**Course**: High Performance Computing Lab

**PRN**: 22510112

**Name**: Harshavardhan Bamane

**Batch**: B1

**Title:** Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

**Problem Statement-1:**

Analyse and implement a Parallel code for below program using OpenMP.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

// sort asendingly

int compare\_asc(const void\* a, const void\* b) {

double val1 = \*(const double\*)a;

double val2 = \*(const double\*)b;

if (val1 < val2) return -1;

if (val1 > val2) return 1;

return 0;

}

// sort descendingly

int compare\_desc(const void\* a, const void\* b) {

double val1 = \*(const double\*)a;

double val2 = \*(const double\*)b;

if (val1 > val2) return -1;

if (val1 < val2) return 1;

return 0;

}

int main() {

long n = 50000000;

// as per cpu of my intel i5 12th gen

int num\_threads = 12;

omp\_set\_num\_threads(num\_threads);

// memory

double\* vec\_a = (double\*)malloc(n \* sizeof(double));

double\* vec\_b = (double\*)malloc(n \* sizeof(double));

if (vec\_a == NULL || vec\_b == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

return 1;

}

// initialization

srand(time(NULL));

for (long i = 0; i < n; i++) {

vec\_a[i] = (double)rand() / RAND\_MAX \* 100.0;

vec\_b[i] = (double)rand() / RAND\_MAX \* 100.0;

}

printf("Minimum Scalar Product Calculation\n");

printf("Vector Size: %ld\n", n);

printf("Using %d threads for the parallel part.\n\n", num\_threads);

// sequential sorting

double start\_sort = omp\_get\_wtime();

qsort(vec\_a, n, sizeof(double), compare\_asc);

qsort(vec\_b, n, sizeof(double), compare\_desc);

double end\_sort = omp\_get\_wtime();

double time\_sort = end\_sort - start\_sort;

printf("Phase 1: Sorting (Serial) took %.4f seconds.\n", time\_sort);

// parallel dot product

double min\_product = 0.0;

double start\_dot = omp\_get\_wtime();

#pragma omp parallel for reduction(+:min\_product)

for (long i = 0; i < n; i++) {

min\_product += vec\_a[i] \* vec\_b[i];

}

double end\_dot = omp\_get\_wtime();

double time\_dot = end\_dot - start\_dot;

printf("Phase 2: Dot Product (Parallel) took %.4f seconds.\n\n", time\_dot);

printf("Minimum Scalar Product: %f\n", min\_product);

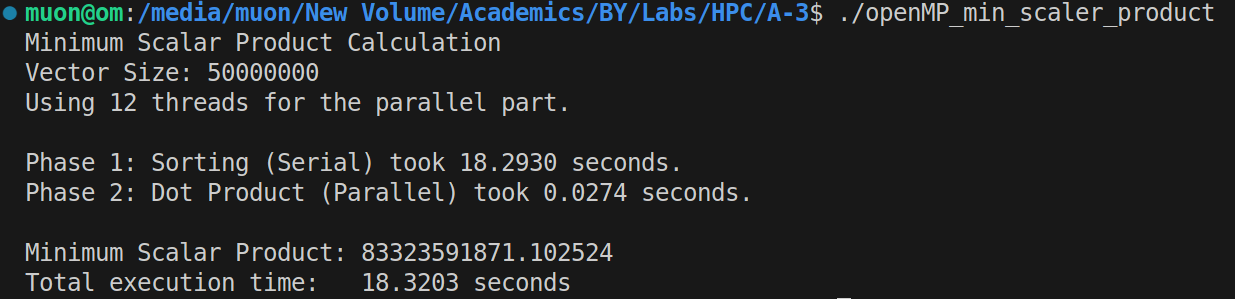
printf("Total execution time: %.4f seconds\n", time\_sort + time\_dot);

free(vec\_a);

free(vec\_b);

return 0;

}

Output: 

Analysis:

- The serialsortingphase created the bottleneck, taking 18.29seconds.

- The paralleldot product was highly optimized, finishing in just 0**.**027seconds.

- This serial portion accounted for over 99**.**8% of the total runtime.

- This result is a clear example of Amdahl'sLaw, where the serial part limits overall speedup.

**Problem Statement-2:**

Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.

ii. Explain whether or not the scaling behaviour is as expected.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

double\*\* allocate\_matrix(int size) {

double\*\* matrix = (double\*\*)malloc(size \* sizeof(double\*));

for (int i = 0; i < size; i++) {

matrix[i] = (double\*)malloc(size \* sizeof(double));

}

return matrix;

}

void initialize\_matrix(int size, double\*\* matrix) {

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

matrix[i][j] = (double)(i + j);

}

}

}

void free\_matrix(int size, double\*\* matrix) {

for (int i = 0; i < size; i++) {

free(matrix[i]);

}

free(matrix);

}

void add\_matrices(int size, double\*\* a, double\*\* b, double\*\* c) {

#pragma omp parallel for collapse(2)

for (int i = 0; i < size; i++) {

for (int j = 0; j < size; j++) {

c[i][j] = a[i][j] + b[i][j];

}

}

}

int main() {

int sizes[] = {250, 500, 750, 1000, 2000};

int num\_sizes = sizeof(sizes) / sizeof(sizes[0]);

int thread\_counts[] = {2, 4, 8, 12};

int num\_thread\_counts = sizeof(thread\_counts) / sizeof(thread\_counts[0]);

printf("--- 2D Matrix Addition Performance Analysis ---\n\n");

for (int i = 0; i < num\_sizes; i++) {

int size = sizes[i];

printf("====================================================\n");

printf("Matrix Size: %d x %d\n", size, size);

printf("----------------------------------------------------\n");

printf("Threads | Time (s) | Speedup\n");

printf("----------------------------------------------------\n");

double\*\* matrix\_a = allocate\_matrix(size);

double\*\* matrix\_b = allocate\_matrix(size);

double\*\* matrix\_c = allocate\_matrix(size);

initialize\_matrix(size, matrix\_a);

initialize\_matrix(size, matrix\_b);

// for 1 thread

omp\_set\_num\_threads(1);

double start\_time\_serial = omp\_get\_wtime();

add\_matrices(size, matrix\_a, matrix\_b, matrix\_c);

double end\_time\_serial = omp\_get\_wtime();

double time\_serial = end\_time\_serial - start\_time\_serial;

printf("%5d | %10.6f | 1.0000\n", 1, time\_serial);

// calculate speedup by changing number of threads

for (int j = 0; j < num\_thread\_counts; j++) {

int threads = thread\_counts[j];

omp\_set\_num\_threads(threads);

double start\_time\_parallel = omp\_get\_wtime();

add\_matrices(size, matrix\_a, matrix\_b, matrix\_c);

double end\_time\_parallel = omp\_get\_wtime();

double time\_parallel = end\_time\_parallel - start\_time\_parallel;

double speedup = time\_serial / time\_parallel;

printf("%5d | %10.6f | %.4f\n", threads, time\_parallel, speedup);

}

printf("====================================================\n\n");

free\_matrix(size, matrix\_a);

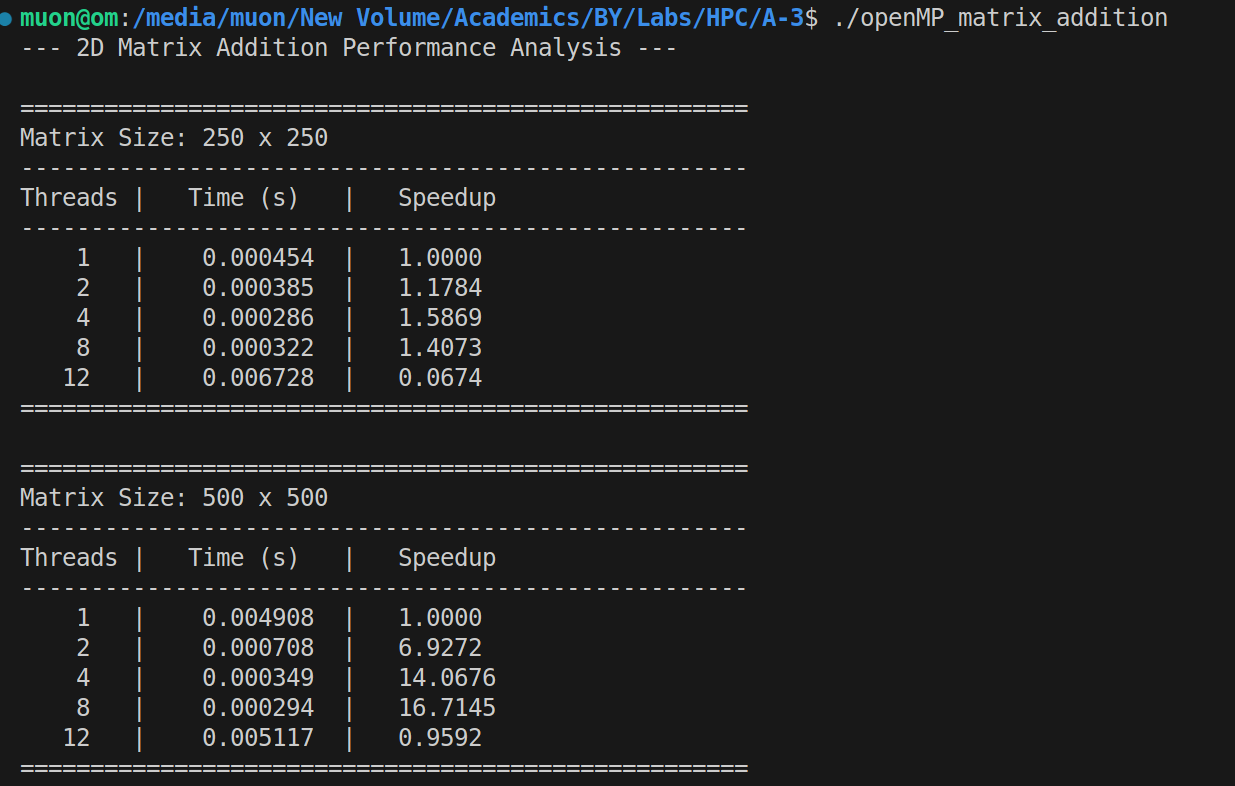
free\_matrix(size, matrix\_b);

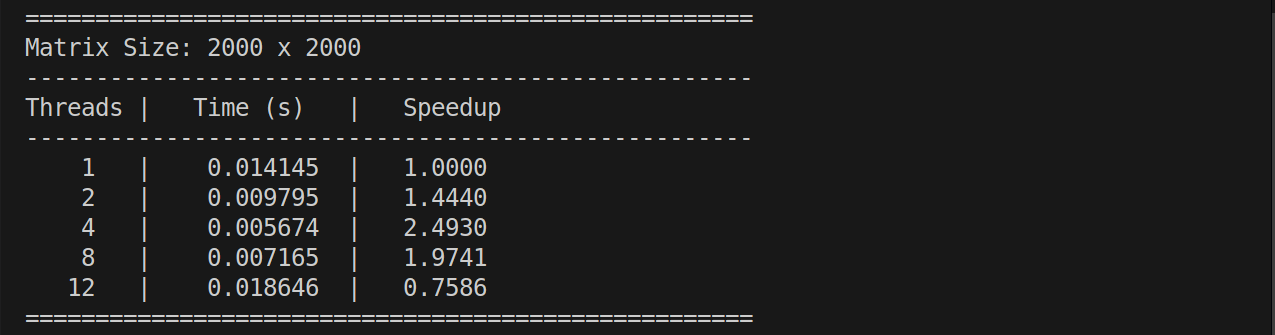
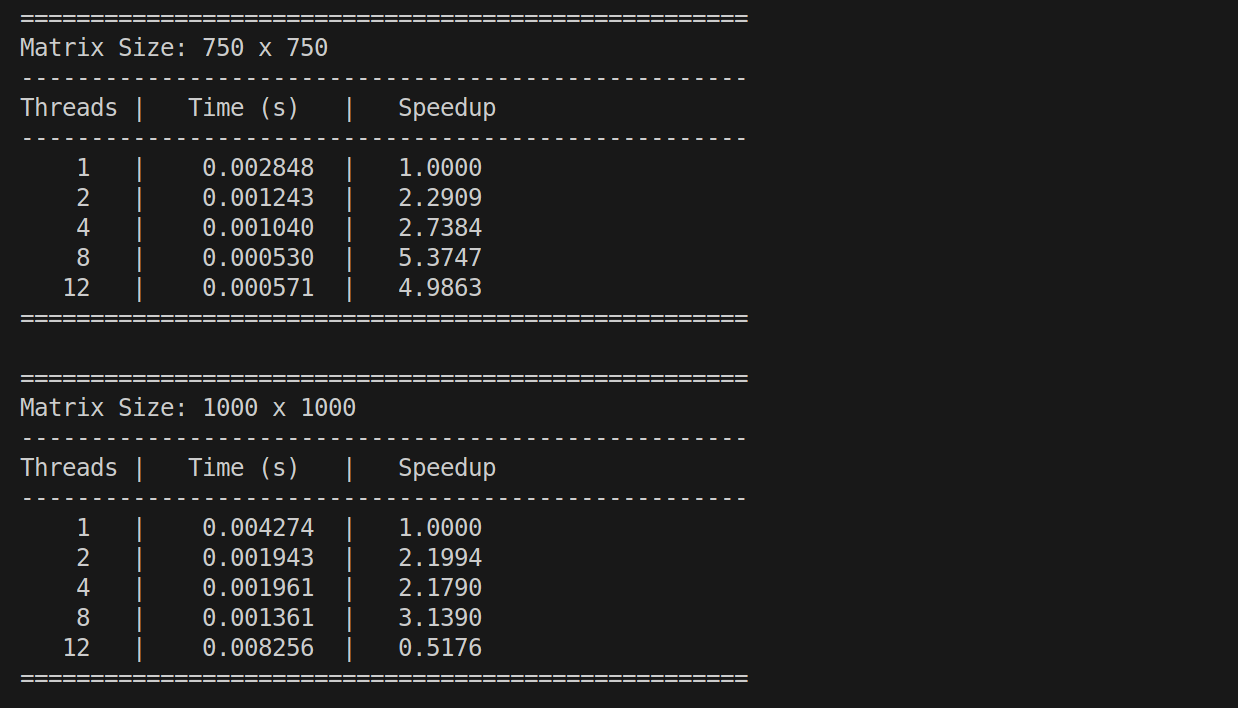
free\_matrix(size, matrix\_c);

}

return 0;

}

Output:   




Analysis:

- For small matrices, timings are unreliable and paralleloverhead causes slowdowns (e.g., 0**.**05x speedup).

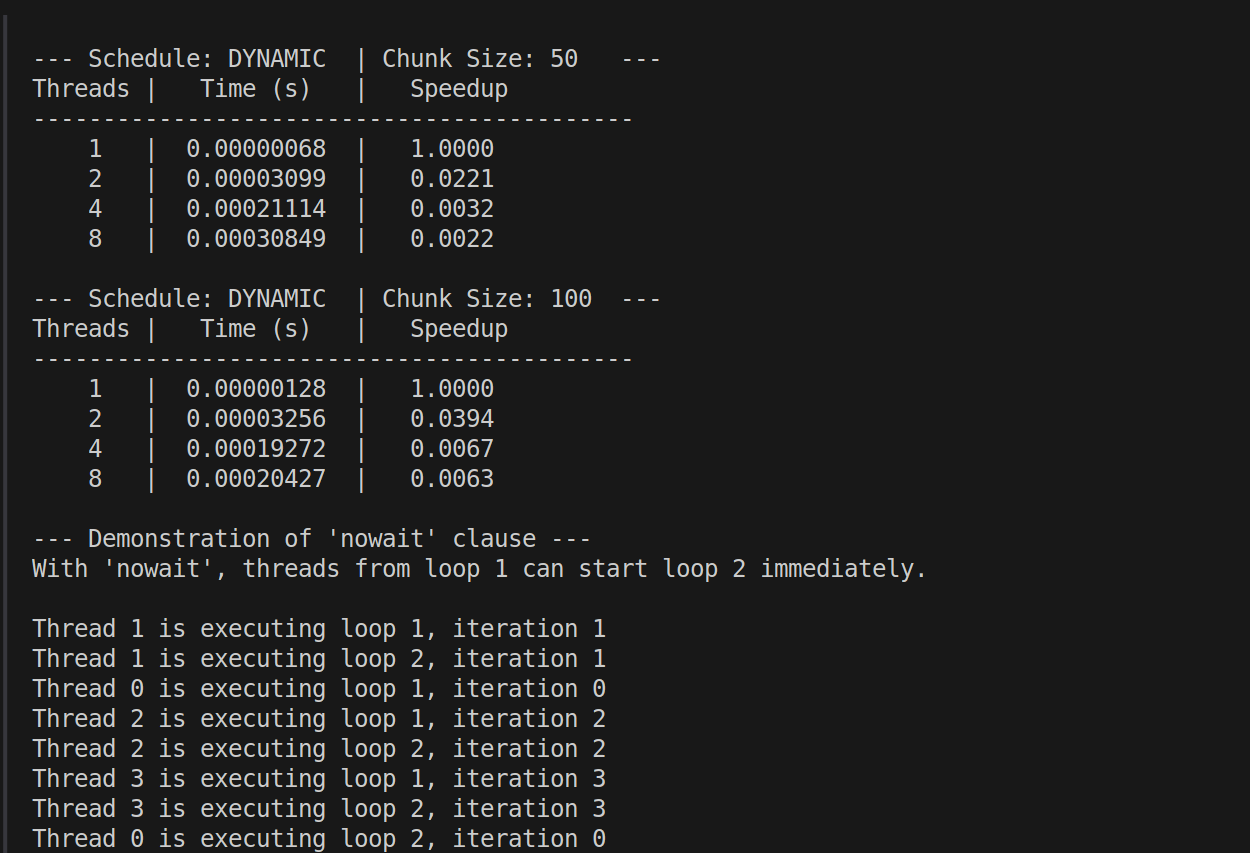
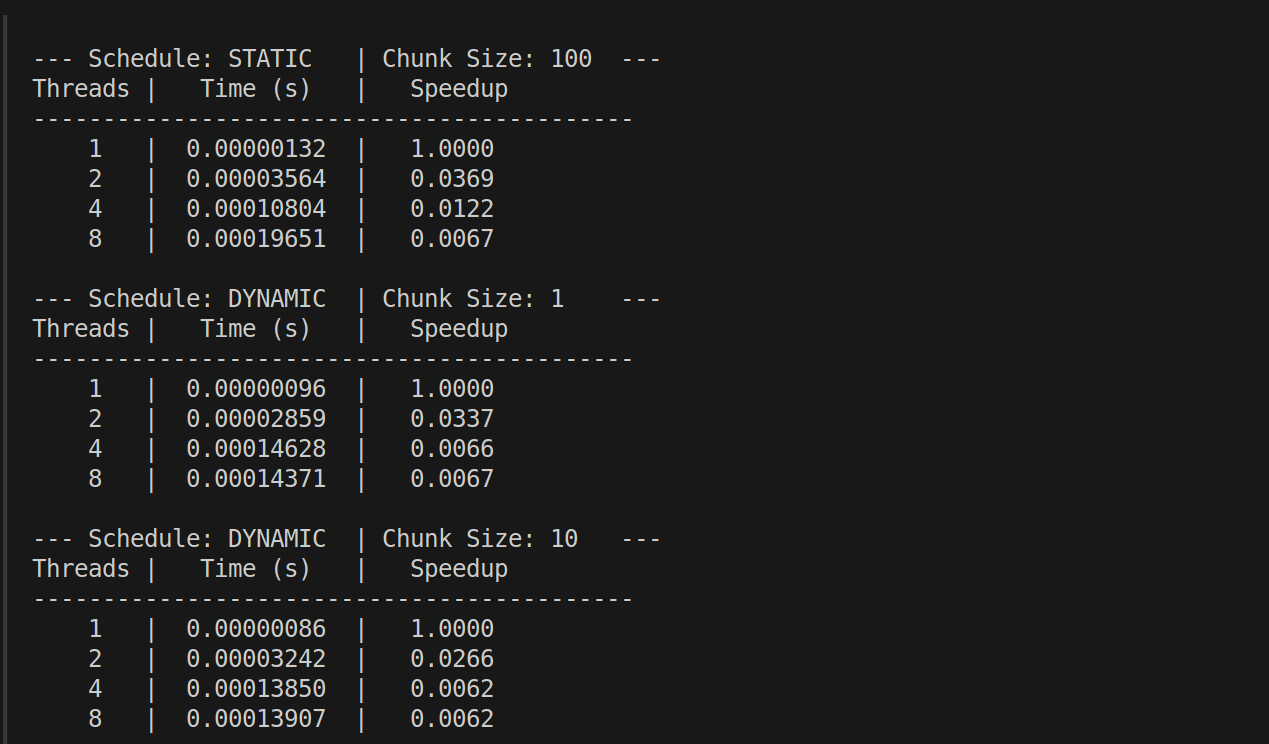
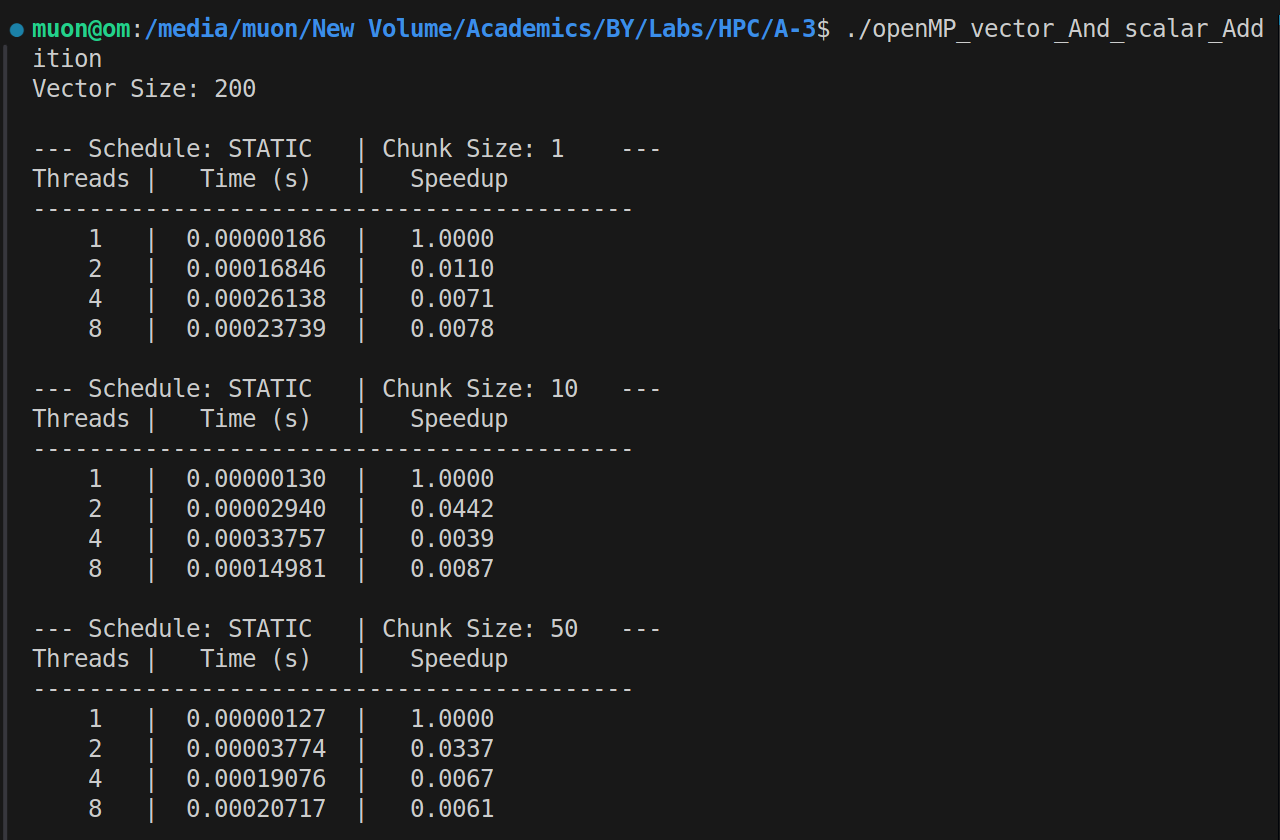
- The impossible 17**.**11x speedup is a clear benchmarking artifact from inconsistent CPU states or caching.

- Larger matrices show the true trend: performance peaks early at 3.22x speedup with just 4threads.

- This indicates the program is memory**-**bound, limited by RAM speed, not CPU processing power.

- Performance degrades beyond 4 threads because paralleloverhead costs more than any gains once the memory bus is saturated.

**Problem Statement-3:**

Output:  


Analysis:

- Significant Slowdown The primary result was not a speedup but a major slowdown. The parallel code, taking around 200 microseconds, was consistently over 100 times slower than the serial version, which took only ~1.8 microseconds.

- Cause: Dominant Parallel Overhead This slowdown is due to parallel overhead. This is the time your system spends creating, managing, and synchronizing the threads. For a tiny job like this (200 additions), this management cost far outweighs the actual computational work.

- Scheduling Policies Explained The two strategies tested are different: schedule(static) pre-assigns work in fixed chunks (low overhead), while schedule(dynamic) lets threads pull work as they become free (higher overhead but more flexible).

- Impact of Scheduling was Negligible Because the huge overhead of simply starting the threads was the main issue, the minor performance differences between static and dynamic scheduling, or changing the chunk size, were insignificant in your results.

- The Default Barrier and nowait By default, OpenMP uses a barrier after a parallel loop, forcing all threads to wait for the slowest one to finish before proceeding. The nowait clause is an optimization that removes this mandatory wait.

- Proof of nowait Functionality Your output clearly proves nowait worked. Fast threads (like Thread 1) finished the first loop and immediately started the second loop, without waiting for the artificially delayed Thread 0. This confirms the barrier was successfully removed.

[Github Link:](https://github.com/om7057/22510034-HPC_Lab.git)