ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

BÁO CÁO LAB 2: OPEN MPI

MÔN HỌC: TÍNH TOÁN SONG SONG

GVCĐ: La Quốc Nhựt Huân

—o0o—

SVTH: Nguyễn Phúc An (2210022)

TP. HỒ CHÍ MINH, THÁNG 10 NĂM 2025

Mục lục

1 Phần cứng hiện thực

Vì LAB1 sử dụng OpenMPI để tăng cường khả năng song song trong phép toán convolution, nên phần cứng phù hợp nhất là các bộ xử lý đa lõi như CPU hoặc GPU. Trong phạm vi LAB2, các mô phỏng và triển khai thực nghiệm sẽ được thực hiện trên máy tính sử dụng CPU với cấu hình như sau:

```
an@LAPTOP-K3OD08GK:/mnt/d/ParCom/LAB/ParCom_Lab/LAB1$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          46 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   20
  On-line CPU(s) list:    0-19
Vendor ID:                GenuineIntel
  Model name:             12th Gen Intel(R) Core(TM) i9-12900H
    CPU family:           6
    Model:                154
    Thread(s) per core:   2
    Core(s) per socket:   10
    Socket(s):            1
    Stepping:             3
    BogoMIPS:             5836.79
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopol
nstop_tsc cpuid tsc_known_freq pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_t
                          imer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep b
seed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves avx_vnni vnmi umip waitpkg gfni vaes vpclmulqdq r
                          dpid movdiri movdir64b fsrm md_clear serialize flush_l1d arch_capabilities
Virtualization features:
  Virtualization:         VT-x
  Hypervisor vendor:      Microsoft
  Virtualization type:    full
Caches (sum of all):
  L1d:                    480 KiB (10 instances)
  L1i:                    320 KiB (10 instances)
  L2:                     12.5 MiB (10 instances)
  L3:                     24 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-19
```

Hình 1: *Cấu hình CPU*

2 Sequential

2.1 Mục tiêu

Mục tiêu của phần này là triển khai phiên bản tuần tự (sequential) của thuật toán mô phỏng sự lan tỏa phóng xạ trên bản đồ 2D, dựa trên phương trình PDE (Advection-Diffusion-Decay). Thuật toán sẽ cập nhật nồng độ phóng xạ tại mỗi ô theo thời gian, dựa trên gió, khuếch tán và sự suy giảm tự nhiên. Kết quả của phiên bản tuần tự sẽ dùng làm cơ sở so sánh với phiên bản song song (parallel) về hiệu năng.

2.2 Mô tả thuật toán

Thuật toán thực hiện mô phỏng tuần tự trên ma trận $n \times n$, trong T bước thời gian, trong bài này T = 100 seconds tức là 100 bước. Mỗi bước thực hiện các thao tác sau trên toàn bộ ma trận, tức là với mỗi giây ta tính PDE toàn bộ ma trận 1 lần:

1. Lấy giá trị nồng độ hiện tại của ô $C[i, j]$.
2. Tính thành phần advection theo gió (upwind scheme):

$$adv_x = u_x \frac{C[i, j] - C[i-1, j]}{dx}, \quad adv_y = u_y \frac{C[i, j] - C[i, j-1]}{dy}$$

   với các điều kiện biên là 0 khi vượt ngoài ranh giới ma trận.

3. Tính thành phần diffusion sử dụng Laplacian rời rạc:

$$\nabla^2 C \approx \frac{C[i+1, j] - 2C[i, j] + C[i-1, j]}{dx^2} + \frac{C[i, j+1] - 2C[i, j] + C[i, j-1]}{dy^2}$$

   và nhân với hệ số khuếch tán D.

4. Tính thành phần decay do phóng xạ và lắng đọng:

$$decay = (\lambda + k) C[i, j]$$

5. Cập nhật nồng độ mới tại ô:

$$C_{\text{new}}[i,j] = C[i,j] + dt\,(-adv_x - adv_y + diffusion - decay)$$

Sau khi tính xong toàn bộ ma trận, giá trị C_{new} được sao chép trở lại C để dùng cho bước thời gian tiếp theo.

2.3 Mã giả thuật toán

```
for iteration = 1 to T:
    for i = 0 to n-1:
        for j = 0 to n-1:

            idx = i * n + j
            current = A[idx] #current cell

            # zero-padding
            if (i > 0):       up    = A[(i-1)*n + j]
            else:             up    = 0.0

            if (i < n-1):     down  = A[(i+1)*n + j]
            else:             down  = 0.0

            if (j > 0):       left  = A[i*n + (j-1)]
            else:             left  = 0.0

            if (j < n-1):     right = A[i*n + (j+1)]
            else:             right = 0.0

            # Advection
            adv_x = ux * (current - up) / dx
            adv_y = uy * (current - left) / dy

            # Diffusion
            lap_x = (down - 2*current + up) / (dx*dx)
            lap_y = (right - 2*current + left) / (dy*dy)
            diffusion = D * (lap_x + lap_y)

            # decay
            decay = (lambda + k) * current

            # PDE update
            Cnew[idx] = current + dt * (diffusion - decay - adv_x - adv_y)

    #Update A after 1s
    for i = 0 to n*n - 1:
        A[i] = Cnew[i]
```

Algorithm 1: Thuật toán PDE tuần tự

2.4 Độ phức tạp tính toán

Độ phức tạp thời gian (Time Complexity): Thuật toán tuần tự cần duyệt toàn bộ ma trận $n \times n$ trong mỗi bước thời gian, và thực hiện các phép tính cố định tại mỗi ô. Với T bước thời gian, số phép toán xấp xỉ:

$$n \cdot n \times T = T \cdot n^2$$

Do đó, độ phức tạp thời gian của thuật toán được ký hiệu là:

$$\boxed{O(T \cdot n^2)}$$

Giải thích:

- Mỗi ô tính PDE bao gồm các phép cộng, trừ, nhân, chia cố định (advection, diffusion, decay).
- Số bước thời gian T càng lớn, tổng số phép toán càng tăng.
- Ví dụ với ma trận 4000×4000 và $T = 100$, số phép toán xấp xỉ:

$$100 \times 4000^2 = 1.6 \times 10^9$$

Độ phức tạp không gian (Space Complexity): Thuật toán cần lưu trữ:

1. Ma trận hiện tại A kích thước $n \times n$.
2. Ma trận mới C_{new} cùng kích thước $n \times n$ để cập nhật.

Tổng số ô lưu trữ:

$$2 \cdot n^2$$

Do đó, độ phức tạp không gian là:

$$\boxed{O(n^2)}$$

Giải thích:

- Không phụ thuộc vào số bước thời gian T.
- Với $n = 4000$, cần lưu trữ $2 \times 4000^2 = 32{,}000{,}000$ ô, mỗi ô là `double` (8 byte) $\rightarrow \approx 256$ MB bộ nhớ.

3 Parallel

3.1 Mục tiêu

Mục tiêu của phần này là triển khai phiên bản song song (parallel) của thuật toán mô phỏng lan tỏa phóng xạ trên ma trận $N \times N$, sử dụng thư viện Open MPI. Phiên bản song song giúp tận dụng nhiều tiến trình để giảm thời gian tính toán, đồng thời vẫn cho kết quả tương đương với phiên bản tuần tự.

3.2 Mô tả thuật toán

Thuật toán song song chia ma trận gốc thành các khối hàng (row blocks), mỗi tiến trình MPI xử lý một khối. Thuật toán tiến hành theo các bước:

1. Scatter: Ma trận gốc được chia và gửi đến các tiến trình bằng MPI_Scatter.
2. Exchange boundary: Mỗi tiến trình gửi hàng đầu và hàng cuối của khối cho tiến trình láng giềng và nhận các hàng biên từ họ bằng MPI_Isend / MPI_Irecv. Việc này đảm bảo các ô biên có dữ liệu từ tiến trình lân cận để tính toán.
3. PDE update:
   - Tính advection (upwind scheme) theo gió u_x, u_y.
   - Tính diffusion sử dụng Laplacian rời rạc (discrete Laplacian) dựa trên hàng và cột lân cận.
   - Tính decay do phóng xạ và lắng đọng.
   - Cập nhật nồng độ mới tại ô:

$$C_{\text{new}}[i,j] = C[i,j] + dt\,(-adv_x - adv_y + D \cdot laplace - (\lambda + k)\,C[i,j])$$

4. Barrier: Đồng bộ các tiến trình trước khi bước sang bước thời gian tiếp theo bằng MPI_Barrier.
5. Reduce: Tính tổng số ô chưa bị nhiễm (clean cells) trong toàn bộ ma trận bằng MPI_Reduce và in ra từ tiến trình 0.
6. Lặp lại các bước trên cho T bước thời gian.
7. Broadcast stop signal: Sau 100 bước, tiến trình 0 phát tín hiệu dừng tới tất cả tiến trình bằng MPI_Bcast.
8. Gather: Thu thập các khối của mỗi tiến trình về tiến trình 0 bằng MPI_Gather để thu được ma trận cuối cùng.

3.3 Mã giả thuật toán

```
# Input per rank:
#   local_A[rows][N]    : block hàng của ma trận
#   local_C[rows][N]    : buffer output
#   rows                : số hàng local (chunk)
#   N                   : số cột toàn cục
#   T, D, k, lambda, ux, uy, dt, dx, dy
#   world_rank, world_size

allocate lowest_row_above[N]
allocate highest_row_below[N]

for t = 0 to T-1:
    up_rank   = (world_rank == 0) ? MPI_PROC_NULL : world_rank - 1
    down_rank = (world_rank == world_size - 1) ? MPI_PROC_NULL : world_rank + 1

    # exchange halo rows
    Irecv(lowest_row_above, N, DOUBLE, up_rank,    tag=0)
    Irecv(highest_row_below, N, DOUBLE, down_rank, tag=1)
    Isend(first_row(local_A),             N, DOUBLE, up_rank,   tag=1)
    Isend(last_row(local_A),              N, DOUBLE, down_rank, tag=0)
    Waitall(4 requests)

    # update PDE on local block
    for i = 0 to rows-1:
        for j = 0 to N-1:
            idx = i * N + j
            current = local_A[idx]

            # --- Advection (upwind) ---
            if i > 0:
                adv_x = ux * (current - local_A[(i-1)*N + j]) / dx
            else:
                adv_x = (world_rank == 0) ? ux * current / dx
                                          : ux * (current - lowest_row_above[j]) / dx

            adv_y = (j > 0) ? uy * (current - local_A[i*N + (j-1)]) / dy
                            : uy * current / dy

            # --- Diffusion ---
            up    = (i > 0)       ? local_A[(i-1)*N + j] : (world_rank == 0 ? 0.0 : lowest_row_abo
            down  = (i < rows-1)  ? local_A[(i+1)*N + j] : (world_rank == world_size-1 ? 0.0 : hig
            left  = (j > 0)       ? local_A[i*N + (j-1)] : 0.0
            right = (j < N-1)     ? local_A[i*N + (j+1)] : 0.0

            laplace = (down - 2*current + up) / (dx*dx) + (right - 2*current + left) / (dy*dy)

            # --- Decay and update ---
            local_C[idx] = current + dt * (D * laplace - (lambda + k) * current - adv_x - adv_y)

    # copy local_C -> local_A for next iteration
    for idx = 0 to rows*N - 1:
        local_A[idx] = local_C[idx]

    # compute local clean cells (ignore padding if last rank)
    actual_rows = (world_rank == world_size-1) ? N - world_rank*rows : rows
    local_clean = 0
    for i = 0 to actual_rows-1:
        for j = 0 to N-1:
```

```
            if local_A[i*N + j] <= 0.0:
                local_clean += 1

    # global sum
    Reduce(local_clean -> global_clean, SUM, root=0)
    Barrier(MPI_COMM_WORLD)

    if world_rank == 0:
        print("Iteration", t, "Clean cells =", global_clean)

# stop signal
stop = 1
Broadcast(stop, root=0)

free lowest_row_above
free highest_row_below
```

3.4 Độ phức tạp tính toán

Độ phức tạp thời gian (Time Complexity): Trong phiên bản song song với P tiến trình MPI, mỗi tiến trình xử lý khoảng $\frac{n^2}{P}$ ô trên mỗi bước thời gian. Với T bước thời gian, số phép toán cục bộ của mỗi tiến trình xấp xỉ:

$$\frac{T \cdot n^2}{P}$$

Tuy nhiên, phiên bản song song còn phải thực hiện các phép giao tiếp MPI:

- Trao đổi hàng biên (boundary exchange) với tiến trình lân cận.
- Đồng bộ (Barrier) và tổng hợp kết quả (Reduce, Broadcast).

Do đó, độ phức tạp thời gian lý thuyết tính toán cục bộ là:

$$\boxed{O\left(\frac{T \cdot n^2}{P}\right)}$$

Nhưng hiệu năng thực tế còn bị chi phối bởi chi phí giao tiếp giữa các tiến trình, đặc biệt khi P lớn và ma trận $N \times N$ lớn (ví dụ 4000×4000).

Độ phức tạp không gian (Space Complexity): Mỗi tiến trình cần lưu trữ:

1. Khối hàng cục bộ *local_A* và buffer *local_C*: tổng khoảng $2 \cdot \frac{n^2}{P}$ ô.
2. Hàng biên nhận từ tiến trình lân cận: $2 \cdot n$ ô.

Do đó, độ phức tạp không gian cục bộ mỗi tiến trình:

$$\boxed{O\left(\frac{n^2}{P} + n\right) \approx O\left(\frac{n^2}{P}\right)}$$

4 Mô phỏng và đánh giá kết quả

Ngoài việc, hiện thực phép tích chập trên nhiều phiên bản, ta cũng cần kiểm tra kết quả liệu các phiên bản có cho ra kết quả giống nhau không. Trong bài này, sinh viên sẽ dùng hàm:

```
void check_matrix_equal(double* A, double* B, unsigned int n){
    for (unsigned int idx = 0; idx < n; ++idx){
        if (A[idx] != B[idx]) {
            unsigned int i = idx / 4000;
            unsigned int j = idx % 4000;
            printf("Difference found at position (%u, %u): A=%.6f, B=%.6f\n",
                   i, j, A[idx], B[idx]);
            printf("Two matrices are not equal.\n");
            return;
        }
    }

    printf("Check matrix equal successfully\n");
}
```

Algorithm 2: Hàm kiểm tra kết quả

Cách sử dụng, mô phỏng toàn chương trình LAB2:

- make build → Biên dịch (build) chương trình.

- make run → Chạy chương trình.

Kết quả:

```
an@LAPTOP-K3OD08GK:/mnt/d/ParCom/LAB/ParCom_Lab/LAB2$ make run
mpirun -np 9 ./ex
Starting reading radioactive_matrix.csv
Reading finished in 55.942655 s
Starting sequential simulation:
Sequential PDE time: 31.709444 seconds
Starting parallel simulation:
Parallel PDE time: 4.474531 seconds
Check matrix equal successfully
Check valid seq and parallel in 0.018061 seconds
```

Hình 2: *Kết quả mô phỏng*

Lưu ý: Kết quả này chạy chưa trong lúc comment dòng code:

```
if world_rank == 0:
    print("Iteration", t, "Clean cells =", global_clean)
```

Nếu uncomment dòng code đó thì có kết quả chạy ra sẽ được thêm cái những thông báo về các ô chưa bị ô nhiễm như này:

```
Iteration 0: Clean cells = 12858447
Iteration 1: Clean cells = 12858439
Iteration 2: Clean cells = 12858427
Iteration 3: Clean cells = 13218020
Iteration 4: Clean cells = 14384384
Iteration 5: Clean cells = 14412167
Iteration 6: Clean cells = 14409538
Iteration 7: Clean cells = 14406242
Iteration 8: Clean cells = 14401772
Iteration 9: Clean cells = 14398805
Iteration 10: Clean cells = 14395452
Iteration 11: Clean cells = 14392362
```

```
Iteration 12: Clean cells = 14390506
Iteration 13: Clean cells = 14386149
Iteration 14: Clean cells = 14384218
Iteration 15: Clean cells = 14381255
Iteration 16: Clean cells = 14378700
Iteration 17: Clean cells = 14375747
Iteration 18: Clean cells = 14372512
Iteration 19: Clean cells = 14370096
Iteration 20: Clean cells = 14366895
Iteration 21: Clean cells = 14363707
Iteration 22: Clean cells = 14361517
Iteration 23: Clean cells = 14357149
Iteration 24: Clean cells = 14355360
Iteration 25: Clean cells = 14351063
Iteration 26: Clean cells = 14350058
Iteration 27: Clean cells = 14345212
Iteration 28: Clean cells = 14344253
Iteration 29: Clean cells = 14339417
Iteration 30: Clean cells = 14338566
Iteration 31: Clean cells = 14333609
Iteration 32: Clean cells = 14332642
Iteration 33: Clean cells = 14328071
Iteration 34: Clean cells = 14326468
Iteration 35: Clean cells = 14322687
Iteration 36: Clean cells = 14320098
Iteration 37: Clean cells = 14317114
Iteration 38: Clean cells = 14313796
Iteration 39: Clean cells = 14311211
Iteration 40: Clean cells = 14307968
Iteration 41: Clean cells = 14305270
Iteration 42: Clean cells = 14301775
Iteration 43: Clean cells = 14299125
Iteration 44: Clean cells = 14295641
Iteration 45: Clean cells = 14293361
Iteration 46: Clean cells = 14289942
Iteration 47: Clean cells = 14287159
Iteration 48: Clean cells = 14284186
Iteration 49: Clean cells = 14281360
Iteration 50: Clean cells = 14278167
Iteration 51: Clean cells = 14275514
Iteration 52: Clean cells = 14272287
Iteration 53: Clean cells = 14269632
Iteration 54: Clean cells = 14266426
Iteration 55: Clean cells = 14263630
Iteration 56: Clean cells = 14260388
Iteration 57: Clean cells = 14258045
Iteration 58: Clean cells = 14254228
Iteration 59: Clean cells = 14252049
Iteration 60: Clean cells = 14248423
Iteration 61: Clean cells = 14245986
Iteration 62: Clean cells = 14242342
Iteration 63: Clean cells = 14240147
Iteration 64: Clean cells = 14236579
Iteration 65: Clean cells = 14234123
Iteration 66: Clean cells = 14230613
Iteration 67: Clean cells = 14228097
Iteration 68: Clean cells = 14224563
Iteration 69: Clean cells = 14222037
Iteration 70: Clean cells = 14218608
Iteration 71: Clean cells = 14215990
```

```
Iteration 72: Clean cells = 14212490
Iteration 73: Clean cells = 14209901
Iteration 74: Clean cells = 14206656
Iteration 75: Clean cells = 14203857
Iteration 76: Clean cells = 14200558
Iteration 71: Clean cells = 14215990
Iteration 72: Clean cells = 14212490
Iteration 73: Clean cells = 14209901
Iteration 74: Clean cells = 14206656
Iteration 75: Clean cells = 14203857
Iteration 76: Clean cells = 14200558
Iteration 77: Clean cells = 14197908
Iteration 78: Clean cells = 14194685
Iteration 72: Clean cells = 14212490
Iteration 73: Clean cells = 14209901
Iteration 74: Clean cells = 14206656
Iteration 75: Clean cells = 14203857
Iteration 76: Clean cells = 14200558
Iteration 77: Clean cells = 14197908
Iteration 78: Clean cells = 14194685
Iteration 79: Clean cells = 14191886
Iteration 80: Clean cells = 14188517
Iteration 75: Clean cells = 14203857
Iteration 76: Clean cells = 14200558
Iteration 77: Clean cells = 14197908
Iteration 78: Clean cells = 14194685
Iteration 79: Clean cells = 14191886
Iteration 80: Clean cells = 14188517
Iteration 76: Clean cells = 14200558
Iteration 77: Clean cells = 14197908
Iteration 78: Clean cells = 14194685
Iteration 79: Clean cells = 14191886
Iteration 80: Clean cells = 14188517
Iteration 81: Clean cells = 14185900
Iteration 82: Clean cells = 14182570
Iteration 77: Clean cells = 14197908
Iteration 78: Clean cells = 14194685
Iteration 79: Clean cells = 14191886
Iteration 80: Clean cells = 14188517
Iteration 81: Clean cells = 14185900
Iteration 82: Clean cells = 14182570
Iteration 79: Clean cells = 14191886
Iteration 80: Clean cells = 14188517
Iteration 81: Clean cells = 14185900
Iteration 82: Clean cells = 14182570
Iteration 83: Clean cells = 14179674
Iteration 84: Clean cells = 14176657
Iteration 85: Clean cells = 14173609
Iteration 86: Clean cells = 14170764
Iteration 87: Clean cells = 14167509
Iteration 81: Clean cells = 14185900
Iteration 82: Clean cells = 14182570
Iteration 83: Clean cells = 14179674
Iteration 84: Clean cells = 14176657
Iteration 85: Clean cells = 14173609
Iteration 86: Clean cells = 14170764
Iteration 87: Clean cells = 14167509
Iteration 88: Clean cells = 14164517
Iteration 89: Clean cells = 14161151
Iteration 90: Clean cells = 14158281
```

```
Iteration 85: Clean cells = 14173609
Iteration 86: Clean cells = 14170764
Iteration 87: Clean cells = 14167509
Iteration 88: Clean cells = 14164517
Iteration 89: Clean cells = 14161151
Iteration 90: Clean cells = 14158281
Iteration 91: Clean cells = 14154911
Iteration 88: Clean cells = 14164517
Iteration 89: Clean cells = 14161151
Iteration 90: Clean cells = 14158281
Iteration 91: Clean cells = 14154911
Iteration 91: Clean cells = 14154911
Iteration 92: Clean cells = 14152268
Iteration 93: Clean cells = 14148696
Iteration 94: Clean cells = 14146175
Iteration 95: Clean cells = 14142696
Iteration 96: Clean cells = 14139992
Iteration 97: Clean cells = 14136738
Iteration 98: Clean cells = 14133727
Iteration 99: Clean cells = 14130798
```

Bài toán PDE tuần tự và song song được thực hiện trên ma trận $N \times N$ với T bước thời gian. Kết quả thực nghiệm trên máy cá nhân với $P = 9$ tiến trình MPI:

- Thời gian tuần tự: $T_{\text{seq}} = 31.71$ s
- Thời gian song song với 9 tiến trình MPI: $T_{\text{par}} = 4.47$ s

Hiệu suất tăng tốc (speedup) đạt được là:

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}} = \frac{31.71}{4.47} \approx 7.1$$

Với $P = 9$ tiến trình, hiệu suất sử dụng tài nguyên (efficiency) là:

$$\text{Efficiency} = \frac{\text{Speedup}}{P} = \frac{7.1}{9} \approx 79\%$$

4.0.1 Phân tích

Để song song hóa bài toán PDE, ma trận $N \times N$ được chia theo hàng cho P tiến trình MPI.

Số hàng mỗi tiến trình nhận được (chunk) được tính bằng:

$$\text{chunk} = \left\lceil \frac{N+P-1}{P} \right\rceil$$

Tổng số hàng được padded (làm tròn lên cho vừa chia đều) là:

$$\text{padded} = \text{chunk} \times P$$

Ví dụ, nếu $N = 4000$ và $P = 9$, ta có:

$$\text{chunk} = \left\lceil \frac{4000+9-1}{9} \right\rceil = 445, \qquad \text{padded} = 445 \times 9 = 4005$$

Mỗi tiến trình r ($0 \le r < P$) nhận một khối (block) gồm chunk hàng liên tiếp:

$$\text{local\_A}_r = A\left[\, r \cdot \text{chunk} : (r+1) \cdot \text{chunk} - 1,\ 0 : N-1 \,\right]$$

Mỗi bước thời gian, các tiến trình tính toán song song với nhau cập nhật PDE tại ô (i, j) ở khối mà chúng được phân công:

$$C_{i,j}^{t+1} = C_{i,j}^{t} + \Delta t \left(-u_x \frac{\partial C}{\partial x} - u_y \frac{\partial C}{\partial y} + D\nabla^2 C - (\lambda + k) C_{i,j}^{t} \right)$$

- Thời gian tính toán mỗi ô: $O(1)$
- Tổng số ô: n^2, tổng số bước thời gian: T
- Độ phức tạp thời gian tuần tự: $T_{\text{seq}} = O(T \cdot n^2)$

Khi song song hóa theo hàng với P tiến trình:

$$T_{\text{par}} \approx \frac{T_{\text{seq}}}{P} + T_{\text{comm}}$$

Trong đó T_{comm} bao gồm:

- Trao đổi hàng rìa giữa các tiến trình
- Barrier và Reduce mỗi vòng lặp

4.0.2 Nhận xét

Mục tiêu lý thuyết: $T_{\text{par}} \approx T_{\text{seq}}/P \approx 31.71/9 \approx 3.52$ s, nhưng thực tế đạt $T_{\text{par}} = 4.47$ s. Nguyên nhân:

- Overhead giao tiếp MPI: trao đổi hàng rìa, barrier, reduce
- Load imbalance nếu N không chia hết cho P
- Memory latency, cache miss và chi phí quản lý tiến trình

Hiệu suất thực tế vẫn đạt 79%, thể hiện thuật toán song song giảm đáng kể thời gian so với tuần tự.