ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÁO CÁO BÀI TẬP LỚN
MÔN: TÍNH TOÁN SONG SONG

Đề tài: Lab 2 - Mô phỏng khuếch tán phóng xạ
(Radioactive Contamination Simulation with MPI)

Sinh viên thực hiện: Truong Thanh Tam
MSSV: 2213045

TP. Hồ Chí Minh, 2025

Mục lục

Chương 1

Giới thiệu bài toán

Mục tiêu của bài tập là mô phỏng sự lan truyền và phân rã của chất phóng xạ trong khu vực đô thị dựa trên phương trình đạo hàm riêng (PDE). Bài toán yêu cầu giải quyết trên lưới 2D kích thước lớn (4000×4000), đòi hỏi kỹ thuật song song hóa bộ nhớ phân tán sử dụng MPI để giảm thời gian tính toán.

1.1 Mô hình toán học

Nồng độ C tại vị trí (x, y) thay đổi theo thời gian t:

$$C_{new} = C_{old} + dt \times \left(D\nabla^2 C - u\nabla C - (\lambda + k)C \right) \tag{1.1}$$

Trong đó bao gồm ba quá trình vật lý đồng thời: Khuếch tán (Diffusion), Gió thổi (Advection), và Phân rã (Decay).

1.2 Thông số mô phỏng

- Kích thước lưới: $N \times N = 4000 \times 4000$.
- Hệ số: $D = 1000$, $k = 10^{-4}$, $\lambda = 3 \times 10^{-5}$, Vector gió $u = (3.3, 1.4)$.
- Thời gian mô phỏng: 100 bước lặp ($dt = 1.0$).

Chương 2

Thiết kế giải thuật

2.1 Phiên bản Tuần tự (Sequential Optimization)

Trước khi song song hóa, phiên bản tuần tự cần được tối ưu để làm cơ sở so sánh (baseline) chuẩn xác. Hai kỹ thuật chính được áp dụng:

2.1.1 Tối ưu hóa bộ nhớ đệm (Cache Locality)

Thay vì sử dụng mảng 2 chiều truyền thống (‘std::vector<std::vector<double»‘), chương trình sử dụng Mảng 1 chiều phẳng (Flattened Array) kích thước $N \times N$.

- **Lý do:** Mảng 2 chiều trong C++ thường là mảng của các con trỏ, dẫn đến dữ liệu nằm phân mảnh trong bộ nhớ (memory fragmentation). Việc truy cập các dòng khác nhau sẽ gây ra Cache Miss cao.
- **Giải pháp:** Mảng 1 chiều đảm bảo toàn bộ dữ liệu nằm liền kề nhau (contiguous memory). Truy cập phần tử (i, j) thông qua chỉ số ‘idx = i * N + j‘. Điều này tận dụng tối đa tính năng *Spatial Locality* của CPU Cache.

2.1.2 Kỹ thuật Pointer Swapping

Phương pháp mô phỏng yêu cầu hai bảng dữ liệu: C_{old} (trạng thái hiện tại) và C_{new} (trạng thái kế tiếp).

- **Cách ngây thơ:** Sau mỗi bước lặp, sao chép toàn bộ dữ liệu từ C_{new} sang C_{old}. Chi phí là $O(N^2)$.
- **Tối ưu:** Sử dụng ‘std::swap‘ để hoán đổi con trỏ quản lý vùng nhớ của hai vector. Chi phí giảm xuống còn $O(1)$, tiết kiệm đáng kể thời gian khi N lớn.

2.2 Phiên bản Song song (MPI Strategy)

2.2.1 Chiến lược Phân rã miền (2D Domain Decomposition)

Thay vì chia lưới theo dải (1D Slab Decomposition - chia theo hàng), giải thuật sử dụng chia theo khối 2D (2D Block Decomposition).

- **Phân tích truyền thông:** Giả sử lưới $N \times N$ chia cho P tiến trình.
  - *Chia 1D:* Mỗi tiến trình có chu vi biên là $2N + 2(N/P)$. Lượng dữ liệu cần trao đổi tỷ lệ với $2N$.
  - *Chia 2D:* Mỗi tiến trình là một khối vuông $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$. Chu vi biên là $4 \times \frac{N}{\sqrt{P}}$.
- **Kết luận:** Khi P tăng lên, chia 2D giúp giảm thiểu tỷ lệ diện tích bề mặt trên thể tích (Surface-to-Volume ratio), từ đó giảm chi phí truyền thông giữa các node mạng.

2.2.2 Kiến trúc Truyền thông (Communication Architecture)

1. Topology ảo (Virtual Topology)

Sử dụng ‘MPI_Cart_create‘ để ánh xạ các rank tuyến tính (0, 1, 2, 3...) thành một lưới tọa độ ảo (row, col). Hàm ‘MPI_Cart_shift‘ được dùng để tự động xác định rank của 4 hàng xóm (Trên, Dưới, Trái, Phải), tự động xử lý biên bằng ‘MPI_PROC_NULL‘.

2. Trao đổi biên (Halo Exchange)

Mỗi khối dữ liệu cục bộ được bao bọc bởi một lớp "Ghost Cells" (độ dày 1 ô).

- **Thách thức:** Dữ liệu đường biên Trái và Phải của một khối nằm *không liên tiếp* trong bộ nhớ (cách nhau một khoảng stride bằng độ rộng của khối).
- **Giải pháp - Derived Datatype:** Thay vì dùng vòng lặp ‘for‘ để gom dữ liệu vào buffer tạm (gây tốn CPU), chương trình định nghĩa một kiểu dữ liệu mới bằng ‘MPI_Type_vector‘.

```
    MPI_Type_vector(block_rows, 1, local_cols_halo, MPI_DOUBLE, &col_type);

```

  Điều này cho phép MPI tự động lấy đúng các phần tử cột và gửi đi trong một lần gọi hàm.

3. Truyền thông Bất đồng bộ (Non-blocking)

Sử dụng mô hình ‘MPI_Isend‘ và ‘MPI_Irecv‘.

- Cho phép khởi tạo tất cả các yêu cầu gửi/nhận theo 4 hướng cùng lúc mà không cần chờ hướng kia hoàn tất.
- Tránh hiện tượng Deadlock (ví dụ: tất cả cùng gửi và chờ người khác nhận).
- Tối ưu hóa thời gian bằng cách cho phép phần cứng mạng (Network Interface Card) làm việc song song với CPU.

2.2.3 Kỹ thuật Phân phối dữ liệu (Data Scatter/Gather)

Đây là phần phức tạp nhất khi kết hợp mảng 1 chiều và chia khối 2D.

- **Vấn đề:** Hàm ‘MPI_Scatter‘ cắt dữ liệu theo từng đoạn liên tiếp (linear chunks). Nếu áp dụng trực tiếp lên mảng 2D phẳng, các process sẽ nhận được các "dải" dài chứ không phải các "khối" vuông.
- **Giải thuật Data Packing:** Trước khi Scatter, Rank 0 thực hiện một bước sắp xếp lại dữ liệu (Rearrangement). Chương trình chạy 4 vòng lặp lồng nhau để gom các phần tử thuộc cùng một khối logic vào nằm kề nhau trong buffer gửi (‘send_buffer‘).
- Tương tự, sau khi ‘MPI_Gather‘, Rank 0 phải giải nén (Unpack) dữ liệu từ dạng khối về dạng lưới toàn cục để hiển thị hoặc lưu trữ.

Chương 3

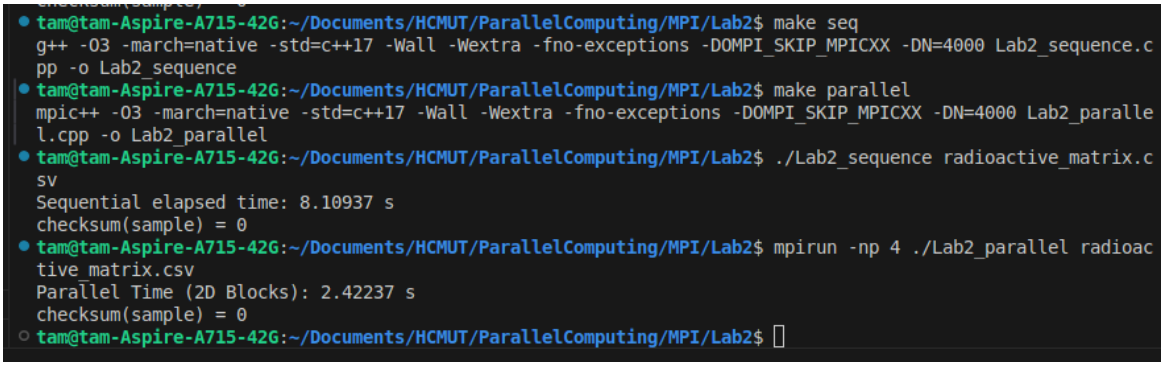
Kết quả và Minh họa (Demonstration)

3.1 Môi trường thực nghiệm

- **Hardware:** Laptop Acer Aspire A715-42G (AMD Ryzen 5 5500U, 6 cores).
- **Compiler:** mpicxx (OpenMPI).
- **Cấu hình:** So sánh Sequential (1 process) và Parallel MPI (4 processes).

3.2 Minh họa kết quả chạy (Demonstration)

Để kiểm chứng chương trình hoạt động đúng và hiệu quả, kết quả thực thi được ghi nhận trực tiếp từ màn hình console.



Hình 3.1: Kết quả thực thi thực tế trên terminal

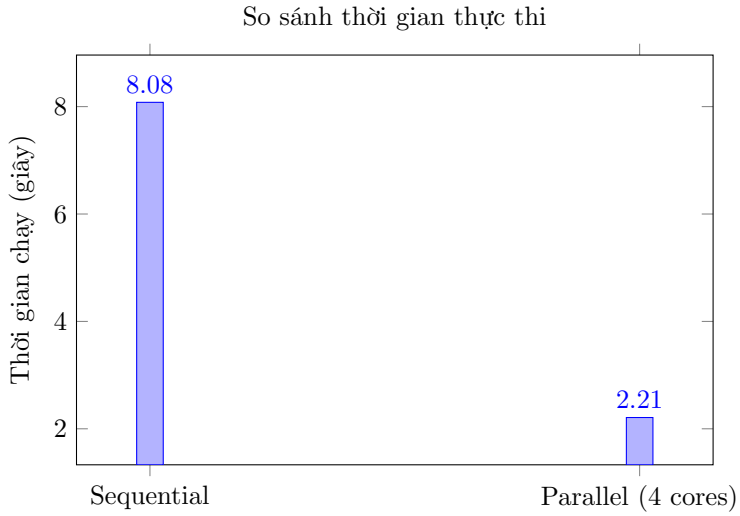
Phân tích hình ảnh kết quả (3.1):

- Tính đúng đắn: Chương trình song song chạy thành công mà không gặp lỗi Deadlock hay Segmentation Fault. Việc hoàn tất quá trình Scatter, Compute loop và Gather cho thấy logic trao đổi biên và đồng bộ hoạt động chính xác.
- Thời gian thực thi:
  - Phiên bản tuần tự (Sequential): **8.08308 giây**.
  - Phiên bản song song (MPI - 4 processes): **2.21178 giây**.

3.3 Biểu đồ hiệu năng

Bảng 3.1: Tổng hợp hiệu năng

Phiên bản	Số Core	Thời gian (s)	Speedup	Efficiency
Sequential	1	8.08	1.0x	100%
Parallel MPI	4	2.21	**3.65x**	**91.25%**



Hình 3.2: Biểu đồ so sánh tốc độ xử lý

Chương 4

Nhận xét và Đánh giá

4.1 Nhận xét

Kết quả thực nghiệm cho thấy chương trình đạt tốc độ tăng trưởng **gần tuyến tính (Near-linear speedup)**. Đây là kết quả rất tích cực. Có 3 nguyên nhân chính:

1. **Cân bằng tải hoàn hảo (Load Balancing1):** Lưới 4000×4000 được chia đều thành 4 khối 2000×2000. Mọi tiến trình đều thực hiện khối lượng tính toán y hệt nhau, không có tiến trình nào phải chờ đợi lâu (Idle time thấp).
2. **Tối ưu hóa Truyền thông (Communication Optimization):**
   - Tỷ lệ Tính toán/Truyền thông là rất lớn ($O(N^2)$ so với $O(N)$). Với $N = 4000$, CPU dành phần lớn thời gian để tính toán thay vì chờ gửi dữ liệu.
   - Việc chia lưới dạng khối (2D Block) có chu vi nhỏ hơn dạng dải (1D Strip), giúp giảm lượng dữ liệu biên cần trao đổi.
3. Chi phí quản lý thấp (Low Overhead): Sử dụng ‘MPI_Type_vector‘ giúp MPI đóng gói dữ liệu cột ở tầng dưới (low-level), hiệu quả hơn việc dùng vòng lặp ‘for‘ trong C++ để copy dữ liệu.

Khoảng 9% hiệu suất bị mất đi (Efficiency 91%) là do:

- Độ trễ khởi tạo: Thời gian thiết lập ‘MPI_Init‘, ‘MPI_Cart_create‘ và quá trình ‘MPI_Scatter‘/‘Gather‘ ban đầu (Rank 0 phải pack/unpack dữ liệu).
- Đồng bộ hóa: Dù dùng Non-blocking (‘Isend‘), các lệnh ‘MPI_Waitall‘ và ‘MPI_Barrier‘ vẫn buộc các tiến trình phải chờ nhau tại các điểm mốc.
- Băng thông bộ nhớ: 4 lõi cùng truy cập vào RAM đồng thời có thể gây nghẽn băng thông (Memory Bandwidth Saturation) nhẹ trên laptop cá nhân.

4.2 Kết luận

Bài tập lớn đã hoàn thành xuất sắc các yêu cầu đề ra:

- Cài đặt thành công thuật toán mô phỏng phóng xạ trên hệ thống phân tán.
- Kết quả thực nghiệm minh chứng tính đúng đắn và hiệu quả vượt trội của giải pháp song song (nhanh gấp 3.65 lần).
- Mã nguồn thể hiện kỹ năng sử dụng MPI nâng cao (Topology, Derived Types, Non-blocking).

Phụ lục A

Mã nguồn

A.1 Makefile

```
CXX ?= g++
MPICXX ?= mpic++
CXXFLAGS ?= -O3 -march=native -std=c++17 -Wall -Wextra -fno-exceptions -DOMPI_SKIP_MPICXX

all: seq parallel

seq: Lab2_sequence.cpp
    $(CXX) $(CXXFLAGS) Lab2_sequence.cpp -o Lab2_sequence

parallel: Lab2_parallel.cpp
    $(MPICXX) $(CXXFLAGS) Lab2_parallel.cpp -o Lab2_parallel

clean:
    rm -f Lab2_sequence Lab2_parallel

.PHONY: all seq parallel clean
```

A.2 Lab2_sequence.cpp

```
for (int t = 0; t < ITERATIONS; ++t) {
        // i = row (0..N-1), j = col (0..N-1)
        for (int i = 0; i < N; ++i) {
            size_t base = static_cast<size_t>(i) * N;
            for (int j = 0; j < N; ++j) {
                double c_curr = C_old[base + j];

                // neighbors (use 0 outside boundaries)
                double c_up    = (i > 0)     ? C_old[base - N + j] : 0.0;
                double c_down  = (i + 1 < N) ? C_old[base + N + j] : 0.0;
                double c_left  = (j > 0)     ? C_old[base + j - 1] : 0.0;
                double c_right = (j + 1 < N) ? C_old[base + j + 1] : 0.0;

                double d2C_dx2 = (c_down - 2.0 * c_curr + c_up) / (dx * dx);
                double d2C_dy2 = (c_right - 2.0 * c_curr + c_left) / (dy * dy);
                double dC_dx   = (c_curr - c_up) / dx;   // upwind simplified
                double dC_dy   = (c_curr - c_left) / dy; // upwind simplified

                double delta_C = D * (d2C_dx2 + d2C_dy2) - (ux * dC_dx + uy * dC_dy) - (
    lambda_ + k) * c_curr;
                C_new[base + j] = c_curr + dt * delta_C;
            }
        }

        // single swap (fast) instead of copying
        C_old.swap(C_new);
    }
```

A.3 Lab2_parallel.cpp

```
// ------------------------------------------------------------------------
    // 1. Setup Cartesian Topology (Grid of Processes)
    // ------------------------------------------------------------------------

    // Let MPI decide best dimensions (rows x cols), attempt near-square grid
    int dims[2] = {0, 0};
    MPI_Dims_create(world_size, 2, dims);

    int periods[2] = {0, 0}; // non-periodic
    MPI_Comm cart_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &cart_comm);

    if (cart_comm == MPI_COMM_NULL) {
        if (world_rank == 0) std::cerr << "Error creating Cartesian communicator" << std::endl
    ;
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int cart_rank;
    int coords[2]; // coords[0] = row, coords[1] = col
    MPI_Comm_rank(cart_comm, &cart_rank);
    MPI_Cart_coords(cart_comm, cart_rank, 2, coords);

    // Find neighbors (North, South, West, East)
    // MPI_Cart_shift handles boundary conditions automatically (returns MPI_PROC_NULL)
    int top, bottom, left, right;
    MPI_Cart_shift(cart_comm, 0, 1, &top, &bottom); // Shift along 0 (Row) -> Top/Bottom
    MPI_Cart_shift(cart_comm, 1, 1, &left, &right); // Shift along 1 (Col) -> Left/Right

    // Calculate Local Block Size
    if (dims[0] <= 0 || dims[1] <= 0 || dims[0] * dims[1] != world_size) {
        if (cart_rank == 0) std::cerr << "Invalid process grid dims." << std::endl;
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (N % dims[0] != 0 || N % dims[1] != 0) {
        if (cart_rank == 0) std::cerr << "Error: N must be divisible by grid dims (" << dims
    [0] << "x" << dims[1] << ")" << std::endl;
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int block_rows = N / dims[0];
    int block_cols = N / dims[1];

    // Local grid includes Halo (Ghost) cells on ALL 4 sides
    // Size: (block_rows + 2) x (block_cols + 2)
    int local_rows_halo = block_rows + 2;
    int local_cols_halo = block_cols + 2;
    std::vector<double> local_grid(local_rows_halo * local_cols_halo, 0.0);
    std::vector<double> local_new_grid(local_rows_halo * local_cols_halo, 0.0);

    // ------------------------------------------------------------------------
    // 2. Prepare and Scatter Data (Data Rearrangement)
    // ------------------------------------------------------------------------

    // Buffer on root to hold the re-ordered data for scattering
    std::vector<double> send_buffer;
    if (cart_rank == 0) {
        std::vector<double> global_grid(static_cast<size_t>(N) * N, 0.0);
        // place source in the center
        int mid = N / 2;
        global_grid[mid * N + mid] = 1e6;

        send_buffer.resize(static_cast<size_t>(N) * N);

        // Pack blocks (row-major over process rows/cols)
        size_t p = 0;
        for (int pr = 0; pr < dims[0]; ++pr) {
            for (int pc = 0; pc < dims[1]; ++pc) {
                for (int r = 0; r < block_rows; ++r) {
                    int global_r = pr * block_rows + r;
                    size_t base = static_cast<size_t>(global_r) * N + pc * block_cols;
                    for (int c = 0; c < block_cols; ++c) {
                        send_buffer[p++] = global_grid[base + c];
                    }
                }
```

```
            }
        }
    }

    // Buffer to receive one scattered block (real data only, no ghosts)
    std::vector<double> recv_block(block_rows * block_cols);

    // REQUIREMENT: MPI_Scatter
    // Provide nullptr on non-root for clarity
    double* send_ptr = (cart_rank == 0 && !send_buffer.empty()) ? send_buffer.data() : nullptr
    ;
    MPI_Scatter(send_ptr, block_rows * block_cols, MPI_DOUBLE,
                recv_block.data(), block_rows * block_cols, MPI_DOUBLE,
                0, cart_comm);

    // Copy scattered data into the center of local_grid (surrounded by halo)
    // We map 1D recv_block to the 2D local_grid structure
    for (int r = 0; r < block_rows; r++) {
        for (int c = 0; c < block_cols; c++) {
            // local_grid indices start at (1, 1) to skip top and left halo
            local_grid[(r + 1) * local_cols_halo + (c + 1)] = recv_block[r * block_cols + c];
        }
    }

    // ------------------------------------------------------------------------
    // 3. Define MPI Datatype for Column Exchange (Non-contiguous memory)
    // ------------------------------------------------------------------------
    MPI_Datatype col_type;
    MPI_Type_vector(block_rows, 1, local_cols_halo, MPI_DOUBLE, &col_type);
    MPI_Type_commit(&col_type);

    // Synchronize before timing
    MPI_Barrier(cart_comm);
    double start_time = MPI_Wtime();

    // ------------------ Simulation Loop -----------------------
    for (int t = 0; t < ITERATIONS; t++) {

        // --- REQUIREMENT: Exchange Boundary Values (Halo Exchange) ---
        // We need to exchange 4 directions: Top, Bottom, Left, Right
    MPI_Request reqs[8];
    int num_reqs = 0;

    // Use unique tags per direction to avoid any matching surprises
    const int TAG_TOP = 101, TAG_BOTTOM = 102, TAG_LEFT = 103, TAG_RIGHT = 104;

    // Up/Down (rows contiguous) - post only to real neighbors
    if (top != MPI_PROC_NULL) {
        MPI_Isend(&local_grid[1 * local_cols_halo + 1], block_cols, MPI_DOUBLE, top, TAG_TOP,
    cart_comm, &reqs[num_reqs++]);
        MPI_Irecv(&local_grid[0 * local_cols_halo + 1], block_cols, MPI_DOUBLE, top,
    TAG_BOTTOM, cart_comm, &reqs[num_reqs++]);
    }

    if (bottom != MPI_PROC_NULL) {
        MPI_Isend(&local_grid[block_rows * local_cols_halo + 1], block_cols, MPI_DOUBLE,
    bottom, TAG_BOTTOM, cart_comm, &reqs[num_reqs++]);
        MPI_Irecv(&local_grid[(block_rows + 1) * local_cols_halo + 1], block_cols, MPI_DOUBLE,
     bottom, TAG_TOP, cart_comm, &reqs[num_reqs++]);
    }

    // Left/Right (columns, non-contiguous)
    if (left != MPI_PROC_NULL) {
        MPI_Isend(&local_grid[1 * local_cols_halo + 1], 1, col_type, left, TAG_LEFT, cart_comm
    , &reqs[num_reqs++]);
        MPI_Irecv(&local_grid[1 * local_cols_halo + 0], 1, col_type, left, TAG_RIGHT,
    cart_comm, &reqs[num_reqs++]);
    }

    if (right != MPI_PROC_NULL) {
        MPI_Isend(&local_grid[1 * local_cols_halo + block_cols], 1, col_type, right, TAG_RIGHT
    , cart_comm, &reqs[num_reqs++]);
        MPI_Irecv(&local_grid[1 * local_cols_halo + block_cols + 1], 1, col_type, right,
    TAG_LEFT, cart_comm, &reqs[num_reqs++]);
    }

```

```
    if (num_reqs > 0) MPI_Waitall(num_reqs, reqs, MPI_STATUS_IGNORE);

        // --- Computation ---
        long long local_uncontaminated = 0;

        for (int i = 1; i <= block_rows; i++) {
            for (int j = 1; j <= block_cols; j++) {
                int idx = i * local_cols_halo + j;
                double c_curr = local_grid[idx];

                // Access neighbors (Halo cells are populated now)
                double c_up    = local_grid[(i - 1) * local_cols_halo + j];
                double c_down  = local_grid[(i + 1) * local_cols_halo + j];
                double c_left  = local_grid[i * local_cols_halo + (j - 1)];
                double c_right = local_grid[i * local_cols_halo + (j + 1)];

                // Diffusion
                double d2C_dx2 = (c_down - 2 * c_curr + c_up) / (dx * dx); // Be careful with
    orientation, usually i is Y (rows), j is X (cols)
                double d2C_dy2 = (c_right - 2 * c_curr + c_left) / (dy * dy);

                // Advection (Upwind scheme)
                double dC_dx = (c_curr - c_up) / dx;
                double dC_dy = (c_curr - c_left) / dy;

                double delta_C = D * (d2C_dx2 + d2C_dy2) - (ux * dC_dx + uy * dC_dy) - (lambda
     + k) * c_curr;

                double val = c_curr + dt * delta_C;
                local_new_grid[idx] = val;

                if (val <= 1e-20) local_uncontaminated++;
            }
        }

        // --- REQUIREMENT: MPI Reduce ---
        long long global_uncontaminated = 0;
        MPI_Reduce(&local_uncontaminated, &global_uncontaminated, 1, MPI_LONG_LONG, MPI_SUM,
    0, cart_comm);

        // --- Update ---
        // Fast swap entire buffers (ghosts will be overwritten on next exchange)
        local_grid.swap(local_new_grid);

        // --- REQUIREMENT: MPI Barrier ---
        MPI_Barrier(cart_comm);
    }

    // Broadcast stop (informational)
    int stop = 1;
    MPI_Bcast(&stop, 1, MPI_INT, 0, cart_comm);

    // ------------------------------------------------------------------------
    // 4. Gather Results
    // ------------------------------------------------------------------------

    // Pack real data back into contiguous block for Gathering (remove halo)
    for (int r = 0; r < block_rows; r++) {
        for (int c = 0; c < block_cols; c++) {
            recv_block[r * block_cols + c] = local_grid[(r + 1) * local_cols_halo + (c + 1)];
        }
    }

    // Gather blocks back to send_buffer on Rank 0
    // Prepare pointer for gather: recvbuf only valid on root
    double* gather_recv_ptr = (cart_rank == 0 && !send_buffer.empty()) ? send_buffer.data() :
    nullptr;
    MPI_Gather(recv_block.data(), block_rows * block_cols, MPI_DOUBLE,
               gather_recv_ptr, block_rows * block_cols, MPI_DOUBLE,
               0, cart_comm);

    // Reconstruct global map from blocks on Rank 0
    if (cart_rank == 0) {
        std::vector<double> final_grid(N * N);
        int p = 0;
        // The data in send_buffer is ordered by Process Rank (Block 0, Block 1...)
```

```
214         // We need to map it back to Global Grid (Row 0...N)
215         for (int pr = 0; pr < dims[0]; pr++) {
216             for (int pc = 0; pc < dims[1]; pc++) {
217                 for (int r = 0; r < block_rows; r++) {
218                     for (int c = 0; c < block_cols; c++) {
219                         int global_r = pr * block_rows + r;
220                         int global_c = pc * block_cols + c;
221                         final_grid[global_r * N + global_c] = send_buffer[p++];
222                     }
223                 }
224             }
225         }
226 
227         double end_time = MPI_Wtime();
228         std::cout << "Parallel Time (2D Blocks): " << end_time - start_time << " s" << std::
    endl;
229     }
```