HPC Project

High Performance Computing

Authors:
HABAGNO Antonin
Léo GUERIN

Date:
December 2, 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	0.000309	Slower than Seq
OpenMP	4	0.000645	Slower than Seq
OpenMP	6	0.000190	Slower than Seq
OpenMP	8	0.000246	Slower than seq

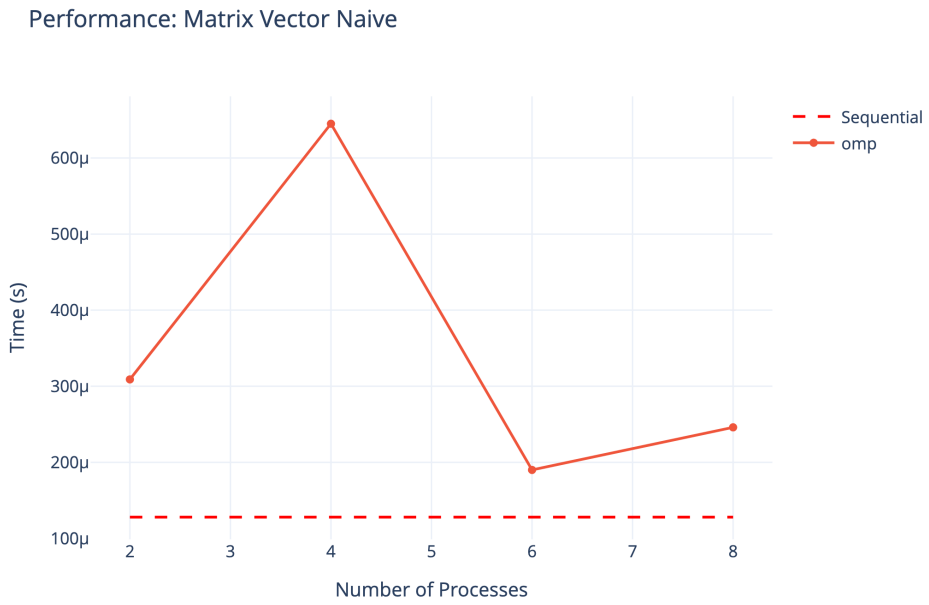

Figure 1: Times in function of the number of threads / procs

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.

2.1.6 OpenMP Parallelization with Optimized Storage

With the $O(N)$ storage, we were able to increase the problem size to $N = 100000000$. We applied OpenMP parallelism to the optimized loop.

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

2.1.7 MPI Parallelization with Optimized Storage

We implemented MPI parallelization to distribute the workload across multiple processes. The input vectors are decomposed into chunks, and each process is responsible for calculating a specific range of indices.

A critical challenge in decomposing Tridiagonal Matrix Multiplication is the dependency on neighbors:

$$y[i] = L[i-1] \times x[i-1] + D[i] \times x[i] + U[i] \times x[i+1] \tag{1}$$

To calculate index i, a process needs $x[i-1]$ and $x[i+1]$.

```
// Exchange with LEFT neighbor (Rank - 1) to get x[start-1]
if (rank > 0) {
  MPI_Sendrecv(&local_vec[0], 1, MPI_INT, rank - 1, 0,
               &ghost_left, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
}

// Exchange with RIGHT neighbor (Rank + 1) to get x[end+1]
if (rank < size - 1) {
  MPI_Sendrecv(&local_vec[local_n - 1], 1, MPI_INT, rank + 1, 0,
               &ghost_right, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD, &status);
}
```

2.1.8 Experimental Results with optimized storage ($N = 100,000,000$)

Implementation	Threads	Time (s)	Speedup
Sequential	1	0.380672	Baseline
OpenMP	2	0.221757	1.7x Speedup
OpenMP	4	0.119725	3.2x Speedup
OpenMP	6	0.080580	4.7x Speedup
OpenMP	8	0.078778	4.8x Speedup
MPI	2	0.396344	0.9x Speedup
MPI	4	0.279268	1.4x Speedup
MPI	6	0.242146	1.6x Speedup
MPI	8	0.275798	1.4x Speedup

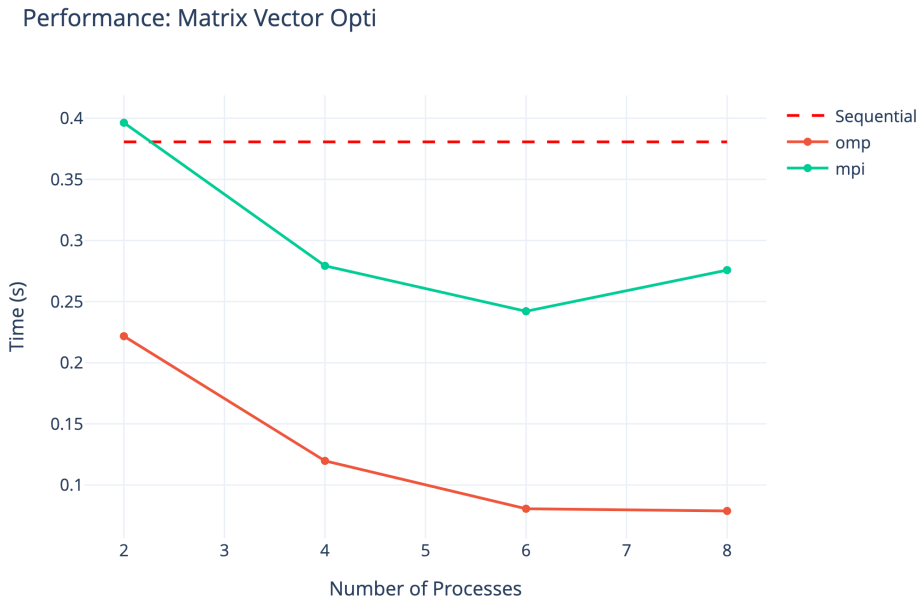

Figure 2: Times in function of the number of threads / procs

Contrary to OpenMP, the MPI implementation shows poor performance on a single shared-memory machine.

1. Communication Overhead: The operations $MPI\_Scatterv$ and $MPI\_Gatherv$ require Rank 0 to serialize and transmit the entire dataset (approx 1.2 GB for N=108) to other processes. In a shared-memory context, this involves expensive memory-to-memory copies that are slower than direct pointer access used in OpenMP.

2. Computation-to-Communication Ratio: The matrix-vector product is an O(N) operation with very few arithmetic operations per byte loaded. The time saved by parallelizing the calculation is overshadowed by the time spent distributing the data.

2.1.9 Matrix-vector multiplication conclusion

This study demonstrates that Data Structures are the primary driver of performance in sparse linear algebra. Switching from int** to Compressed Storage was necessary to even run the benchmark.

1. OpenMP proved highly effective, offering a 5x speedup on 8 threads, limited only by the physical memory bandwidth of the machine.

2. MPI, while correctly implemented with Halo Exchanges, is not suitable for this specific workload on a single node due to the high cost of data distribution relative to the low computational intensity of the kernel. MPI would only become beneficial if the problem size N exceeded the RAM of a single machine, requiring distribution across a cluster.

2.2 Matrix Power

2.2.1 Introduction

In this section, we extend our analysis to matrix exponentiation, specifically computing A^2 and A^3 for a tridiagonal matrix A. Unlike matrix-vector multiplication, squaring a sparse matrix increases its bandwidth.

- If A is tridiagonal (bandwidth 1), A^2 is pentadiagonal (bandwidth 2).
- A^3 is heptadiagonal (bandwidth 3).

To maintain $O(N)$ memory complexity, we implemented custom data structures to store only the non-zero diagonals.

2.2.2 Sequential Implementation

We implemented specific functions to compute the diagonals of the resulting matrices directly. For A^2, the element $(A^2)_{ij}$ is given by $\sum_k A_{ik}A_{kj}$. Due to the sparsity of A, only a few terms in this sum are non-zero.

```
// Example: Computing the main diagonal of A^2
int val = M[i] * M[i];
if (i > 0) val += L[i - 1] * U[i - 1];
if (i < n - 1) val += U[i] * L[i];
R->main[i] = val;
```

For A^3, we compute $A^3 = A \times A^2$. This involves multiplying a tridiagonal matrix by a pentadiagonal matrix.

2.2.3 OpenMP Parallelization

Since the computation of each row of the result matrix is independent, we can trivially parallelize the outer loops using OpenMP. We used `#pragma omp parallel for` to distribute the rows among threads.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // Compute diagonals for row i
    // ...
}
```

2.2.4 Experimental Results ($N = 100,000,000$)

We measured the execution time for computing A^2 and A^3 with varying numbers of OpenMP threads.

Results for A^2:

Implementation	Threads	Time (s)	Speedup
Sequential	1	1.239	Baseline
OpenMP	2	0.747	1.65x
OpenMP	4	0.383	3.23x
OpenMP	6	0.277	4.47x
OpenMP	8	0.290	4.27x

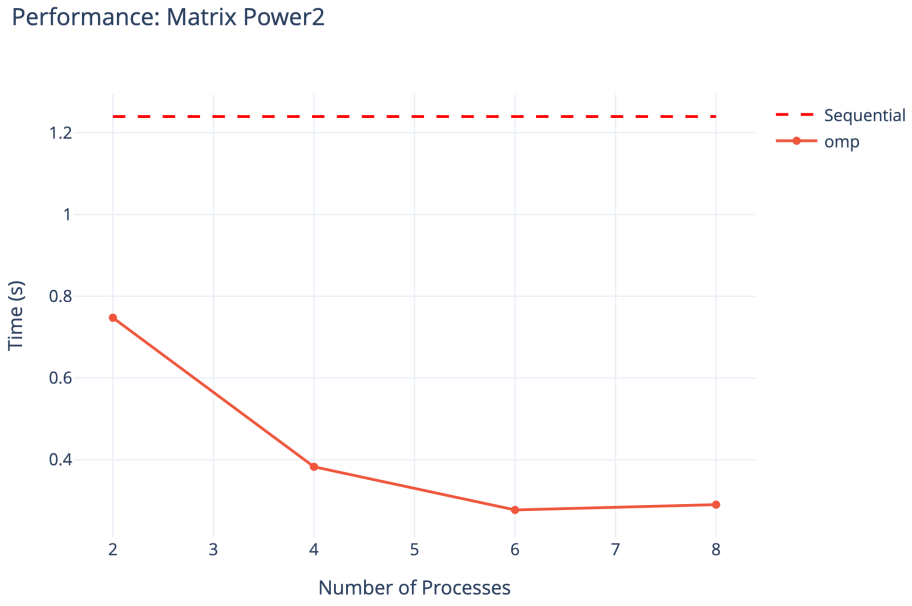

Figure 3: Execution time for A^2 vs number of threads

Results for A^3:

Implementation	Threads	Time (s)	Speedup
Sequential	1	8.143	Baseline
OpenMP	2	3.939	2.06x
OpenMP	4	2.028	4.01x
OpenMP	6	1.469	5.54x
OpenMP	8	1.272	6.40x

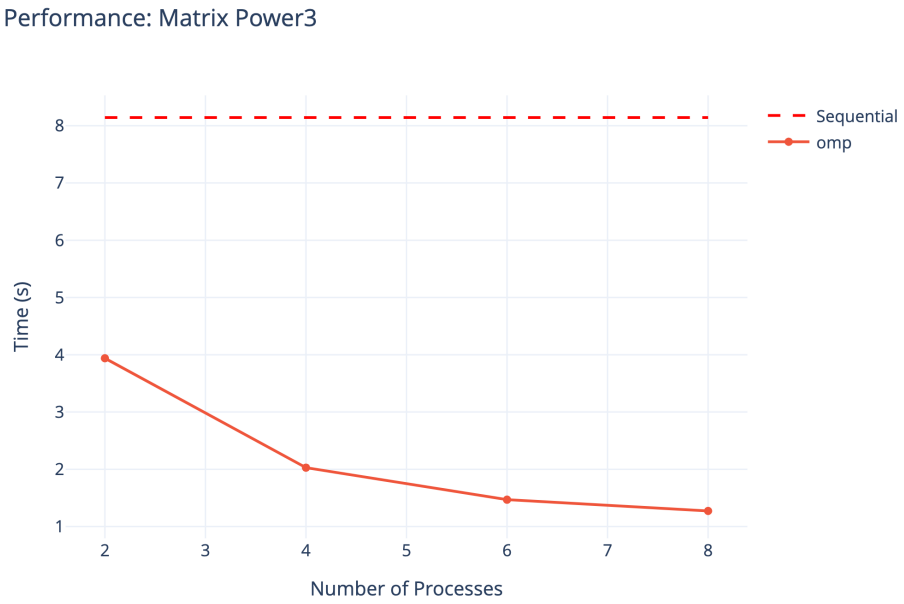

Figure 4: Execution time for A^3 vs number of threads

2.2.5 Analysis

For A^2, we observe good scalability up to 6 threads. At 8 threads, performance slightly degrades, likely due to memory bandwidth saturation, as the arithmetic intensity of A^2 is relatively low. For A^3, which involves more arithmetic operations per element (multiplying tridiagonal by pentadiagonal), the scalability is better, achieving a 6.4x speedup on 8 threads. This suggests that the more computationally intensive the kernel, the better the parallel efficiency on this hardware.