N + (jj + j)] -= temp[ii * block_size + jj];
            }
        }
        free(temp);
    }
}
```

Figure 8: Updating the rest of the matrix code

To facilitate a deeper understanding and simplify the comparison, the serial version of the algorithm was initially implemented in Python before being gradually ported to C. It was important to first understand and verify the correctness of the algorithm's serial logic before parallelizing it.

In the C implementation, the choice was made to store the matrix A as a one-dimensional array using row-major order. In C, arrays are stored in row-major order by default, meaning that elements of a 2D array are stored row by row in memory. This decision was made for simplicity and efficiency, as it simplifies memory management and ensures better cache locality.

The process of parallelizing the algorithm involved strategically adding OpenMP directives to the code. Initially,`#pragma omp parallel for` was added to parallelize outer loops, enabling parallel execution of loop iterations. For small nested loops, the directive `#pragma omp for collapse (2) schedule (dynamic)` was employed. This directive collapses two nested loops into one and dynamically schedules iterations across threads to balance the workload, enhancing parallel performance.

Various approaches were experimented with, to optimize the code further. This included identifying opportunities for loop vectorization using `#pragma omp simd`. By vectorizing loops, SIMD (Single Instruction, Multiple Data) operations were utilized, improving computational efficiency for data-parallel tasks.

Also I tried to use pararel computing in back/forward substitution but it caused the overhead and increased the runtime but eventually each equation system is solved in one thread.

Care was taken to ensure that each process overwrites the matrix A to avoid excessive memory consumption. It means the final results can be shown in ma-

trix. Additionally, memory allocation and deallocation were managed dynamically for temporary variables, optimizing memory usage and avoiding memory leaks.

Initially, the solution was limited to cases where the block size was divisible by the matrix size, leading to incorrect results for other cases. This issue was resolved by padding the matrix with zeros to the nearest size divisible by the block size, ensuring accurate results across all scenarios.

Experiments

We conducted a series of experiments to evaluate the performance and correctness of the LU-factorization algorithm in both serial and parallel implementations. These experiments involved testing matrices of various sizes and block sizes while varying the number of threads used in parallel execution. The key findings from these experiments are summarized below:

One of the most notable improvements in the parallel implementation was observed after incorporating vectorization (SIMD). For a matrix size of $N = 10000$ and a block size of 100, the runtime was reduced significantly from approximately 440 seconds to 131 seconds.

Another critical optimization involved modifying the code to accumulate results in a local variable `sum` within the loop. This change reduced memory accesses, improved cache efficiency, and significantly decreased the runtime. Specifically, for $N = 10000$ and a block size of 100, the runtime was further reduced from 131 seconds to 46 seconds, representing a nearly 30% reduction in execution time.

Hardware Specifications

- **Processor:** Intel Core i7-7700HQ (2.80 GHz)
- **Number of Cores:** 4
- **Number of Threads:** 8 (Logical Processor)
- **RAM:** 16 GB DDR4

These hardware specifications played a crucial role in determining the scalability and efficiency of the parallel algorithms.

The performance evaluation involved comparing the execution times of the parallel LU-factorization algorithm with its serial counterpart across matrices of varying sizes. Table 1 presents a comparison between the serial and parallel implementations:

The results indicate a significant performance improvement with the parallel implementation, especially for larger matrices. This demonstrates the effectiveness of parallel LU-factorization using block-wise decomposition in enhancing computational efficiency.

serial or parallel	N	block_size	runtime (s)
serial	5000	-	31
parallel	5000	100	5.63
serial	10000	-	270
parallel	10000	100	44.07

Table 1: Comparison of Parallel algorithm vs Serial algorithm

Figure 9 shows how runtime increases with matrix size for a fixed block size and number of threads.

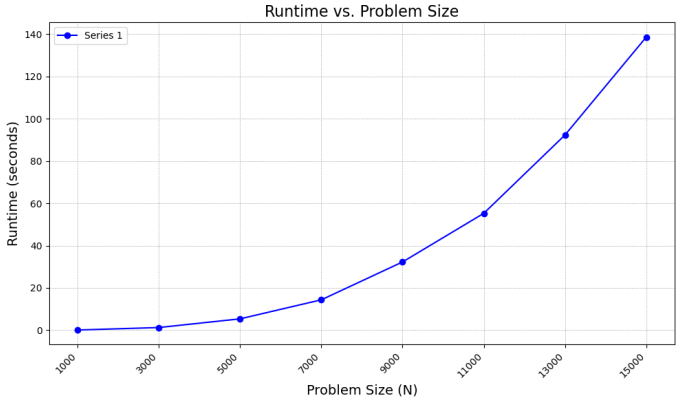

Figure 9: Comparison of Parallel Block LU Factorization Runtimes for Different Matrix Sizes

Figure 10 illustrates the impact of different block sizes on runtime, providing insights into the optimal block size selection for the block LU factorization algorithm. The experimental results reveal that a block size of 100 achieves the best runtime performance, with a recorded runtime of 32.28s for a matrix size of 9000. This finding underscores the significance of selecting an appropriate block size to maximize computational efficiency and minimize runtime. A block size of 100 strikes a balance between granularity and computational overhead, allowing for efficient utilization of cache memory and minimizing data transfer overhead. As a result, it emerges as the optimal choice for achieving optimal performance in block LU factorization.

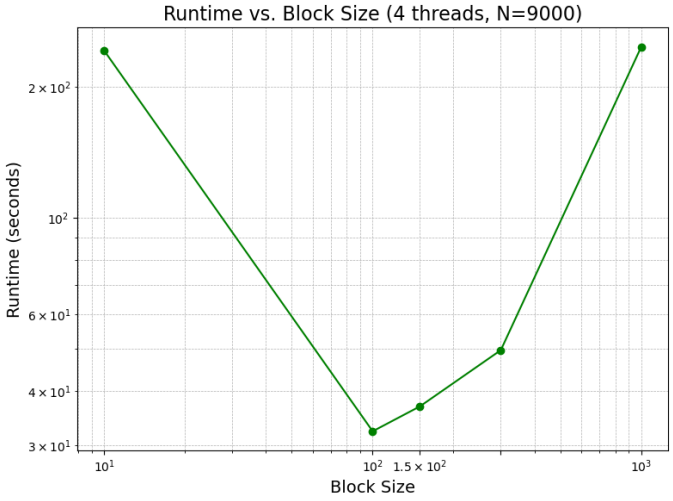

Figure 10: Comparison of Parallel Block LU Factorization Runtimes for Different Block Sizes in log-log scale for clearer visualization

Furthermore, Figure 11 demonstrates the effect of varying the number of threads on runtime for a fixed matrix size and block size. The results indicate that the best runtime achieved was 24.83s, observed when utilizing 8 threads. This highlights the importance of optimizing thread utilization to achieve maximum parallel efficiency.

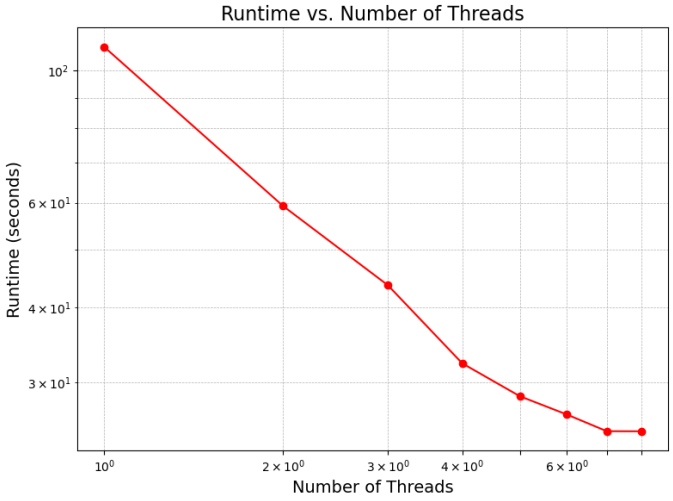

Figure 11: Comparison of Parallel Block LU Factorization Runtimes for Different number of threads (Matirx Size = 9000, Blocksize = 100) in log-log scale

We also examined the algorithm's performance with varying block sizes using the Cachegrind tool to assess cache efficiency.

Table 2: Performance Metrics for Different Block Sizes using cachegrind tool (matrix size = 3000, n-threads = 4)

Block Size	Execution Time (s)	D1 Miss Rate (%)	LLd Miss Rate (%)	LL Miss Rate (%)
100	230.48	8.5	0.1	0.0
250	202.85	7.9	0.1	0.0
500	259.54	47.3	0.1	0.0
1000	335.00	57.7	4.6	1.2
3000	150.00	26.0	15.2	4.5

For all cases, the L1 and LL cache miss rates are very low, nearly 0.00%. The block size of 250 yielded the best overall performance in terms of execution time (202.85 seconds). For smaller block sizes (100, 250), the cache miss rates are lower, particularly for L1 and LL cache misses, indicating better cache utilization. As block size increases, cache miss rates rise significantly (especially for block sizes 1000 and 500). However, for block size 3000, the LL miss rate decreases significantly, suggesting that at this size, the algorithm is better optimized for cache usage, possibly due to better alignment with higher levels of cache.

The results emphasize the importance of tuning block size for cache efficiency, particularly in performance-critical applications where execution time is crucial.

Conclusions

The optimization efforts focused on parallelizing LU factorization, and the experiments clearly demonstrated that block-wise decomposition can significantly enhance the efficiency of LU factorization for large matrices. The parallel implementation achieved substantial speedups compared to the serial version, especially for larger matrices, highlighting the effectiveness of parallel processing in improving computational performance.

Further optimization opportunities include exploring alternative matrix multiplication algorithms, such as Strassen's algorithm, which could enhance the performance of the most computationally intensive parts of the algorithm. Additionally, addressing cases where LU decomposition may not be feasible could provide further avenues for improvement. These findings underline the potential of parallel LU factorization for high-performance computing applications and suggest areas for future research and development.

References

[1] Wikipedia. *LU Decomposition*. n.d. URL: https://en.wikipedia.org/wiki/LU_decomposition.

[2] Wikipedia. *Block LU Decomposition*. n.d. URL: https://en.wikipedia.org/wiki/Block_LU_decomposition.

[3] Beny Neta and Heng-Ming Tai. "LU Factorization on Parallel Computers". In: *Computers & Mathematics with Applications* 11.6 (1985), pp. 573–579. ISSN: 0898-1221. DOI: 10.1016/0898-1221(85)90039-2. URL: https://doi.org/10.1016/0898-1221(85)90039-2.

[4] James Demmel, Nicholas Higham, and Robert Schreiber. "Block LU Factorization". In: (Mar. 2000).

[5] HPC2N. *Task-Based Parallelism*. 2021. URL: https://hpc2n.github.io/Task-based-parallelism/branch/spring2021/task-basics-lu/.