HPCL\_Assignment3

Name: Shashank Shenoy

PRN:22510013

Batch : B5

**Title of practical:**

Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

**Problem Statement 1:**

C Program to find the minimum scalar product of two vectors (dot product)

Code :

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int asc(const void \*a, const void \*b) {

    return (\*(int\*)a - \*(int\*)b);

}

int desc(const void \*a, const void \*b) {

    return (\*(int\*)b - \*(int\*)a);

}

int main() {

    int n = 10000000;

    int \*A = (int\*)malloc(n \* sizeof(int));

    int \*B = (int\*)malloc(n \* sizeof(int));

    for (int i = 0; i < n; i++) {

        A[i] = rand() % 100;

        B[i] = rand() % 100;

    }

    qsort(A, n, sizeof(int), asc);

    qsort(B, n, sizeof(int), desc);

    int threads[] = {1, 2, 4, 6, 8, 10};

    for (int t = 0; t < sizeof(threads)/sizeof(threads[0]); t++) {

        omp\_set\_num\_threads(threads[t]);

        long long result = 0;

        double start = omp\_get\_wtime();

        #pragma omp parallel for reduction(+:result) schedule(static)

        for (int i = 0; i < n; i++) {

            result += (long long)A[i] \* B[i];

        }

        double end = omp\_get\_wtime();

        printf("Threads: %2d | Min Scalar Product = %lld | Time = %.6f s\n",

               threads[t], result, end - start);

    }

    free(A);

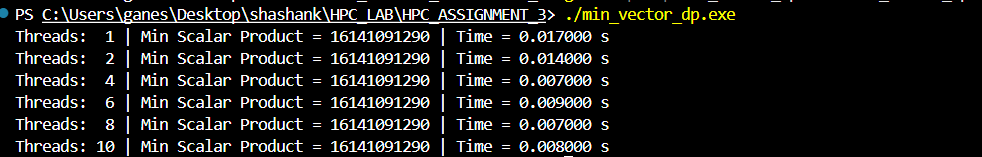
    free(B);

    return 0;

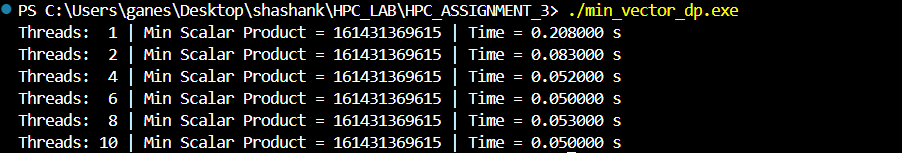
}

OUTPUT:

10^7



10^8



**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.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

void add\_matrices(int \*\*A, int \*\*B, int \*\*C, int N, int num\_threads) {

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

    #pragma omp parallel for collapse(2)

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            C[i][j] = A[i][j] + B[i][j];

        }

    }

}

int \*\*allocate\_matrix(int N) {

    int \*\*mat = (int \*\*)malloc(N \* sizeof(int \*));

    for (int i = 0; i < N; i++) {

        mat[i] = (int \*)malloc(N \* sizeof(int));

    }

    return mat;

}

void fill\_matrix(int \*\*mat, int N) {

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            mat[i][j] = rand() % 100;

}

void free\_matrix(int \*\*mat, int N) {

    for (int i = 0; i < N; i++)

        free(mat[i]);

    free(mat);

}

int main() {

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

    int threads[] = {1, 2, 4, 6, 8, 10};

    for (int s = 0; s < sizeof(sizes) / sizeof(sizes[0]); s++) {

        int N = sizes[s];

        printf("\nMatrix Size: %d x %d\n", N, N);

        int \*\*A = allocate\_matrix(N);

        int \*\*B = allocate\_matrix(N);

        int \*\*C = allocate\_matrix(N);

        fill\_matrix(A, N);

        fill\_matrix(B, N);

        for (int t = 0; t < sizeof(threads) / sizeof(threads[0]); t++) {

            int num\_threads = threads[t];

            double start = omp\_get\_wtime();

            add\_matrices(A, B, C, N, num\_threads);

            double end = omp\_get\_wtime();

            printf("Threads: %d | Time: %.6f seconds\n", num\_threads, end - start);

        }

        free\_matrix(A, N);

        free\_matrix(B, N);

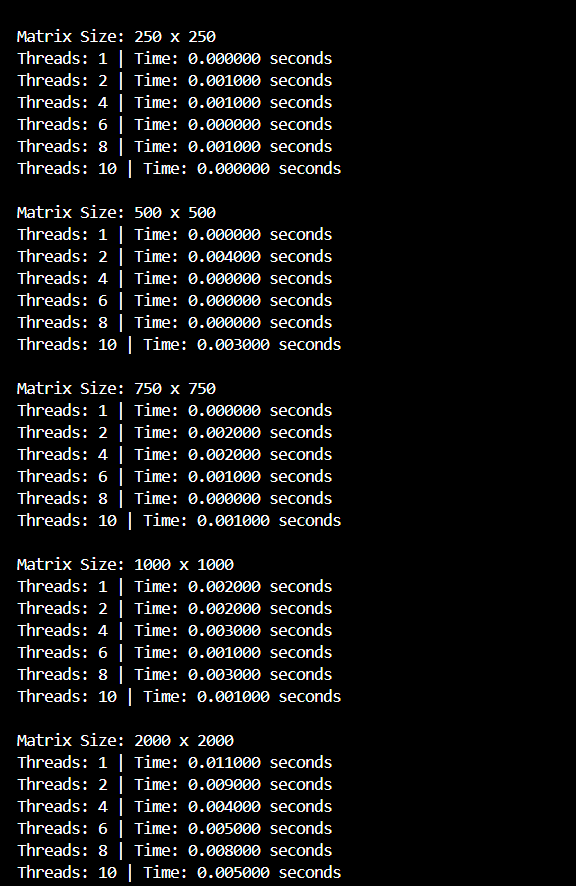
        free\_matrix(C, N);

    }

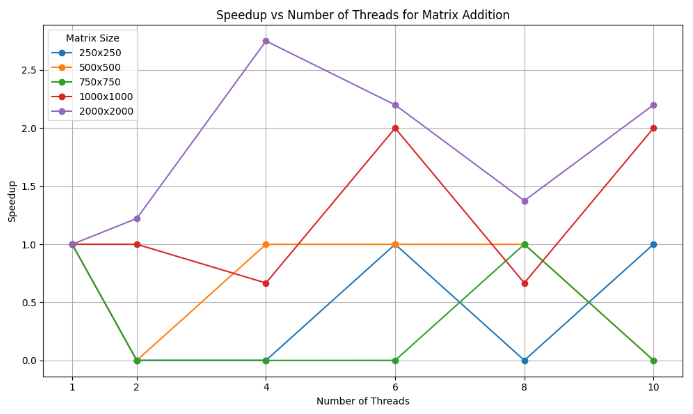
    return 0;

}

OUTPUT :



PLOT :



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

For small sizes of matrix the scaling behaviour is not significantly visible but for large size like 2000 and 1000 the speed up as threads increases can be seen also some fluctuations can be seen

**Problem Statement 3:**

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000000

#define SCALAR 5

void fill\_vector(int \*A, int n) {

    for (int i = 0; i < n; i++)

        A[i] = rand() % 100;

}

void vector\_scalar\_add\_static(int \*A, int \*B, int n, int chunk, int num\_threads) {

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

    #pragma omp parallel for schedule(static, chunk)

    for (int i = 0; i < n; i++) {

        B[i] = A[i] + SCALAR;

    }

}

void vector\_scalar\_add\_dynamic(int \*A, int \*B, int n, int chunk, int num\_threads) {

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

    #pragma omp parallel for schedule(dynamic, chunk)

    for (int i = 0; i < n; i++) {

        B[i] = A[i] + SCALAR;

    }

}

void demo\_nowait(int \*A, int \*B, int \*C, int n, int num\_threads) {

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

    #pragma omp parallel

    {

        #pragma omp for nowait

        for (int i = 0; i < n; i++) {

            B[i] = A[i] + SCALAR;

        }

        #pragma omp for

        for (int i = 0; i < n; i++) {

            C[i] = A[i] \* SCALAR;

        }

    }

}

int main() {

    int \*A = (int \*)malloc(SIZE \* sizeof(int));

    int \*B = (int \*)malloc(SIZE \* sizeof(int));

    int \*C = (int \*)malloc(SIZE \* sizeof(int));

    fill\_vector(A, SIZE);

    int chunks[] = {1, 5, 10, 20, 50, 100};

    int threads[] = {1, 2, 4};

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

    for (int t = 0; t < sizeof(threads)/sizeof(threads[0]); t++) {

        for (int c = 0; c < sizeof(chunks)/sizeof(chunks[0]); c++) {

            double start = omp\_get\_wtime();

            vector\_scalar\_add\_static(A, B, SIZE, chunks[c], threads[t]);

            double end = omp\_get\_wtime();

            printf("Threads: %d | Chunk: %3d | Time: %.6f s\n",

                   threads[t], chunks[c], end - start);

        }

    }

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

    for (int t = 0; t < sizeof(threads)/sizeof(threads[0]); t++) {

        for (int c = 0; c < sizeof(chunks)/sizeof(chunks[0]); c++) {

            double start = omp\_get\_wtime();

            vector\_scalar\_add\_dynamic(A, B, SIZE, chunks[c], threads[t]);

            double end = omp\_get\_wtime();

            printf("Threads: %d | Chunk: %3d | Time: %.6f s\n",

                   threads[t], chunks[c], end - start);

        }

    }

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

    double start = omp\_get\_wtime();

    demo\_nowait(A, B, C, SIZE, 4);

    double end = omp\_get\_wtime();

    printf("Nowait demo completed in %.6f s\n", end - start);

    free(A);

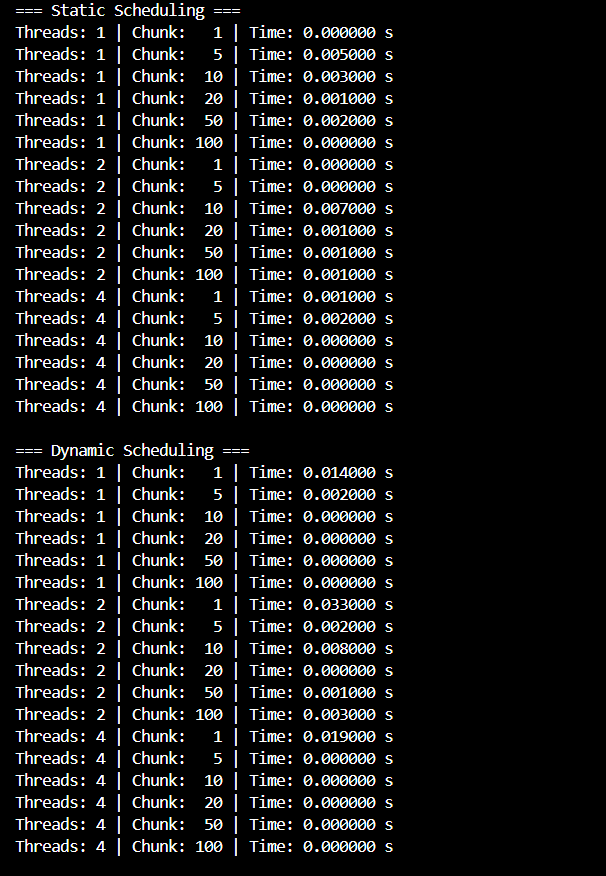
    free(B);

    free(C);

    return 0;

}

OUTPUT:





ANALYSIS :  
the performance of vector-scalar addition on a 1D array of size 1000000. Using static scheduling, we observed that increasing the chunk size generally improved performance, especially with multiple threads, as the workload was evenly distributed and overhead was minimal. In contrast, dynamic scheduling introduced noticeable delays with smaller chunk sizes due to the overhead of frequent task assignments, though performance improved with larger chunks. Additionally, the use of the nowait clause allowed threads to skip the implicit barrier after a loop and begin the next independent loop immediately, reducing idle time and improving efficiency. Overall, static scheduling with larger chunk sizes proved to be the most stable and efficient for this uniform, small-sized task.