HPC Report
Cholesky factorization

Arduini Daniel 248727
Tezza Michael 248725

Contents

Abstract

TODO

1 Introduction

1.1 Problem Description

Solving systems of linear equations ($Ax = b$) is one of the most common problems in scientific computing. It is used in many fields, such as physics simulations, financial modeling, and engineering.

When the matrix A is **symmetric** (it is equal to its transpose, $A = A^T$) and **positive-definite**, the most efficient method to solve the system is the **Cholesky Factorization**. This algorithm decomposes the matrix A into the product of a lower triangular matrix L and its transpose L^T:

$$A = LL^T \tag{1}$$

Once we have calculated L, we can solve the original system $Ax = b$ very quickly. However, calculating L is expensive. The computational complexity is $\mathcal{O}(N^3)$, which means that if we double the size of the matrix, the time required increases by 8 times. For very large matrices, a single processor is not fast enough, so we must use parallel computing to distribute the work across many processors.

1.2 Sequential Algorithm Analysis

The standard algorithm to compute L works column by column, from left to right. For a matrix of size $N \times N$, the algorithm consists of three main steps for each column k:

1. **Diagonal Update:** Calculate the square root of the diagonal element ($L_{k,k}$).
2. **Column Update:** Divide the elements below the diagonal in the current column by the diagonal element.
3. **Trailing Matrix Update:** Update the rest of the matrix (the submatrix to the right) using the values calculated in the current column.

The pseudocode for the serial implementation is shown below:

Algorithm 1 Sequential Cholesky Factorization

```
1: for k = 0 to N − 1 do
2:     A[k][k] = √A[k][k]                          ▷ Step 1: Diagonal
3:     for i = k + 1 to N − 1 do
4:         A[i][k] = A[i][k]/A[k][k]               ▷ Step 2: Column
5:     end for
6:     for j = k + 1 to N − 1 do
7:         for i = j to N − 1 do
8:             A[i][j] = A[i][j] − A[i][k] ∗ A[j][k]   ▷ Step 3: Trailing Update
9:         end for
10:    end for
11: end for
```

2 Parallel Design

In this section, we analyze the design space for parallelizing the Cholesky factorization. We explore different strategies for data distribution and processor organization, highlighting the limitations of naive approaches and justifying our final choice.

2.1 Evaluation of Intermediate Strategies

The core challenge is mapping the matrix onto the processors efficiently. We considered some evolutionary steps to identify the optimal strategy.

2.1.1 Step 1: 1D Row Decomposition (The Naive Approach)

The simplest strategy is to slice the matrix horizontally. We assign contiguous groups of rows to each processor. For example, Processor 1 gets the first row, Processor 1 gets the second row...

Visualization: If we strictly assign one row per processor:

0	0	0	0	← P0 (Finished quickly)
1	1	1	1	← P1
2	2	2	2	← P2
3	3	3	3	← P3 (Idle start, busy end)

Figure 1: Visualization of 1D Row Decomposition.

- **Limitation:** As the algorithm moves down the diagonal (from row 0 to 3), the top processors finish their work and become idle. Processor 0 does nothing after the first step. This leads to **poor load balancing**.

2.1.2 Step 2: 1D Cyclic Decomposition

To solve the load balancing issue, we can distribute rows in a round-robin fashion (like dealing cards).

0	0	0	0	← Assigned to P0
1	1	1	1	← Assigned to P1
0	0	0	0	← Assigned to P0 (Cyclic)
1	1	1	1	← Assigned to P1 (Cyclic)

Figure 2: Visualization of 1D Cyclic Decomposition.

- **Limitation:** While computation is balanced, **communication is inefficient**. After factorizing a diagonal block, the owner must broadcast the result so that other processors can update the column elements below it. Since the rows are distributed cyclically among *all* P processors, almost every processor owns a part of that column.

Consequently, the broadcast becomes a global operation involving the entire cluster. This wastes network bandwidth because it forces a global synchronization, whereas a more optimized design would restrict this traffic to only a small vertical subset of processors.

2.1.3 Step 3: Introduction of Block Algorithms

Before optimizing communication, we address the computational efficiency of the node.

The Problem (Memory Hierarchy): Accessing data from the main memory (RAM) is significantly slower than performing calculations in the CPU. If the algorithm processes single elements one by one, the processor frequently waits for data to arrive from RAM.

The Solution (Blocking): To mitigate this latency, we divide the matrix into square sub-matrices called **blocks** (e.g., 64×64). When a block is accessed, it can remain in the CPU's cache across multiple operations. Since the Cholesky algorithm performs many operations on the same data, working on blocks allows the processor to reuse the data residing in the Cache multiple times before needing to access the RAM again.

(a) Element-wise Access (b) Block Access

Figure 3: Comparison between (a) accessing single elements vs (b) loading a full block.

- **Impact:** This improves the ratio of computation to memory access, resulting in higher computational throughput, measured in GFLOPS (Giga Floating Point Operations Per Second), compared to the element-wise approach.

2.2 Proposed Solution: 2D Block-Cyclic Decomposition

By combining the strengths of the previous steps (Load Balancing + Blocking), we implemented the **2D Block-Cyclic Decomposition**. This is the optimal strategy because it solves the communication bottleneck of Step 2 by arranging processors in a grid.

2.2.1 Process Grid and Distribution Logic

To implement the 2D decomposition, we first organize the available processors into a logical grid. For example, if we have $P = 4$ processors, we arrange them into a 2×2 grid ($R = 2, C = 2$).

Each processor is identified by its coordinates (P_{row}, P_{col}):

$$\begin{matrix} (0,0) & (0,1) \\ (1,0) & (1,1) \end{matrix} \quad \longrightarrow \quad \begin{matrix} P_0 & P_1 \\ P_2 & P_3 \end{matrix}$$

Mapping Blocks: We do not distribute single elements. We view the global matrix A as a grid of blocks, where each block contains $B \times B$ elements (e.g., 64×64). We map these blocks to processors in a round-robin (cyclic) fashion.

The owner of a block at global block index (i, j) is determined by:

$$\begin{aligned} P_{row} &= i \pmod R \\ P_{col} &= j \pmod C \end{aligned} \tag{2}$$

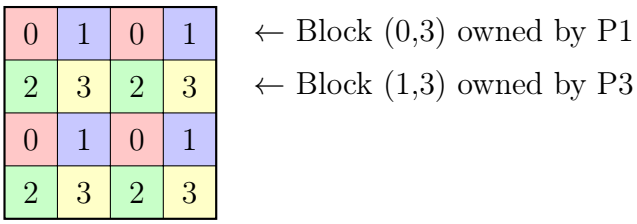

Figure 4: Visualization of 2D Block-Cyclic distribution. Each square represents a **data block** (e.g., 64×64 doubles), not a single element.

Why this mapping is effective: This distribution scatters the blocks owned by a single processor across the entire matrix. As seen in Figure 4, *Processor 0* owns blocks at grid positions $(0, 0), (0, 2), (2, 0)$, and $(2, 2)$. Even when the "active" part of the matrix shrinks (e.g., after the first few iterations), Processor 0 still holds valid blocks further down the matrix (at row 2), ensuring it remains busy and contributing to the computation.

Communication Efficiency: Beyond load balancing, this grid structure fundamentally changes the communication pattern. In a 1D distribution (Step 2), a broadcast involves all P processors. In this 2D grid, operations like broadcasting a diagonal block or a panel are restricted to the processors in a specific row or column (involving only $\sqrt{P}$ processors). This reduces network contention and significantly improves scalability as the cluster size grows.

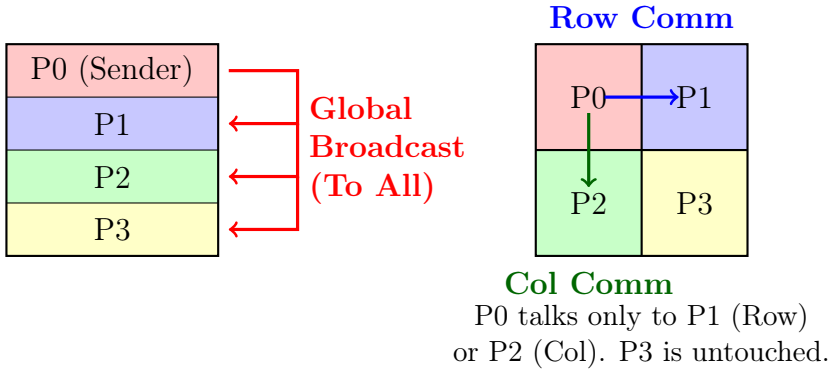

Figure 5: Communication Pattern Comparison. Left: In 1D, P0 must send to everyone. Right: In 2D, communication is restricted to the specific Row or Column subset.

2.3 Parallel Algorithm Workflow

With the data distributed across the 2D grid, the Cholesky factorization proceeds in iterations $k = 0 \dots N_{blocks} - 1$. In each iteration, we process one block column and update the rest of the matrix.

Since we are not using external linear algebra libraries, we implemented the standard Level-3 BLAS operations manually.

2.3.1 Phase 1: Factorize Diagonal Block (POTRF)

The process that owns the current diagonal block (P_{diag}) computes the Cholesky factorization of that single block locally.

$$A_{kk} \leftarrow \text{Chol}(A_{kk})$$

- **Action:** We perform a sequential Cholesky factorization on the local $B \times B$ block to obtain the lower triangular factor L_{kk}.
- **Communication:** P_{diag} broadcasts L_{kk} vertically to all other processors in the same **grid column**.

2.3.2 Phase 2: Update Panel (TRSM)

All processors in the current process column must update their blocks A_{ik} located below the diagonal. This corresponds to solving a triangular system of equations:

$$A_{ik} \leftarrow A_{ik}(L_{kk}^T)^{-1}$$

- **Action:** We implemented the **TRSM** (Triangular Solve with Multiple Right-Hand Sides) algorithm. Since L_{kk}^T is upper triangular, we solve the system $XL_{kk}^T = A_{ik}$ using forward substitution.
- **Communication:** Once updated, these blocks are broadcast horizontally to all processors in their respective **grid rows**.

2.3.3 Phase 3: Update Trailing Matrix (GEMM / SYRK)

Finally, all processors update their local portion of the remaining submatrix using the data received from the broadcasts.

$$A_{ij} \leftarrow A_{ij} - A_{ik}A_{jk}^T$$

- **Action:** We implemented the update using a unified loop structure restricted to $j \leq i$. This *exploits the matrix symmetry* to reduce computational cost. By updating only the unique blocks in the lower triangle, we avoid redundant calculations for the upper triangular part (which is logically identical but not stored/accessed).

2.4 Analysis of Parallelism and Dependencies

To understand the scalability of our design, we must analyze the **data dependencies** that dictate which operations can be performed in parallel and which must be serialized.

2.4.1 Data Dependencies

The Cholesky algorithm has a strict dependency chain:

1. **Diagonal Dependency:** The diagonal block A_{kk} must be factorized first. No other operation in the current iteration k can proceed until this is finished.
2. **Panel Dependency:** The blocks in the current column A_{ik} cannot be updated until they receive the factorized diagonal block L_{kk}.

3. **Trailing Matrix Dependency:** The update of any trailing block A_{ij} (where $i, j > k$) requires the corresponding updated blocks from the current panel (A_{ik} and A_{jk}).

2.4.2 The Opportunity for Parallelism

Despite these dependencies, the algorithm exposes massive parallelism in the *Trailing Matrix Update (Phase 3)*. Once the panel broadcast is complete, the update of each block A_{ij} in the trailing submatrix is *mathematically independent* of all other block updates in that submatrix.

$$A_{ij}^{(new)} = A_{ij}^{(old)} - A_{ik} \cdot A_{jk}^{T}$$

This independence allows every processor to update all its local blocks simultaneously without synchronization with neighbors.

2.4.3 Degree of Concurrency

The available parallelism (concurrency) varies by phase:

- **Phase 1 (Diagonal): Serial bottleneck.** Only 1 process is active.
- **Phase 2 (Panel): Partial Parallelism.** Only $\sqrt{P}$ processes (one column) are active.
- **Phase 3 (Trailing): Full Parallelism.** All P processes are active.

Since Phase 3 dominates the computational cost (performing $O(N^3)$ operations versus $O(N^2)$ for the panel), the highly parallel part hides the inefficiency of the serial and partial phases as the matrix size N grows.

3 Implementation Details

The proposed parallel design was implemented in C using a hybrid MPI + OpenMP approach. The codebase focuses on minimizing memory footprint and maximizing communication efficiency without relying on external linear algebra libraries (like BLAS or LAPACK).

3.1 Software Architecture and Process Grid

To manage the complexity of the distributed state, we encapsulated the simulation data in a `CholeskyContext` structure. The logical grid topology is managed by a custom `ProcGrid` structure, initialized during the setup phase.

- **Process Grid Topology:** We organize the P available MPI processes into a logical 2D grid of dimensions $P_{rows} \times P_{cols}$.
  - This grid is distinct from the data matrix; it represents the layout of execution units.
  - The dimensions are calculated dynamically: we attempt to create a square grid ($P_{rows} \approx \sqrt{P}$) to minimize the perimeter (and thus communication costs). If P is not a perfect square (e.g., $P = 8$), the initialization logic produces the most compact rectangular grid possible (e.g., 2×4).
- **Custom Communicators:** To implement the row and column broadcasts described in Section 2 efficiently, we utilize `MPI_Comm_split` to partition the global `MPI_COMM_WORLD` based on the process grid coordinates:
  - `row_comm`: Grouping all processes with the same P_{row} coordinate.
  - `col_comm`: Grouping all processes with the same P_{col} coordinate.

  This ensures that collective operations like `MPI_Bcast` are restricted strictly to the relevant subset of processes (a single row or column of the process grid) rather than involving the entire cluster.

3.2 Distributed Data Generation (Memory Scalability)

A critical optimization in our implementation is the **distributed generation** of the input matrix. Standard approaches often generate the full matrix on the root process and scatter it, which bottlenecks the maximum problem size to the RAM of a single node.

Instead, we implemented a parallel generation routine where each processor allocates and computes *only* the matrix elements belonging to its specific local blocks:

$$A_{ij} = \begin{cases} N + 1 & \text{if } i = j \\ \frac{1}{1+|i-j|} & \text{if } i \neq j \end{cases}$$

By using the block-cyclic mapping formulas to determine ownership, we ensure that the full matrix never exists in the memory of a single node. This grants our solution linear memory scalability, allowing us to handle problem sizes that are constrained only by the aggregate memory of the cluster rather than the capacity of individual nodes.

References