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 1: OPENMP

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 OpenMP 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 LAB1, 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 Tích chập tuần tự (Sequential Convolution)

2.1 Mục tiêu

Mục tiêu của phần này là hiện thực phiên bản tuần tự của phép tích chập 2 chiều nhằm mô phỏng quá trình khuếch tán nhiệt sau một vụ nổ hạt nhân. Bản đồ nhiệt kích thước 4000×4000 được cập nhật lặp lại 100 lần bằng kernel khuếch tán kích thước 3×3. Việc xử lý biên được thực hiện bằng replicate padding với giá trị nhiệt độ nền không đổi là 30°C.

2.2 Mô tả thuật toán

Thuật toán thực hiện phép tích chập hai chiều (2D convolution) trên ma trận kích thước $A_{\text{size}} \times A_{\text{size}}$ với kernel có kích thước $k_{\text{size}} \times k_{\text{size}}$.

Tại mỗi bước lặp, cửa sổ tích chập được đặt tại vị trí (i, j) trên ma trận A, và giá trị mới được tính bằng tổng có trọng số giữa các phần tử lân cận và kernel:

$$B[i,j] = \sum_{u=0}^{k_{\text{size}}-1} \sum_{v=0}^{k_{\text{size}}-1} \text{value}(i+u-\text{pad},\ j+v-\text{pad}) \cdot K[u,v],$$

trong đó

$$\text{pad} = \left\lfloor \frac{k_{\text{size}}}{2} \right\rfloor$$

được sử dụng để đảm bảo kernel được căn giữa tại mỗi phần tử của ma trận. Việc này giúp chỉ số truy cập đúng vị trí lân cận xung quanh ô trung tâm, kể cả khi kernel có kích thước lẻ.

Nếu chỉ số truy cập vượt ra ngoài biên của ma trận A, thuật toán sử dụng giá trị mặc định là 30:

$$\text{value}(x,y) = \begin{cases} A[x,y], & \text{nếu } 0 \le x,y < A_{\text{size}}, \\ 30, & \text{ngược lại.} \end{cases}$$

Mỗi phép tích chập được thực hiện cho toàn bộ ma trận, tạo ra ma trận kết quả B. Sau mỗi vòng lặp, ma trận A được cập nhật bằng giá trị của B để chuẩn bị cho bước lặp tiếp theo:

$$A \leftarrow B.$$

Quá trình trên được lặp lại 100 lần nhằm mô phỏng sự lan truyền nhiệt qua thời gian.

2.3 Mã giả thuật toán

```
pad = ksize / 2

for iteration = 1 to 100:
    for i = 0 to Asize-1:
        for j = 0 to Asize-1:
            sum = 0
            for ki = 0 to ksize-1:
                for kj = 0 to ksize-1:
                    ii = i + ki - pad
                    jj = j + kj - pad

                    if (ii < 0 or jj < 0 or ii >= Asize or jj >= Asize):
                        sum = sum + 30.0 * K[ki][kj]
                    else:
                        sum = sum + A[ii * Asize + jj] * K[ki][kj]

            B[i * Asize + j] = sum

    #update A in each iteration
    for i = 0 to Asize-1:
        for j = 0 to Asize-1:
            A[i * Asize + j] = B[i * Asize + j]
```

Algorithm 1: Tích chập tuần tự

2.4 Biểu diễn dữ liệu

Để tối ưu hiệu năng truy cập bộ nhớ và giảm hiện tượng cache miss, ma trận kích thước 4000×4000 được lưu trữ dưới dạng mảng một chiều thay vì một vector 2 chiều. Với $N = 4000$, phần tử ở hàng i và cột j được ánh xạ vào mảng một chiều theo công thức:

$$A[i][j] = A[i \cdot N + j].$$

Cách lưu trữ này tận dụng tính liên tục trong bộ nhớ, giúp CPU đọc dữ liệu theo cache line, tăng tốc độ truy cập so với việc dùng vector-of-vectors, đặc biệt quan trọng khi xử lý các ma trận lớn.

2.5 Độ phức tạp tính toán

Xét tích chập 2D tuần tự trên ma trận kích thước $N \times N$ với kernel kích thước $k_{\text{size}} \times k_{\text{size}}$.

- Độ phức tạp thời gian (Time complexity). Mỗi ô đầu ra cần thực hiện k_{size}^2 phép nhân và $(k_{\text{size}}^2 - 1)$ phép cộng, do đó số phép toán (theo hằng số nhân/cộng) cho toàn ma trận là

  $$T(N) = O\big(k_{\text{size}}^2 \cdot N^2\big).$$

  Vì k_{size} thường là hằng số (ở đây $k_{\text{size}} = 3$), suy ra

  $$T(N) = O(N^2).$$

Với $k_{\text{size}} = 3$ và lặp I lần (ví dụ $I = 100$), tổng chi phí thời gian là

$$T_{\text{total}}(N, I) = O\big(I \cdot N^2\big) = O(N^2)$$

(vì I là hằng số).

- Độ phức tạp không gian (Space complexity). Lưu trữ ma trận vào (input) và ma trận ra (output) cùng các cấu trúc phụ cố định (kernel, biến tạm) yêu cầu:

$$S(N) = O(N^2).$$

Với $N = 4000$, $k_{\text{size}} = 3$ và $I = 100$, ta có số ô:

$$N^2 = 4000^2 = 16 \times 10^6,$$

và chi phí thời gian tỉ lệ với $I \cdot N^2 = 100 \cdot 16 \times 10^6 = O(1.6 \times 10^9)$ đơn vị công việc (hằng số hệ số phụ tùy cách đếm phép toán).

3 Parallel Convolution 1

3.1 Mục tiêu

Mục tiêu của phần này là xây dựng phiên bản song song của thuật toán tích chập 2D đã thực hiện ở phần tuần tự, sử dụng thư viện OpenMP. Phiên bản này nhằm tăng tốc độ xử lý 100 lần phép tích chập toàn bộ ma trận kích thước 4000×4000 với bằng cách phân chia đều không gian tính toán cho 10 luồng xử lý song song. Phần biểu diễn dữ liệu của phần này sẽ giống phần tuần tự.

3.2 Mô tả thuật toán

Thuật toán thực hiện phép tích chập 2D song song theo các bước sau:

- Khởi tạo một vùng song song duy nhất với 10 luồng (tránh chi phí tạo/destroy vùng nhiều lần).
- Trong mỗi vòng lặp (tổng cộng 100 vòng):
  - Các luồng cùng chia sẻ công việc tính toán từng phần tử của ma trận B.
  - Hai vòng lặp (i, j) được gộp bằng `collapse(2)`, biến thành một vòng lặp phẳng lớn có N^2 lần lặp để tính 1 lần tích chập cả ma trận 4000×4000. Sau đó OpenMP chia vòng lặp này thành các *chunk* bằng `schedule(static, N*N/NUM_THREADS)`.
  - Mỗi phần tử (i, j) thực hiện phép tích chập với kernel 3×3.
  - Nếu vị trí truy cập vượt ngoài biên, giá trị mặc định 30 được sử dụng.
- Sau khi tính xong ma trận B, một vòng `omp for` riêng được dùng để sao chép dữ liệu từ B sang A.

3.3 Mã giả thuật toán

```
pad = ksize / 2

#pragma omp parallel num_threads(10)
for iteration = 1 to 100:
    #pragma omp for collapse(2) schedule(dynamic, N * N / 10)
    for i = 0 to Asize-1:
        for j = 0 to Asize-1:
            sum = 0
            for ki = 0 to ksize-1:
                for kj = 0 to ksize-1:
                    ii = i + ki - pad
                    jj = j + kj - pad
                    if (ii < 0 or jj < 0 or ii >= Asize or jj >= Asize):
                        sum = sum + 30.0 * K[ki][kj]
                    else:
                        sum = sum + A[ii * Asize + jj] * K[ki][kj]

            B[i * Asize + j] = sum

    #pragma omp for schedule(dynamic, N * N / 10)
    for t = 0 to Asize*Asize-1:
        A[t] = B[t]
```

Algorithm 2: Parallel 2D Convolution (OpenMP)

3.4 Độ phức tạp tính toán

Ta vẫn giữ các ký hiệu: ma trận kích thước $N \times N$, kernel kích thước $k_{\text{size}} \times k_{\text{size}}$ (ở đây $k_{\text{size}} = 3$), và số vòng lặp I (ở đây $I = 100$). Với $N = 4000$ ta có

$$N^2 = 4000^2 = 16{,}000{,}000.$$

Độ phức tạp

- Work (tổng công việc): tổng số phép toán cho mỗi vòng vẫn tỉ lệ $O(k_{\text{size}}^2 N^2)$. Vì k_{size} là hằng số, ta có

$$\text{Work} = O(N^2).$$

- Time (với P luồng lý tưởng): nếu không có overhead và phân bố tải hoàn hảo, thời gian sẽ xấp xỉ

$$T_{\text{ideal}}(P) = \frac{T_{\text{seq}}}{P}.$$

- Thực tế (có overhead): khi tính đến chi phí tạo vùng song song và đồng bộ, mô hình đơn giản là

$$T_{\text{par}}(P) \approx T_{\text{overhead}} + \frac{T_{\text{seq}}}{P},$$

trong đó T_{overhead} là chi phí cố định do OpenMP (tạo/khởi động thread, barrier, scheduling, v.v.).

Speedup và Efficiency Định nghĩa:

$$\text{Speedup } S(P) = \frac{T_{\text{seq}}}{T_{\text{par}}(P)}, \qquad \text{Efficiency } E(P) = \frac{S(P)}{P}.$$

4 Parallel Convolution 2

4.1 Mục tiêu

Mục tiêu của phần này là xây dựng một phiên bản song song cải tiến của thuật toán tích chập 2D, sử dụng kỹ thuật *block-based* để tận dụng cache tốt hơn, vẫn dùng thư viện OpenMP với 10 luồng. Phiên bản này nhằm tăng tốc độ xử lý 100 lần phép tích chập toàn bộ ma trận 4000×4000, đồng thời giảm chi phí truy cập bộ nhớ nhờ chia ma trận thành các *block* kích thước 128×128.

4.2 Mô tả thuật toán

Thuật toán thực hiện phép tích chập 2D song song theo các bước sau:

- Khởi tạo một vùng song song duy nhất với 10 luồng.
- Trong mỗi vòng lặp (tổng cộng 100 vòng):
  - Ma trận A được chia thành các block con kích thước 128×128. Mỗi block là một phần ma trận liên tục, giúp tăng khả năng *locality* trong cache.
  - Các luồng cùng chia sẻ công việc tính toán từng block, mỗi luồng xử lý một hoặc nhiều block.
  - Hai vòng lặp (bi, bj) duyệt các block con, và hai vòng lặp (i, j) bên trong block thực hiện phép tích chập. Với kernel 3×3, mỗi ô (i, j) vẫn thực hiện 9 phép nhân và cộng.
  - Nếu vị trí truy cập vượt ngoài biên, giá trị mặc định 30 được sử dụng.
  - OpenMP sử dụng `collapse(2)` để hợp nhất hai vòng lặp duyệt block, và `schedule(dynamic)` để phân phối block linh hoạt cho các luồng. Điều này giúp cân bằng tải tốt hơn so với phân phối tĩnh, nhất là khi workload không đồng đều.
- Sau khi tính xong ma trận B, một vòng `omp for` riêng được dùng để sao chép dữ liệu từ B sang A với `schedule(dynamic)`, đảm bảo các luồng hoạt động song song.

4.3 Giải thích chi tiết và ví dụ minh họa

Chia block (tile) — lý do và công thức

Để tận dụng cơ chế cache reuse và tăng hiệu quả song song, thuật toán tích chập 2D được chia thành các block có kích thước $B \times B$.

Với ma trận kích thước $N \times N$ và block $B \times B$, số block theo mỗi chiều được tính bằng:

$$n_{\text{block}} = \left\lceil \frac{N}{B} \right\rceil.$$

Ví dụ: với $N = 4000$ và $B = 128$:

$$n_{\text{block}} = \left\lceil \frac{4000}{128} \right\rceil = 32.$$

Tổng số block trên toàn ma trận:

$$n_{\text{total}} = n_{\text{block}}^2 = 32^2 = 1024.$$

Mỗi block chứa tối đa:

$$B \times B = 128 \times 128 = 16\,384 \text{ phần tử.}$$

Lý do chọn block size dựa vào cache

Xét cache L1 data/core = 48 KiB, L2/core $\approx$ 1.25 MiB, L3 = 24 MiB (Cấu hình CPU sử dụng). Với dữ liệu double (8 byte), tổng kích thước dữ liệu cần lưu trong L1 để tính một block (bao gồm input tile + output tile) là:

$$\text{Bytes}_{\text{block}} \approx (B+2)^2 \cdot 8 + B^2 \cdot 8$$

trong đó $B+2$ xét halo của kernel 3×3.

Ví dụ với $B = 128$:

$$(130)^2 \cdot 8 + 128^2 \cdot 8 \approx 33{,}800 \text{ B} + 16{,}384 \cdot 8 \approx 50.2 \text{ KiB}.$$

Tại sao block giúp cache locality

- Ma trận lưu theo row-major: phần tử $A[i][j]$ nằm tại chỉ số $A[i \cdot N + j]$.
- Khi xử lý một block $B \times B$, inner loop theo cột j giúp truy xuất các phần tử liên tiếp $\rightarrow$ spatial locality.
- Kernel 3×3 truy xuất phần tử lân cận $(i \pm 1, j \pm 1)$. Nếu block vừa đủ, các phần tử này thường đã nằm trong cache $\rightarrow$ tăng cache hit, giảm truy cập RAM.

Phân phối công việc: `collapse(2)` + `schedule(dynamic)`

- `collapse(2)` gộp hai vòng lặp block bi và bj thành một vòng lặp 1-D với n_{total} lần lặp; mỗi lần lặp tương ứng một block.
- `schedule(dynamic)` cho phép thread lấy block tiếp theo khi hoàn thành block hiện tại. Khác với static:
  - Cân bằng tải: thread không idle nếu block khác nặng hơn.
  - Chi phí điều phối: dynamic có overhead cao hơn static nhưng bù lại giảm idle và tăng locality.

So sánh với chia chunk tĩnh

- Static chunk: OpenMP phân công phần tử 1-D cho mỗi thread. Các phần tử rải rác nhiều hàng $\rightarrow$ giảm cache hit.
- Block-based + dynamic: gán block vuông liên tiếp trong bộ nhớ $\rightarrow$ tận dụng cache line tốt hơn, ít truy cập RAM hơn.

Có thể cải thiện dựa vào phần cứng

- Kích thước block B cần: quá nhỏ $\rightarrow$ nhiều block $\rightarrow$ overhead tăng, quá lớn $\rightarrow$ giảm lợi ích cache.
- Thử nhiều giá trị $B = 32, 64, 128, 256$ trên phần cứng mục tiêu để chọn block tối ưu.

Lợi ích tổng quát

Chọn block hợp lý kết hợp với `collapse(2)` + `schedule(dynamic)` giúp:

1. Giảm cache-miss, tăng reuse dữ liệu trong L1/L2.
2. Cân bằng tải cho nhiều thread, tránh thread idle.
3. Tối ưu hóa hiệu năng memory-bound của thuật toán tích chập 2D.
4. Cho phép scale tốt hơn khi tăng số thread hoặc ma trận lớn.

4.4 Mã giả thuật toán

```
pad = ksize / 2
block = 128

#pragma omp parallel num_threads(10)
for iteration = 1 to 100:
    #pragma omp for collapse(2) schedule(dynamic)
    for bi = 0 to Asize-1 step block:
        for bj = 0 to Asize-1 step block:
            for i = bi to min(bi+block, Asize)-1:
                for j = bj to min(bj+block, Asize)-1:
                    sum = 0
                    for ki = 0 to ksize-1:
                        for kj = 0 to ksize-1:
                            ii = i + ki - pad
                            jj = j + kj - pad
                            if ii < 0 or jj < 0 or ii >= Asize or jj >= Asize:
                                sum += 30.0 * K[ki][kj]
                            else:
                                sum += A[ii*Asize + jj] * K[ki][kj]
                    B[i*Asize + j] = sum

    #pragma omp for schedule(dynamic, N*N/10)
    for t = 0 to Asize*Asize-1:
        A[t] = B[t]
```

Algorithm 3: Parallel 2D Convolution (OpenMP, Block-based)

4.5 Độ phức tạp tính toán

Ta vẫn giữ các ký hiệu: ma trận kích thước $N \times N$, kernel kích thước $k_{\text{size}} \times k_{\text{size}}$ (ở đây $k_{\text{size}} = 3$), số vòng lặp $I = 100$, và số luồng P. Với $N = 4000$ ta có

$$N^2 = 4000^2 = 16{,}000{,}000.$$

Độ phức tạp Work

- Tổng số phép toán mỗi vòng vẫn tỉ lệ $O(k_{\text{size}}^2 N^2)$, do mỗi ô ma trận thực hiện $k_{\text{size}}^2 = 9$ phép nhân/cộng:

$$\text{Work}_{\text{block}} = O(N^2).$$

- Số phép toán không thay đổi so với parallel1, nhưng **cách truy xuất dữ liệu khác** nhờ block-based → cache hit tăng, giảm chi phí memory-bound.

Thời gian lý tưởng Nếu có P luồng và phân phối tải hoàn hảo:

$$T_{\text{ideal}}(P) = \frac{T_{\text{seq}}}{P}.$$

Thực tế với overhead Block-based + dynamic scheduling tạo thêm overhead từ:

- Phân chia block cho các thread.
- Tạo vùng song song và đồng bộ giữa các thread.
- Scheduling động (`schedule(dynamic)`) giúp cân bằng tải nhưng tốn chi phí quản lý.

Mô hình thời gian thực tế:

$$T_{\text{parallel2}}(P) \approx T_{\text{overhead,block}} + \frac{T_{\text{seq}}}{P_{\text{eff}}},$$

trong đó:

- $T_{\text{overhead,block}}$ là chi phí cố định cho thread, scheduling block, barrier.
- $P_{\text{eff}} \leq P$ là số luồng hiệu quả sau khi cân bằng tải (dynamic scheduling giúp P_{eff} gần bằng P).

Lợi ích so với parallel1

1. Cache locality cao hơn: block nhỏ gọn vừa L1/L2 → tăng cache hit, giảm truy cập RAM → giảm T_{seq} hiệu quả hơn.
2. Cân bằng tải tốt hơn: dynamic scheduling giúp thread luôn có block để xử lý, giảm idle → P_{eff} gần bằng P.
3. Overhead có thể tăng nhẹ, nhưng lợi ích cache + cân bằng tải thường bù trừ, dẫn đến $T_{\text{parallel2}} < T_{\text{parallel1}}$ thực tế.

Speedup và Efficiency

$$\text{Speedup } S(P) = \frac{T_{\text{seq}}}{T_{\text{parallel2}}(P)}, \qquad \text{Efficiency } E(P) = \frac{S(P)}{P}.$$

5 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 4: Hàm kiểm tra kết quả

Cách sử dụng, mô phỏng toàn chương trình LAB1:

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

- make run → Chạy chương trình.

Kết quả:

5.1 Phân tích Speedup và Efficiency

Dựa trên kết quả thực nghiệm, ta có thời gian chạy của các thuật toán:

Giả sử số luồng sử dụng: $N_{\text{threads}} = 10$.

```
an@LAPTOP-K3OD08GK:/mnt/d/ParCom/LAB/ParCom_Lab/LAB1$ make build
g++ -fopenmp -c ex.cpp -o ex.o
g++ -fopenmp -o ex ex.o
an@LAPTOP-K3OD08GK:/mnt/d/ParCom/LAB/ParCom_Lab/LAB1$ make run
./ex
Starting reading heat_matrix.csv
Time to read file: 34.220738 seconds
Time to sequence convolution: 50.325834 seconds
Time to parallel convolution 1: 11.574920 seconds
Time to parallel convolution 2: 10.532840 seconds
Check correctness of convolution_seq and convolution_parallel_1: Check matrix equal successfully
Check correctness of convolution_seq and convolution_parallel_2: Check matrix equal successfully
Time to check: 0.000024 seconds
an@LAPTOP-K3OD08GK:/mnt/d/ParCom/LAB/ParCom_Lab/LAB1$
```

Hình 2: *Kết quả của cả LAB1*

Thuật toán	Thời gian (s)
Sequential (convolution_seq)	50.325834
Parallel 1 (convolution_parallel_1)	11.574920
Parallel 2 (convolution_parallel_2)	10.532840

Bảng 1: *Thời gian thực nghiệm các thuật toán tích chập*

5.1.1 Công thức:

$$\text{Speedup } S = \frac{T_{\text{seq}}}{T_{\text{parallel}}}, \qquad \text{Efficiency } E = \frac{S}{N_{\text{threads}}}.$$

5.1.2 Tính toán:

Parallel 1:

$$S_1 = \frac{50.325834}{11.574920} \approx 4.35, \qquad E_1 = \frac{4.35}{10} \approx 0.435\,(43.5\%)$$

Parallel 2:

$$S_2 = \frac{50.325834}{10.532840} \approx 4.78, \qquad E_2 = \frac{4.78}{10} \approx 0.478\,(47.8\%)$$

5.1.3 Phân tích chi tiết:

- Parallel 2 có speedup cao hơn Parallel 1 nhờ cơ chế block-based kết hợp với `collapse(2)` và `dynamic scheduling`:
  - Tăng cache locality, giảm cache miss.
  - Cân bằng tải giữa các thread, tránh thread idle khi block biên hoặc block cuối nhỏ hơn.
- Efficiency của Parallel 2 cao hơn Parallel 1, tức mỗi thread được sử dụng hiệu quả hơn.
- Chưa đạt 100% do: overhead từ tạo vùng song song, barrier, dynamic scheduling và thuật toán *memory-bound*.
- Parallel 2 cải thiện khoảng 10% so với Parallel 1.