**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 2020BTECS00037**

**Title of practical:**

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.

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

**Screenshots:**

#include <stdio.h>

#include <time.h>

#include <omp.h>

int main() {

    int size;

    printf("Enter size of array = ");

    scanf("%d", &size);

    int arr1[size];

    int arr2[size];

    for (int i = 0; i < size; i++) {

        arr1[i] = i;

        arr2[i] = i;

    }

    int n = sizeof(arr1) / sizeof(arr1[0]);

    clock\_t st = clock();

    // Ascending

    for (int i = 0; i < n; i++) {

        for (int j = i + 1; j < n; j++) {

            if (arr1[i] > arr1[j]) {

                int temp = arr1[i];

                arr1[i] = arr1[j];

                arr1[j] = temp;

            }

        }

    }

    // Descending

    for (int i = 0; i < n; i++) {

        for (int j = i + 1; j < n; j++) {

            if (arr2[i] < arr2[j]) {

                int temp = arr2[i];

                arr2[i] = arr2[j];

                arr2[j] = temp;

            }

        }

    }

    double product = 0;

    omp\_set\_num\_threads(8);

    #pragma omp parallel for schedule(static, 2) reduction(+:product)

    for (int i = 0; i < n; i++) {

        product += (double)arr1[i] \* arr2[i];

        int thread = omp\_get\_thread\_num();

        printf("\n%d. Thread = %d, Product = %f", i, thread, product);

    }

    clock\_t et = clock();

    double elapsed\_time = (double)(et - st) / CLOCKS\_PER\_SEC;

    double elapsed\_milliseconds = elapsed\_time \* 1000;

    printf("\nProduct: %f", product);

    printf("\nTime taken: %f milliseconds", elapsed\_milliseconds);

    printf("\nTime taken: %f seconds\n", elapsed\_time);

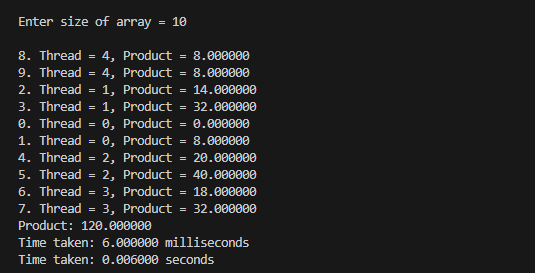
    return 0;

}

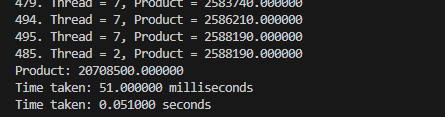
**OUTPUT:**

Keeping number of threads constant and varying size of Data.

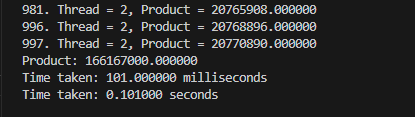
Threads = 8, Array size = 10



Threads = 8, Array size = 500

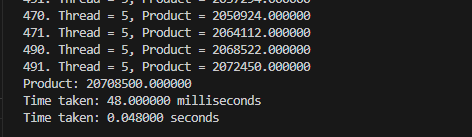


Threads = 8, Array size = 1000

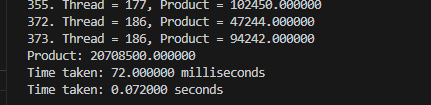


Keeping data constant and increasing number of threads.

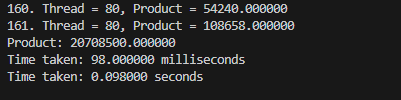
Threads = 10, Array size = 500



Threads = 250, Array size = 500



Threads = 500, Array size = 500



**Information and analysis:**

**1. schedule clause:** The schedule clause in OpenMP is used to specify how loop iterations are divided and scheduled among threads in a parallel loop construct.

1. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
2. **Dynamic Schedule (schedule(dynamic, chunk)):** Divides iterations into smaller, dynamic chunks, allowing threads to pick new chunks when they finish their current work. Useful when loop iterations have varying workloads.

**Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Threads | Data Size | Sequential Time(sec) | Parallel Time(sec) |
| **8** | **10** | 0.003000 | 0.006000 |
| **8** | **500** | 0.205000 | 0.051000 |
| **8** | **1000** | 0.152000 | 0.101000 |
| **10** | **500** | 0.001400 | 0.048000 |
| **250** | **500** | 0.001400 | 0.072000 |
| **500** | **500** | 0.001400 | 0.098000 |

**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.

#include <stdio.h>

#include <omp.h>

int main() {

    int dimension;

    printf("Enter dimension for 2D matrix = ");

    scanf("%d", &dimension);

    // Parallel code

    int mp1[dimension][dimension], mp2[dimension][dimension];

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

    #pragma omp parallel for schedule(dynamic) num\_threads(1) collapse(2)

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

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

            mp1[i][j] = i + j;

            mp2[i][i] = i - j;

        }

    }

    int ans1[dimension][dimension];

    #pragma omp parallel for schedule(dynamic) num\_threads(1) collapse(2)

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

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

            ans1[i][j] = mp1[i][j] + mp2[i][j];

        }

    }

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

    printf("\nParallel Method Time: %f seconds\n", (end\_time\_parallel - start\_time\_parallel));

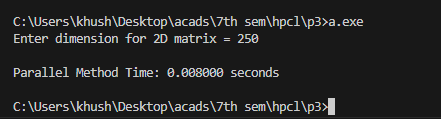
    return 0;

}

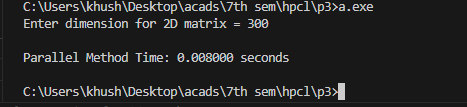
**Screenshots:**

Threads = 2

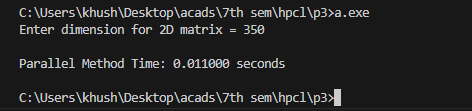
Matrix size = 250



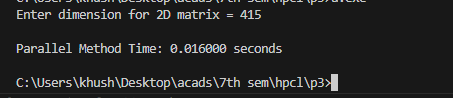
Matrix size = 300



Matrix size = 350

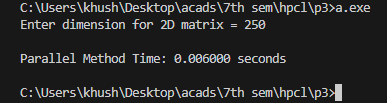


Matrix size = 415

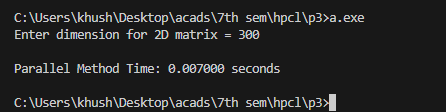


Threads = 4

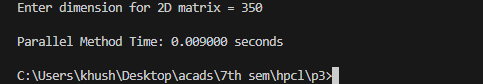
Matrix size = 250



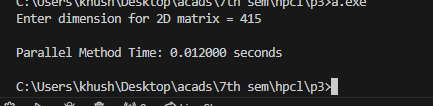
Matrix size = 300



Matrix size = 350

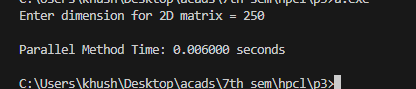


Matrix size = 415

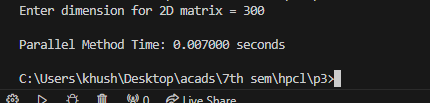


Threads = 8

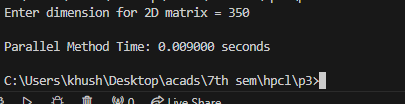
Matrix size = 250



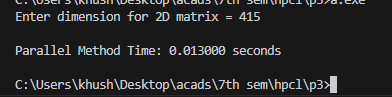
Matrix size = 300



Matrix size = 350



Matrix size = 415



**Information and analysis:**

It is observed that large number of data size requires more execution time independent from number of threads used to execute. There is slight increase in execution time while number of threads are increased, due to the mapping of logical thread to physical thread, but here increase in time is negligible.

**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.

#include <stdio.h>

#include <omp.h>

int main() {

    int n = 0;

    printf("Enter Vector size: ");

    scanf("%d", &n);

    float vector[n];

    double scalar;

    printf("Enter scalar value: ");

    scanf("%lf", &scalar);

    // Serial Code

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

    for (int i = 0; i < n; i++) {

        vector[i] = i + 100.987453323212;

    }

    for (int i = 0; i < n; i++) {

        vector[i] += scalar;

    }

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

    printf("Serial Method Time: %f seconds\n", (end\_time\_serial - start\_time\_serial));

    // Parallel Code

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

    #pragma omp parallel for schedule(static, 4) num\_threads(2) private(scalar)

    for (int i = 0; i < n; i++) {

        vector[i] = i + 100.987453323212;

    }

    #pragma omp parallel for schedule(dynamic, 4) num\_threads(2) private(scalar)

    for (int i = 0; i < n; i++) {

        vector[i] += scalar;

    }

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

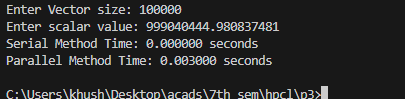
    printf("Parallel Method Time: %f seconds\n", (end\_time\_parallel - start\_time\_parallel));

    return 0;

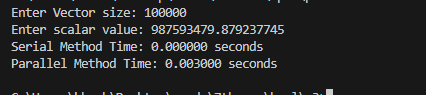
}

**Screenshots:**

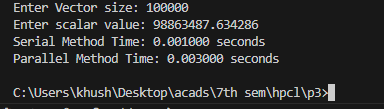
Threads = 2 Vector Size= 100000



Threads = 4 Vector Size= 100000



Threads = 8 Vector Size= 100000



As there is no sufficient data to perform parallelism, changing clause to static or dynamic, or varying the size of threads will not affect execution time.

**Information and analysis:**

1. **nowait clause:** Threads can continue execution immediately after completing their portion of work inside the parallel region, without waiting for others. They still synchronize at the end of the parallel region
2. **schedule clause:** The schedule clause in OpenMP is used to specify how loop iterations are divided and scheduled among threads in a parallel loop construct.
   1. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
   2. **Dynamic Schedule (schedule(dynamic, chunk)):** Divides iterations into smaller, dynamic chunks, allowing threads to pick new chunks when they finish their current work. Useful when loop iterations have varying workloads.