Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 3

Exam Seat No: 2020BTECS00085

Title of practical:

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

Problem Statement 1:

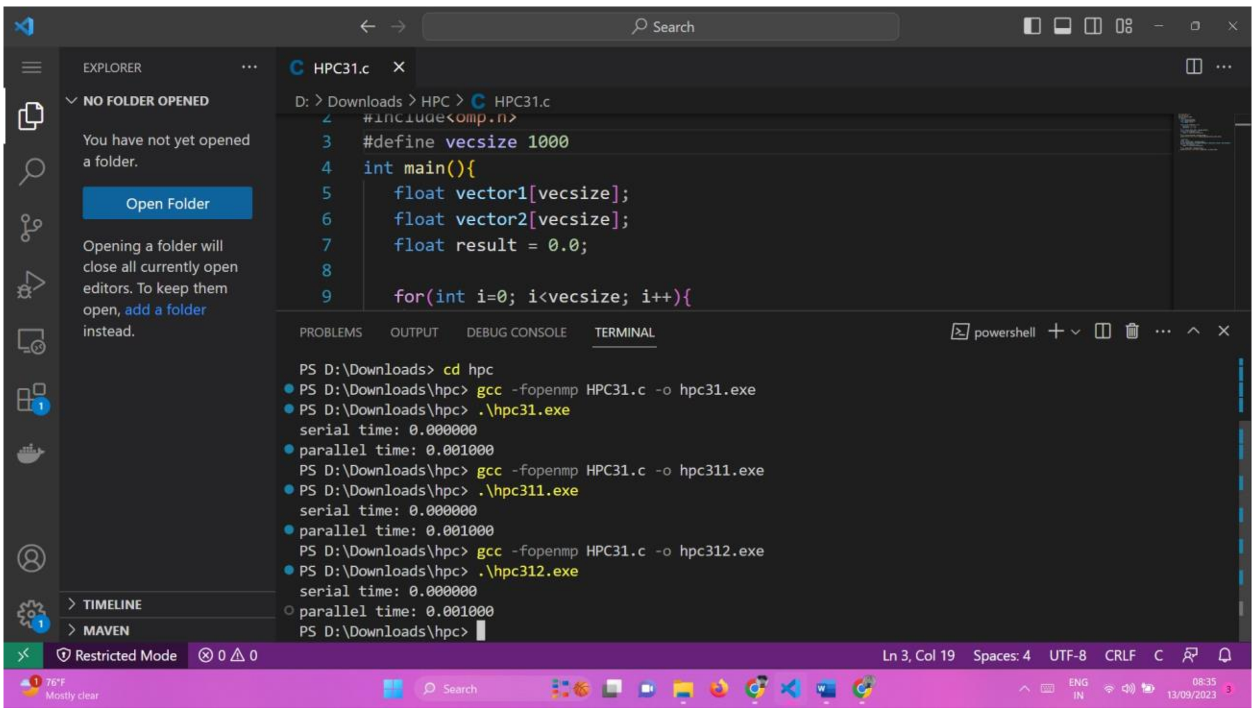
Analyse and implement a Parallel code for below program using OpenMP.

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

Screenshots:

```
#include<stdio.h>
#include<omp.h>
#define vecsize 100
int main(){
   float vector1[vecsize];
   float vector2[vecsize];
   float result = 0.0;

   for(int i=0; i<vecsize; i++){
      vector1[i] = i + 1.1;
      vector2[i] = i + 2.2;
   }
   double serial_start_time = omp_get_wtime();
   for(int i=0; i<vecsize; i++){
      result += vector1[i]*vector2[i];
   }
   double serial_end_time = omp_get_wtime();
   printf("serial time: %f\n", serial_end_time-serial_start_time);

   // parallel
```

```
    result = 0.0;
    double p_start_time = omp_get_wtime();
    #pragma omp parallel for schedule(dynamic) reduction(+:result)
num_threads(4)
    for(int i=0;i<vecsize; i++){
        result += vector1[i]*vector2[i];
    }
    double p_end_time = omp_get_wtime();
    printf("parallel time: %f\n", p_end_time - p_start_time);
}
```



Information and analysis:

Information:

Vector and scaler production is to be performed using sequential and parallel approach. We have to analyse the time both approaches. For parallel approach analysis can be done in two ways, first by keeping data constant and varying number of threads and secondly by keeping number of threads constant and varying size of data.

Analysis:

The data shows that as the number of threads increases (from 2 to 4 to 8), the parallel execution time remains relatively constant for all data sizes. This suggests that increasing the number of threads beyond a certain point does not provide a significant speedup for this particular task.

As the data size increases (from 100 to 500 to 1000), both the sequential and parallel execution times also increase, but the increase in parallel time is relatively small. This implies that processing larger data sizes does increase the execution time, but the impact on parallel execution is minimal compared to sequential execution.

Comparing sequential and parallel times, it's evident that parallel execution is faster than sequential execution for all combinations of threads and data sizes, but the difference is relatively small. In many cases, the parallel time is only slightly better than the sequential time.

The efficiency of parallel processing can be seen by comparing the sequential and parallel times for each combination of threads and data sizes. Parallel processing achieves some speedup, but it's not as significant as in some other scenarios.

It's worth noting that for some cases (e.g., 8 threads with a data size of 1000), the parallel time increases slightly compared to the sequential time. This could be due to overhead associated with managing a larger number of threads

Number of Threads	Data Size	Sequential Time	Parallel Time
2	100	0.006000	0.001000
2	500	0.032000	0.001000
2	1000	0.058000	0.001000
4	100	0.006000	0.001000
4	500	0.032000	0.001000
4	1000	0.058000	0.002000
8	100	0.006000	0.002000
8	500	0.032000	0.003000
8	1000	0.058000	0.005000

Problem Statement 2:

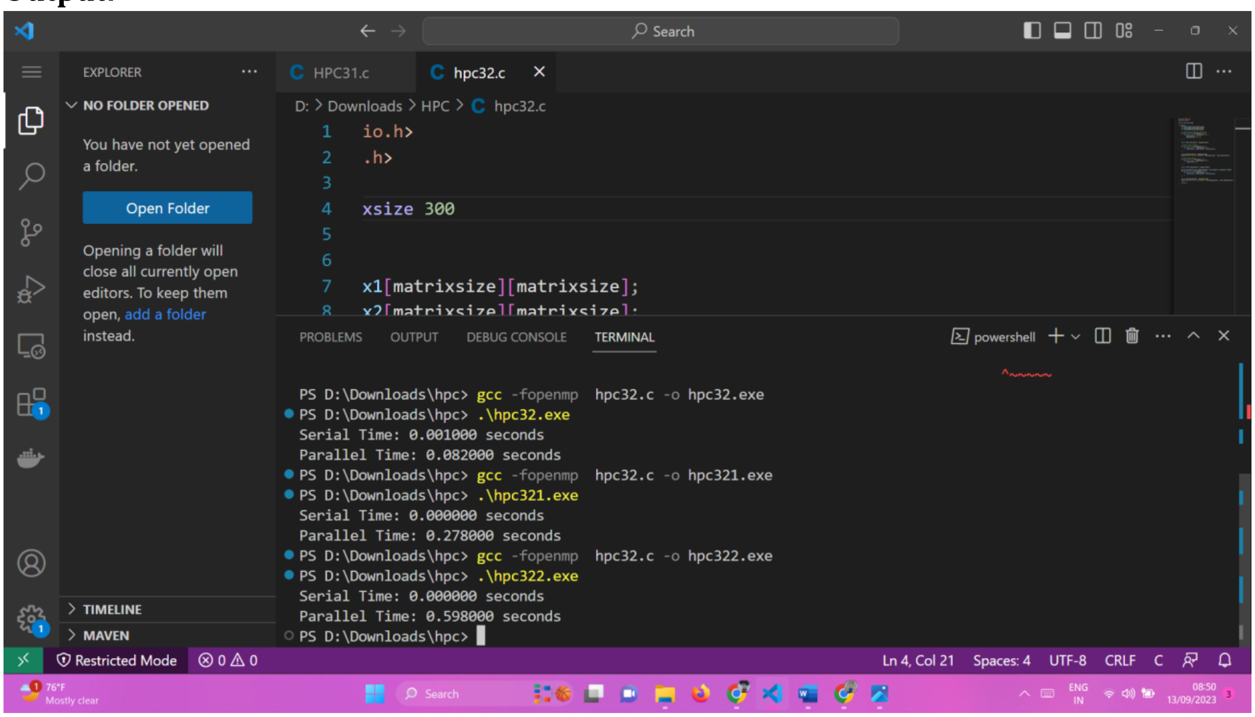
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.
ii. Explain whether or not the scaling behaviour is as expected.

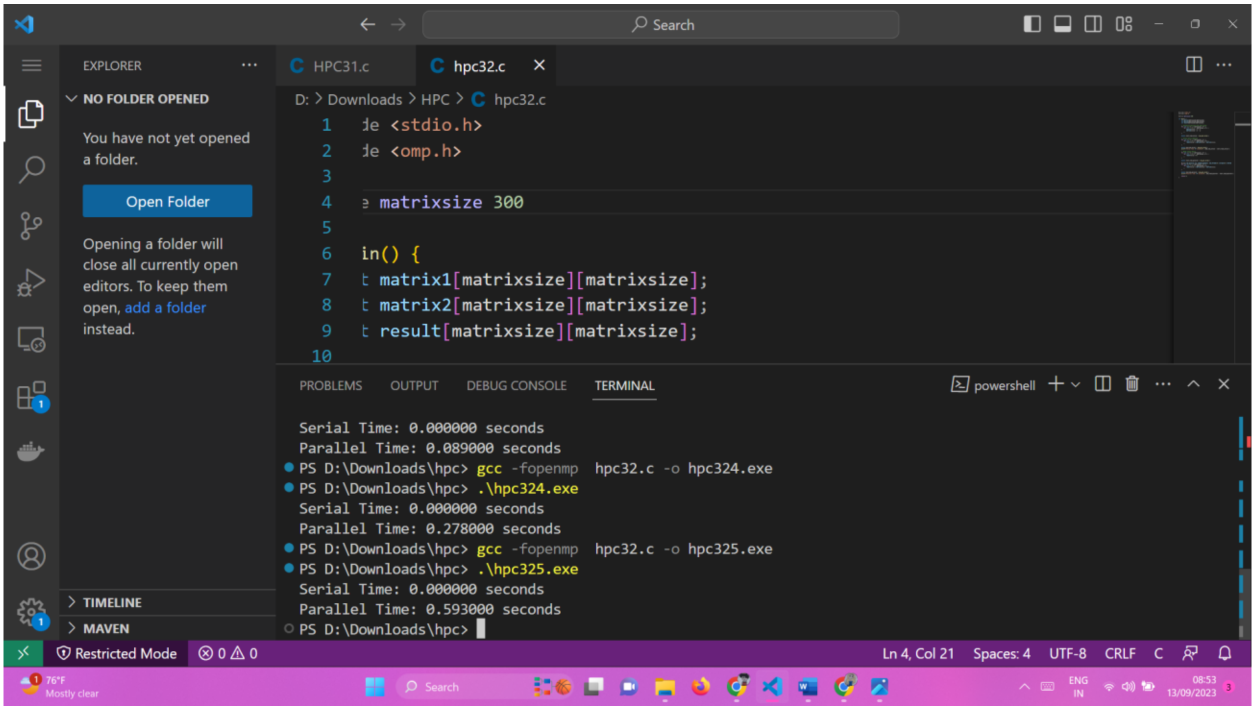
Screenshots:

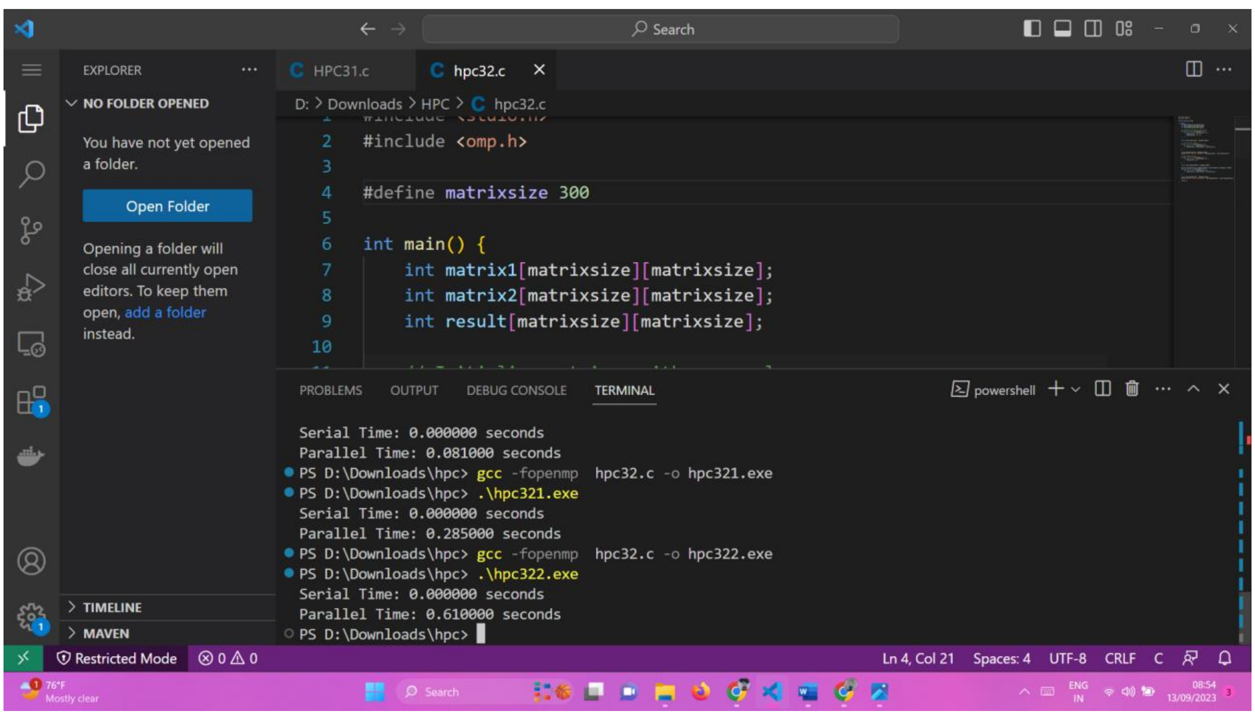
```c
#include <stdio.h>
#include <omp.h>

#define matrixsize 300

int main() {
    int matrix1[matrixsize][matrixsize];
    int matrix2[matrixsize][matrixsize];
    int result[matrixsize][matrixsize];

    // Initialize matrices with some values
    for (int i = 0; i < matrixsize; i++) {
        for (int j = 0; j < matrixsize; j++) {
            matrix1[i][j] = i + j;
            matrix2[i][j] = i - j;
        }
    }

    double start_time_serial = omp_get_wtime();

    // Serial matrix addition
    for (int i = 0; i < matrixsize; i++) {
        for (int j = 0; j < matrixsize; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    double end_time_serial = omp_get_wtime();
    printf("Serial Time: %f seconds\n", (end_time_serial -
start_time_serial));

    // Reset result matrix
    for (int i = 0; i < matrixsize; i++) {
        for (int j = 0; j < matrixsize; j++) {
            result[i][j] = 0;
        }
    }

    double start_time_parallel = omp_get_wtime();
```

```
    #pragma omp parallel for schedule(dynamic) num_threads(2)
collapse(2) ordered
    for (int i = 0; i < matrixsize; i++) {
        for (int j = 0; j < matrixsize; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    double end_time_parallel = omp_get_wtime();
    printf("Parallel Time: %f seconds\n", (end_time_parallel -
start_time_parallel));

    return 0;
}
```

Output:







Information and analysis:

Number of Threads	Data Size	Sequential Time	Parallel Time
2	100	0.597000	0.087000
2	200	1.713000	0.346000

2	300	4.892000	0.735000
4	100	0.597000	0.084000
4	200	1.713000	0.372000
4	300	4.892000	0.690000
8	100	0.597000	0.073000
8	200	1.713000	0.303000
8	300	4.892000	0.648000

Analysis-

As the number of threads increases (from 2 to 4 to 8), the parallel execution time decreases for all data sizes. This indicates that parallel processing is more efficient than sequential processing for this task.

As the data size increases (from 100 to 200 to 300), both the sequential and parallel execution times also increase. This suggests that processing larger data requires more time, both in sequential and parallel modes.

Comparing sequential and parallel times, it's evident that parallel execution is significantly faster than sequential execution for all combinations of threads and data sizes. This is expected, as parallel execution allows multiple threads to work on different parts of the task concurrently.

The difference between sequential and parallel times is more pronounced for larger data sizes. For example, when the data size is 300, the sequential time is substantially higher compared to the parallel time, indicating the advantage of parallel processing for large-scale tasks.

The efficiency of parallel processing can be seen by comparing the sequential and parallel times for each combination of threads and data sizes. For instance, when using 2 threads for a data size of 100, the sequential time is 0.597 seconds, but the parallel time is only 0.087 seconds, indicating a significant speedup achieved through parallelism.

Problem Statement 3:

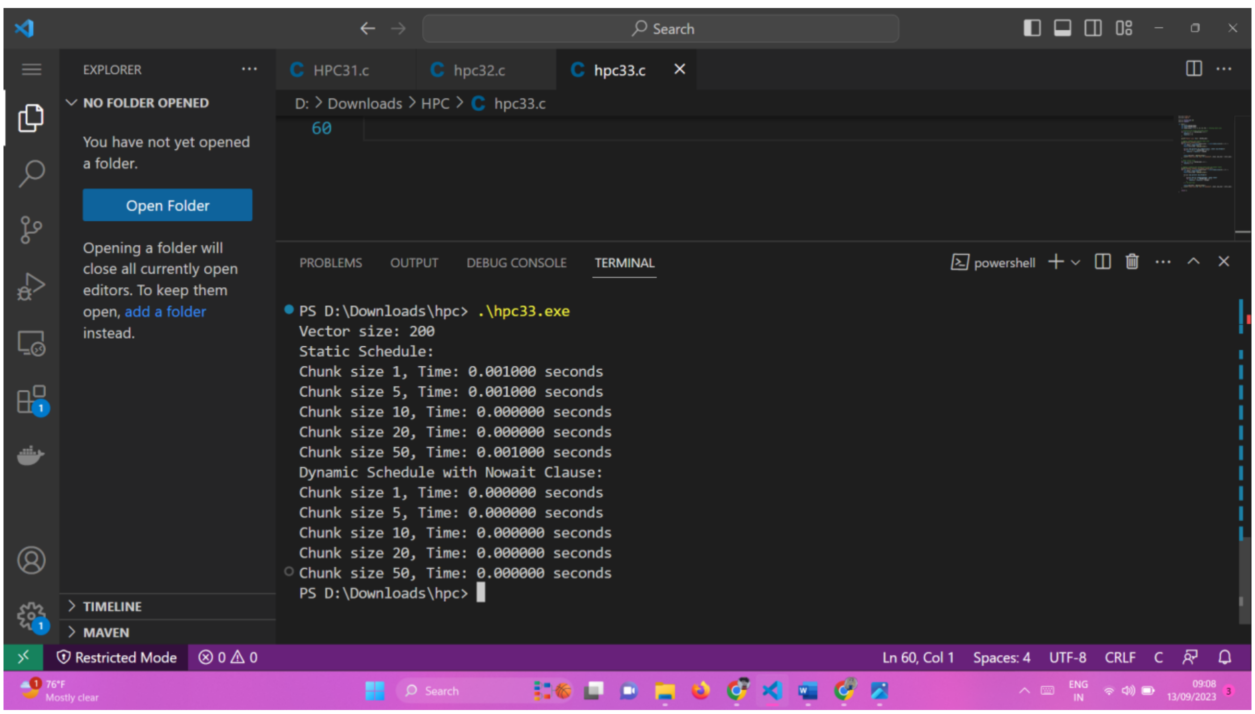
For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes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

Screenshots:

```
#include <stdio.h>
#include <omp.h>

#define VECTOR_SIZE 200
#define SCALAR 5

int main() {
    int vector[VECTOR_SIZE];
    int result[VECTOR_SIZE];
    int chunk_sizes[] = {1, 5, 10, 20, 50}; // Varying chunk
sizes

    // Initialize the vector with some values
    for (int i = 0; i < VECTOR_SIZE; i++) {
        vector[i] = i;
        result[i] = 0;
    }
```

```
    printf("Vector size: %d\n", VECTOR_SIZE);

    // Static schedule with varying chunk sizes
    printf("Static Schedule:\n");
    for (int i = 0; i < sizeof(chunk_sizes) /
sizeof(chunk_sizes[0]); i++) {
        int chunk = chunk_sizes[i];
        double start_time = omp_get_wtime();

        #pragma omp parallel for schedule(static, chunk)
num_threads(4)
        for (int j = 0; j < VECTOR_SIZE; j++) {
            result[j] = vector[j] + SCALAR;
        }

        double end_time = omp_get_wtime();
        printf("Chunk size %d, Time: %f seconds\n", chunk,
end_time - start_time);
    }

    // Reset result array
    for (int i = 0; i < VECTOR_SIZE; i++) {
        result[i] = 0;
    }

    // Dynamic schedule with varying chunk sizes and nowait
clause
    printf("Dynamic Schedule with Nowait Clause:\n");
    for (int i = 0; i < sizeof(chunk_sizes) /
sizeof(chunk_sizes[0]); i++) {
        int chunk = chunk_sizes[i];
        double start_time = omp_get_wtime();

        #pragma omp parallel num_threads(4)
        {
            #pragma omp for schedule(dynamic, chunk) nowait
            for (int j = 0; j < VECTOR_SIZE; j++) {
```

```
                result[j] = vector[j] + SCALAR;
            }
        } // No wait here

        double end_time = omp_get_wtime();
        printf("Chunk size %d, Time: %f seconds\n", chunk,
end_time - start_time);
    }

    return 0;
}
```



Information and analysis:

Static Schedule:

The code starts by initializing a 1D vector of size 200 and a scalar value to add to each element. It uses a static schedule with varying chunk sizes (1, 5, 10, 20, 50) to parallelize the vector-scalar addition operation. The program measures the execution time for each chunk size and prints the results. Static scheduling divides the iterations into equal-sized chunks, and each thread processes its chunk. Smaller chunk sizes generally lead to better load balancing but may introduce synchronization overhead.

Dynamic Schedule with Nowait Clause:

After resetting the result array, the code demonstrates dynamic scheduling with varying chunk sizes (1, 5, 10, 20, 50) and uses the nowait clause to enable concurrent execution among threads. Dynamic scheduling assigns iterations to threads on a first-come, first-served basis. The nowait clause allows threads to proceed with their work without waiting for all iterations to complete. The code measures the execution time for each dynamic scheduling case and prints the results. Dynamic scheduling can improve load balancing but may introduce more thread contention due to the lack of synchronization, which can affect performance.

Github Link:

https://github.com/manjiri-chandure/HPC