HPC Project

High Performance Computing

https://github.com/leogue/hpc_project

Authors:
CHABAGNO Antonin
Léo GUERIN

Date:
December 19, 2025

1 Exercise 1: Series Summation

1.1 Objective

The objective of this exercise is to calculate the sum of the series for a given integer $n > 1$:

$$S = \sum_{i=1}^{n} \frac{1}{i(i+1)} \tag{1}$$

We aim to verify the convergence of this series as n increases and to parallelize the computation using OpenMP and MPI. We will then analyze the performance and speedup obtained on a multicore architecture.

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1} \right) = 1 - \frac{1}{n+1} \xrightarrow{n \to \infty} 1 \tag{2}$$

1.2 Implementation

1.2.1 Sequential Implementation

The sequential version is a straightforward loop summing the terms.

```
double fn(int i) {
    return 1.0 / ((double)i * (i + 1));
}

double sum(int n) {
    double total = 0;
    for (int i=1; i<n; i++) {
        total += fn(i);
    }
    return total;
}
```

1.2.2 OpenMP Parallelization

We parallelized the loop using the `omp parallel for` directive with a reduction on the `total` variable to avoid race conditions.

```
double sum(int n, int nb_threads) {
    double total = 0;
    omp_set_num_threads(nb_threads);
    #pragma omp parallel for reduction(+:total) schedule(static)
    for (int i=1; i<n; i++) {
        total += fn(i);
    }
    return total;
}
```

1.2.3 MPI Parallelization

We used a cyclic distribution (stride = size) to distribute the iterations among processes. Each process computes a partial sum, which is then aggregated using `MPI_Reduce`.

```
double sum(int n, int rank, int size) {
    double local_sum = 0.0;
    for (int i = rank + 1; i < n; i += size) {
        local_sum += fn(i);
    }
    return local_sum;
}
// In main:
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

1.3 Experimental Results

We executed the program with $N = 1,000,000,000$ to ensure significant execution time and convergence towards 1.

Method	**Procs/Threads**	**Time (s)**	**Speedup**
Sequential	1	3.470	Baseline
OpenMP	1	3.471	1.00x
OpenMP	2	1.777	1.95x
OpenMP	4	0.897	3.87x
OpenMP	6	0.638	5.44x
OpenMP	8	0.568	6.11x
MPI	1	3.489	0.99x
MPI	2	1.766	1.96x
MPI	4	0.898	3.86x
MPI	6	0.615	5.64x
MPI	8	0.543	6.39x

Table 1: Performance results for N=1,000,000,000

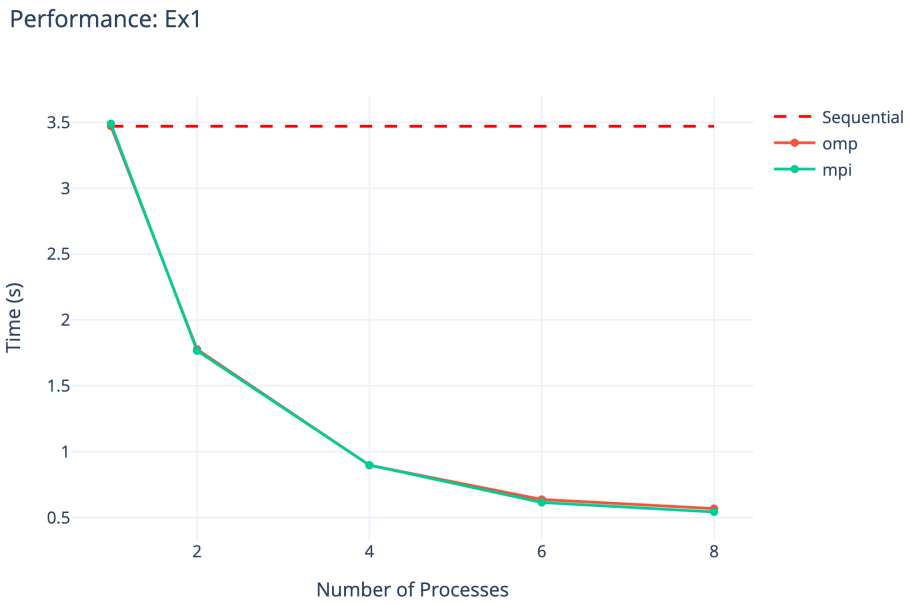

Figure 1: Execution time analysis

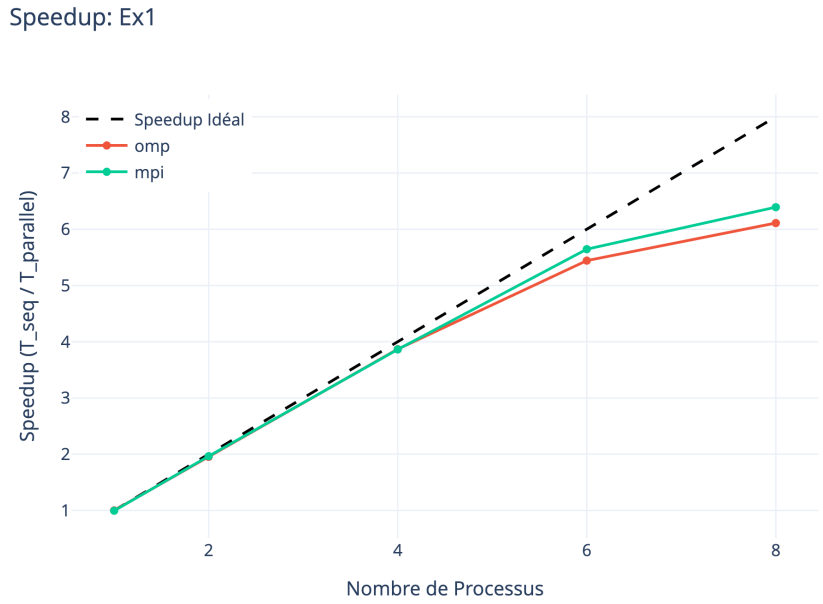

Figure 2: Speedup analysis

The experimental results show nearly theoretical linear speedup for both OpenMP and MPI up to 6 processes. Since the computation is embarrassingly parallel with minimal communication (only one reduction at the end), the overhead is negligible. Interestingly, MPI matches or slightly outperforms OpenMP at 8 processes.

However, a noticeable deviation from the ideal speedup curve is observed when scaling from 6 to 8 processes. This behavior is likely attributed to the specific hardware architecture used for these benchmarks. The tests were executed on a machine equipped with an Apple Silicon M1 processor featuring a heterogeneous multicore architecture.
This specific processor configuration consists of 8 physical cores in total: 6 "Performance" cores and 2 "Efficiency" cores. Up to 6 processes, the operating system is able to schedule the workload exclusively on the high-performance cores, resulting in linear scaling. When the number of processes increases to 8, the system is forced to utilize the two efficiency cores. As these cores are designed for power saving rather than maximum computational throughput (lower clock speeds and different micro-architecture), they introduce a bottleneck relative to the performance cores. Consequently, the average execution speed per core decreases, preventing the total speedup from following the linear trend beyond 6 processes.

2 Exercise 2

2.1 Matrix-vector product

2.1.1 Introduction

The objective of this exercise 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

Table 2: Performance results (N=10,000)

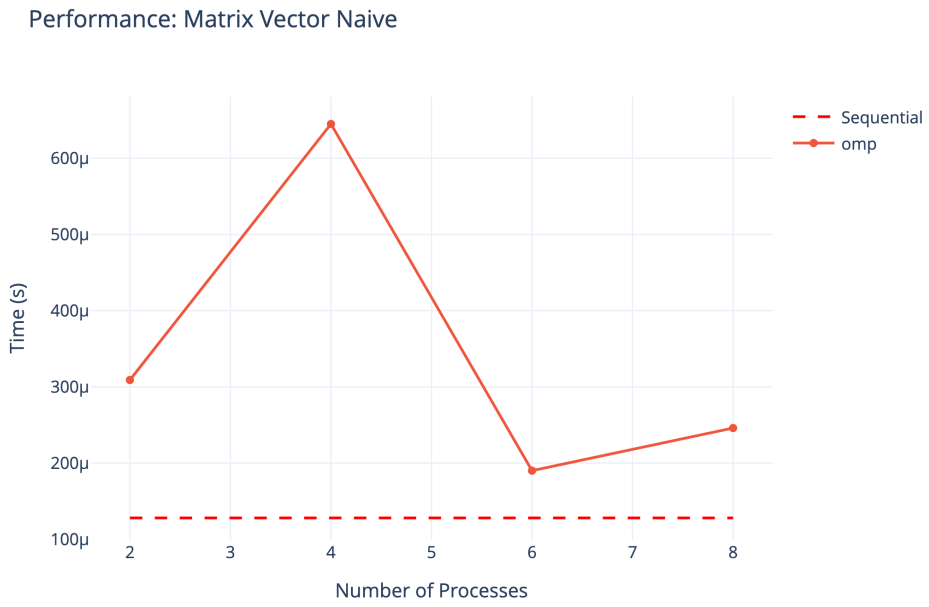

Figure 3: Execution time vs number of threads / procs

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:

$$y[i] = L[i-1] \times x[i-1] + D[i] \times x[i] + U[i] \times x[i+1] \tag{3}$$

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

Table 3: Performance results optimized (N=100,000,000)

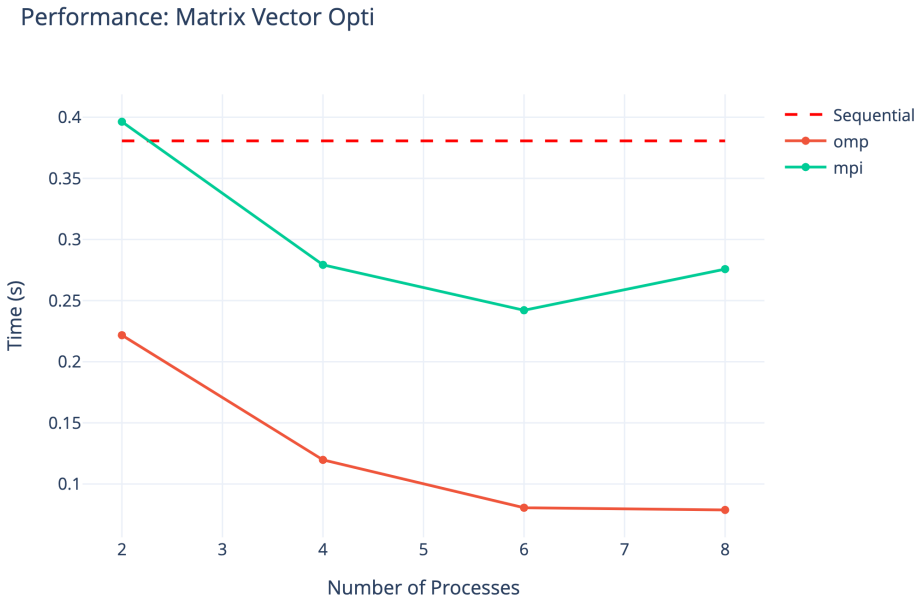

Figure 4: Times in function of the number of threads / procs

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 $N = 10^8$) 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

Table 4: Performance constants for A^2

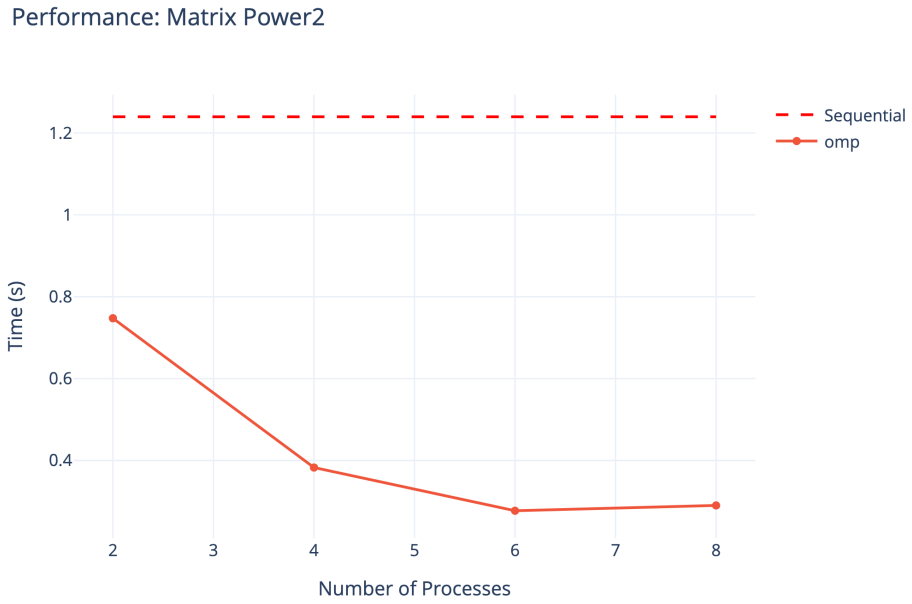

Figure 5: Execution time for A^2 vs number of threads

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

Table 5: Performance constants for A^3

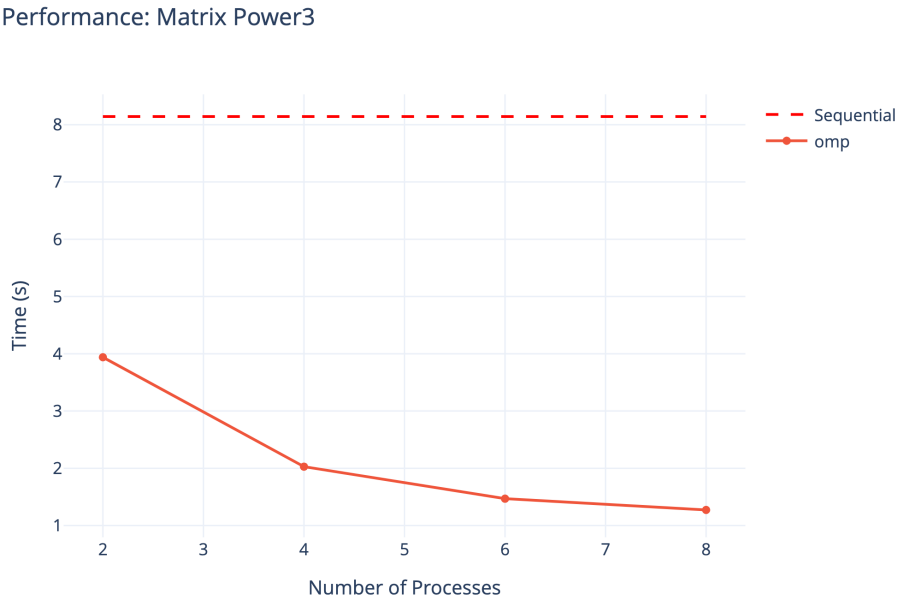

Figure 6: Execution time for A^3 vs number of threads

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.

2.3 Analysis of mpi_mat_vect_mult.c

Question: What does the attached program `mpi_mat_vect_mult.c` do? Adapt the program to do the proposed product A by a random vector.
Analysis: The provided program implements a parallel dense matrix-vector multiplication ($y = Ax$) using MPI.

- **Data Distribution:** The matrix A is distributed by block rows (each process owns N/P rows). The vectors x and y are distributed by blocks (each process owns N/P elements).
- **Algorithm:**
  1. Rank 0 reads the dimensions, matrix, and vector from standard input.
  2. `MPI_Scatter` distributes the matrix rows and vector elements to all processes.

3. `MPI_Allgather` is used to gather the entire vector x on every process. This is necessary because, in a dense matrix-vector product, computing a single element y_i requires the dot product of row i with the entire vector x.
4. Each process computes its local portion of y.
5. `MPI_Gather` collects the results back to Rank 0 for printing.

Adaptation: We adapted the program to perform the multiplication of the specific tridiagonal matrix A (from the project) by a random vector.

- Instead of reading from standard input, we implemented `Generate_matrix` and `Generate_vector` functions to generate the data directly in memory.
- We maintained the dense storage format ($O(N^2)$) as per the original code structure, although this is inefficient for our sparse matrix.
- We added `MPI_Wtime()` calls to measure the execution time.

Performance: For $N = 10,000$ on 4 processes, the execution time was approximately **0.105 seconds**. This confirms that the dense approach involves significant overhead (communication of the full vector and $O(N^2)$ operations) compared to the optimized sparse implementation (< 0.001 s).

3 Exercise 3

3.1 Objective

The goal of this exercise is to:

1. Define a C structure to represent a 3D point.
2. Implement a function to generate a collection of random 3D points.
3. Write an MPI program where Process 2 generates these points and sends them to Process 0.

3.2 Implementation

3.2.1 Data Structure and Generation

We defined a `Point3D` structure containing three doubles (x, y, z). The generation function allocates memory for N points and initializes them with random values between 0 and 100.

```
typedef struct {
    double x;
    double y;
    double z;
} Point3D;

Point3D* generate_points(int n) {
    Point3D* points = (Point3D*) malloc(n * sizeof(Point3D));
    for (int i = 0; i < n; i++) {
        points[i].x = ((double)rand() / RAND_MAX) * 100.0;
        points[i].y = ((double)rand() / RAND_MAX) * 100.0;
        points[i].z = ((double)rand() / RAND_MAX) * 100.0;
    }
    return points;
}
```

3.2.2 MPI Communication

To send an array of structures efficiently, we defined a derived MPI datatype using `MPI_Type_contiguous`. This allows us to treat a `Point3D` struct as a single MPI element consisting of 3 doubles.

```
// Create MPI Datatype for Point3D
MPI_Datatype mpi_point_type;
MPI_Type_contiguous(3, MPI_DOUBLE, &mpi_point_type);
MPI_Type_commit(&mpi_point_type);
```

The communication logic is straightforward:

- **Rank 2:** Generates the points and sends them using `MPI_Send`.
- **Rank 0:** Allocates memory and receives the points using `MPI_Recv`.

```
if (rank == 2) {
    Point3D* points = generate_points(n_points);
    MPI_Send(points, n_points, mpi_point_type, 0, 0, MPI_COMM_WORLD);
} else if (rank == 0) {
    Point3D* received_points = (Point3D*) malloc(n_points * sizeof(Point3D));
    MPI_Recv(received_points, n_points, mpi_point_type, 2, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
}
```

3.3 Execution Results

Running the program with 3 processes confirms the successful transmission of data.

```
Rank 2: Generating 10 3D points...
Rank 2: Sending points to Rank 0...
Rank 0: Waiting for points from Rank 2...
Rank 0: Received points:
Point 0: (44.79, 35.14, 27.44)
Point 1: (90.25, 48.34, 13.32)
...
```

This demonstrates how to handle custom data structures in MPI using derived datatypes, ensuring portable and efficient communication.

4 Summary of the Article

The article presents a parallel numerical method for solving the two-dimensional heat equation, a fundamental partial differential equation describing how heat diffuses over time in a physical domain. Because analytical solutions are rarely available, the author focuses on numerical approximations using a combination of finite difference methods, spectral Fourier methods, and high-performance parallel computing tools.

The study compares several parallelization techniques, mainly MPI (Message Passing Interface) for distributed CPU computation and CUDA for GPU computation. It also integrates the FFTW library, which efficiently computes the Fast Fourier Transform (FFT), a crucial step in spectral approaches for PDEs.

The article begins by explaining how FFTW (and its GPU version CUFFT) accelerates the computation of Fourier transforms, which convert the heat equation into a set of independent ordinary differential equations in spectral space. The author uses two time-integration schemes: the Forward Euler method, which is simple but less accurate, and the fourth-order Runge–Kutta method (RK4), which offers higher accuracy and stability. Figures in the article (Figure 2) show that RK4 reduces numerical error dramatically compared to Euler's method.

A substantial section of the paper describes GPU architecture (Figure 3) and the CUDA programming model based on grids, blocks, and threads (Figure 4). CUDA enables massive parallelism by running thousands of threads simultaneously, which suits the repetitive, local computations typical of numerical PDE solvers. Algorithm 1 presents how the heat equation is solved using finite differences on CUDA, while Algorithm 2 shows the spectral RK4 method implemented with CUFFT.

The article then demonstrates how sparse matrix storage formats, especially CSR (Compressed Sparse Row), drastically improve performance during matrix–vector multiplications required by RK4. Figure 6 illustrates that runtime decreases as the number of threads increases when using CSR.

Multiple numerical tests compare CUDA and MPI. Table 1 shows that GPU computation of the FFT is up to 6× faster than CPU execution for large vectors. Additional figures (7 and 8) show the speed-up and accuracy of the FFTW algorithm on GPU versus MPI. Later tests (Table 2) reveal that GPU execution of the 2D heat equation is up to 100× faster than MPI for large meshes.

Finally, the article validates the physical behavior of solutions: Figure 9 shows the evolution of a random initial temperature field into a smoother distribution due to diffusion, and Figure 10 shows that parallel performance (speed-up) scales well up to 64 processors in an MPI setting.

The conclusion emphasizes that CUDA significantly outperforms MPI, especially for high-resolution grids, due to faster memory access and better parallelization capability. The author plans to extend this work to Navier–Stokes equations, using a similar GPU-accelerated spectral approach.

4.1 Our understanding of this article

From this article, we understood that the main objective is to accelerate the numerical solution of the 2D heat equation by using modern parallel computing tools. Solving this equation on large grids requires performing millions of repetitive calculations, which is slow on a traditional CPU. This is why the author compares two parallel approaches: MPI on CPUs and CUDA on GPUs.
We understood that GPUs are much more efficient because they can run thousands of threads simultaneously and have a much higher memory bandwidth. The article shows that GPU computations, especially when using CUFFT for fast Fourier transforms, are up to 100 times faster than MPI implementations in some tests.
We also understood that the author uses spectral methods based on FFTs to transform the heat equation into simpler equations, and that the fourth-order Runge–Kutta method (RK4) provides high accuracy for time integration.
Overall, we learned that choosing the right computing architecture is essential when solving large scientific problems. The study clearly demonstrates that CUDA GPUs offer superior performance, making them highly suitable for numerical simulations like the heat equation.