HPC Project

High Performance Computing

Authors:
HABAGNO Antonin
Léo GUERIN

Date:
November 19, 2025

1 Exercice 1

2 Exercice 2

2.1 Matrix-vector product

2.1.1 Introduction

The objective of this exercice is to analyze the performance characteristics of Matrix-Vector Multiplication specifically for tridiagonal matrices. We will implement a sequential baseline, parallelize it using OpenMP and MPI, and analyze the impact of the number of threads and processes on the execution time.
A tridiagonal matrix is a sparse matrix where non-zero elements exist only on the main diagonal, the super-diagonal, and the sub-diagonal.

2.1.2 Initial Implementation (Naive Storage)

In the initial phase, we utilized a standard dense matrix representation (int**) to store the tridiagonal matrix. Although the matrix is sparse, memory was allocated for all $N \times N$ elements, with zeros filled explicitly.

```
int** random_tridiagonal_matrix(int n) {
    // 1. Allocation of N pointers (Rows)
    int** matrix = malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        // 2. Allocation of N integers per row
        matrix[i] = malloc(n * sizeof(int));
    }

    // 3. Explicit initialization of zeros (O(N^2) operation)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = 0;
        }
    }

    // 4. Filling the 3 diagonals
    for (int i = 0; i < n; i++) {
        if (i > 0) matrix[i][i-1] = (rand() % 201) - 100; // Lower
        matrix[i][i] = (rand() % 201) - 100;              // Main
        if (i < n-1) matrix[i][i+1] = (rand() % 201) - 100; // Upper
    }
    return matrix;
}
```

While easy to implement, this approach has a space complexity of $O(N^2)$. For a tridiagonal matrix, only $3N - 2$ elements are significant. Allocating N^2 space leads to massive

memory wastage and limits the maximum achievable N before RAM saturation.
Despite the naive storage, the multiplication algorithm was optimized to respect the sparsity pattern. Instead of performing a full dot product (row $\times$ column) which would take $O(N^2)$ operations, we restricted the inner loop to strictly calculate the non-zero diagonals.

```
for (int i = 0; i < n; i++) {
    int sum = 0;
    // Optimization: Only access potentially non-zero elements
    if (i > 0) {
        sum += matrix[i][i-1] * vec[i-1]; // Sub-diagonal
    }
    sum += matrix[i][i] * vec[i];         // Main diagonal
    if (i < n-1) {
        sum += matrix[i][i+1] * vec[i+1]; // Super-diagonal
    }
    result[i] = sum;
}
```

This reduces the computational complexity to $O(N)$ instead of $O(N^2)$.

2.1.3 OpenMP Parallelization

We parallelized the optimized loop using pragma omp parallel for.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    int sum = 0;
    if (i > 0) sum += matrix[i][i-1] * vec[i-1];
    sum += matrix[i][i] * vec[i];
    if (i < n-1) sum += matrix[i][i+1] * vec[i+1];
    result[i] = sum;
}
```

2.1.4 Experimental Results ($N = 10,000$)

Implementation	Threads	Time (s)	Observation
Sequential	1	0.000129	Baseline
OpenMP	1	0.001288	10x Slower
OpenMP	2	0.000139	Slower than Seq
OpenMP	4	0.000231	Slower than Seq
OpenMP	8	0.000226	Slower than seq

The OpenMP implementation failed to provide speedup for this dataset size. This can be attributed to two factors:

1. For N=10,000, the total workload is roughly $30,000$ arithmetic operations. Modern CPUs can execute this in microseconds. The overhead introduced by OpenMP (creating threads, scheduling chunks, and synchronization barriers) is significantly higher than the computation time itself. As seen in the table, the 1-thread OpenMP run includes this overhead without any parallel gain, resulting in a 10x slowdown.

2. The $int**$ structure stores rows in non-contiguous memory locations. When multiple threads try to access matrix$[i]$ and vec, they compete for memory bandwidth. Furthermore, since the computation is extremely light, the CPU spends more time fetching data from RAM than calculating.

2.1.5 Limitations and Proposed Solution

To observe true parallel speedup, we must increase N significantly (e.g., $N > 10^7$). However, the current $int**$ storage makes this impossible :

1. For $N = 1000000$, an $N \times N$ integer matrix requires 10^{12} integers ≈ 4 TB of RAM.

2. This causes the program to crash or swap heavily before we can reach a problem size where OpenMP becomes efficient.

To enable large-scale testing and improve cache locality, we must abandon the int** representation.
We will transition to a Compressed Tridiagonal Storage format using three 1D arrays:

1. lower (size $N - 1$)

2. main (size N)

3. upper (size $N - 1$)

This method reduces space from $O(N^2)$ to $O(N)$. This allows testing $N = 100,000,000$ on standard hardware.