**Course**: High Performance Computing Lab

**PRN**: 22510034

**Name**: Om Kulkarni

**Batch**: B2

**Title:** Study and implementation of basic OpenMP clauses

**Problem Statement-1:**

Vector Scalar Addition

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

void vector\_scalar\_add(long size, double\* vector, double scalar, double\* result\_vector) {

// work sharing construct to parallelize addition of scalar

#pragma omp parallel for

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

result\_vector[i] = vector[i] + scalar;

}

}

int main() {

// taking very large data size

long data\_size = 100000000;

double scalar = 3.14;

int thread\_counts[] = {1, 2, 4, 8, 12};

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

printf("Vector-Scalar Addition\n");

printf("Data Size: %ld elements\n", data\_size);

printf("Scalar Value: %f\n\n", scalar);

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

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

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

// dynamic memory allocation

double\* vector = (double\*)malloc(data\_size \* sizeof(double));

double\* result\_vector = (double\*)malloc(data\_size \* sizeof(double));

if (vector == NULL || result\_vector == NULL) {

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

return 1;

}

for (long i = 0; i < data\_size; i++) {

vector[i] = (double)i;

}

double time\_taken\_serial = 0.0;

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

int num\_threads = thread\_counts[i];

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

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

vector\_scalar\_add(data\_size, vector, scalar, result\_vector);

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

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

if (i == 0) {

time\_taken\_serial = time\_taken;

}

double speedup = time\_taken\_serial / time\_taken;

printf("%5d | %10.6f | %8.4f \n", num\_threads, time\_taken, speedup);

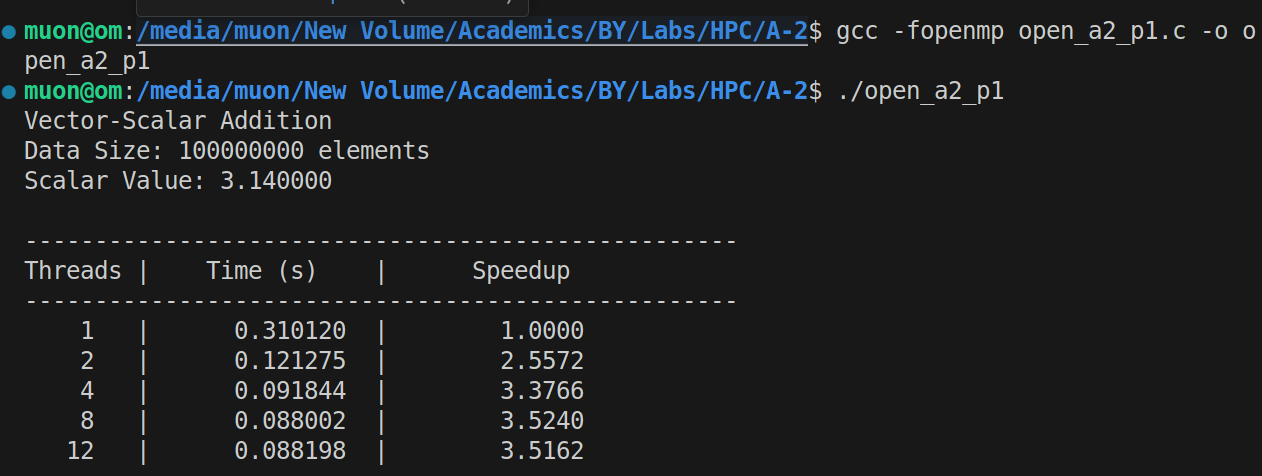
}

free(vector);

free(result\_vector);

return 0;

}

Output: 

Analysis:

The program's performance peaked at a speedup of **3.52x** on 8 threads, cutting the execution time from **0.310 seconds** down to **0.088 seconds**.

The scaling flattened out quickly after 4 threads (at 3.38x speedup) because the task is **memory-bound**, meaning it saturated your system's memory bandwidth. Consequently, at 12 threads, performance slightly worsened (speedup dropped to **3.51x**) as the overhead of managing more threads exceeded any potential gains.

**Problem Statement 2:**

Calculation of value of Pi

Source Code:

#include <stdio.h>

#include <omp.h>

int main() {

long num\_steps = 1000000000;

double step\_width = 1.0 / (double)num\_steps;

double total\_sum = 0.0;

int thread\_counts[] = {1, 2, 4, 8, 12};

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

printf("Pi Calculation using Numerical Integration\n");

printf("Number of Steps: %ld\n\n", num\_steps);

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

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

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

double time\_taken\_serial = 0.0;

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

int num\_threads = thread\_counts[i];

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

total\_sum = 0.0;

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

// 'reduction(+:total\_sum)' clause creates a private copy of 'total\_sum' for each thread.

// avoiding race condition

#pragma omp parallel for reduction(+:total\_sum)

for (long j = 0; j < num\_steps; j++) {

double x = (j + 0.5) \* step\_width;

total\_sum += 4.0 / (1.0 + x \* x);

}

double pi = step\_width \* total\_sum;

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

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

if (i == 0) {

time\_taken\_serial = time\_taken;

}

double speedup = time\_taken\_serial / time\_taken;

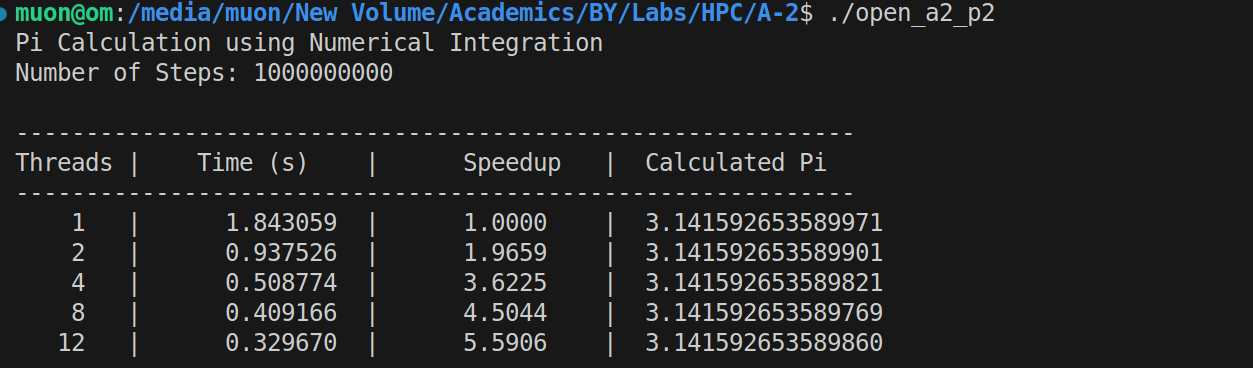
printf("%5d | %10.6f | %8.4f | %.15f\n", num\_threads, time\_taken, speedup, pi);

}

return 0;

}

Output:



Analysis:

Program reduced the execution time from **1.84 seconds** to **0.33 seconds**, achieving a **5.59x** speedup.

This performance gain directly mirrors your CPU's architecture: strong scaling on the P-cores (up to 3.62x), followed by moderate gains from the E-cores (to 4.50x), and a final boost from Hyper-Threading (to 5.59x). The result for Pi was consistently accurate.

Github Link: <https://github.com/om7057/22510034-HPC_Lab.git>