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: 2020BTECS00041

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)

With Reduction Clause:

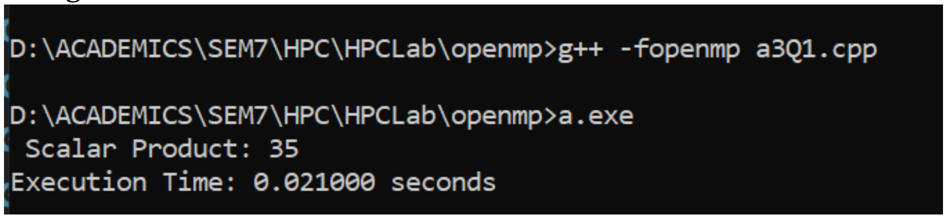
```
#include<bits/stdc++.h>
#include<iostream>
#include <omp.h>
#include <time.h>
using namespace std;
#define NUM_thread 100
#define N 100
int main() {
    // Define the two input vectors
    std::vector<int> vector1 = {1, 2, 3, 4, 5};
    std::vector<int> vector2 = {5, 4, 3, 2, 1};
    srand(1);
    // vector<int>vector1(N),vector2(N);
    // for(int i=0;i<N;i++)
    // {
    //     vector1[i]=rand()+200;
    //     vector2[i]=rand()*23;
    // }

    int product=0;
    // Record the starting time
    clock_t start_time = clock();
    // Parallel section
    int i;
    #pragma omp parallel for num_threads(NUM_thread) reduction(+ : product)
    for ( i = 0; i < vector1.size(); i++) {
        product += vector1[i] * vector2[i];
    }

    std::cout << " Scalar Product: " << product << std::endl;
     // Record the ending time
    clock_t end_time = clock();

    // Calculate the elapsed time in seconds
    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    // Print the execution time
    printf("Execution Time: %f seconds\n", elapsed_time);


    return 0;
}
```

With Nowait clause:

```
#include<bits/stdc++.h>
#include<iostream>
#include <omp.h>
#include <time.h>
using namespace std;
#define NUM_thread 100
#define N 100
int main() {
    // Define the two input vectors
    std::vector<int> vector1 = {1, 2, 3, 4, 5};
    std::vector<int> vector2 = {5, 4, 3, 2, 1};
    srand(1);
    // vector<int>vector1(N),vector2(N);
    // for(int i=0;i<N;i++)
    // {
    //     vector1[i]=rand()+200;
    //     vector2[i]=rand()*23;
    // }

    int product=0;
    // Record the starting time
    clock_t start_time = clock();
    // Parallel section
    int i;
    #pragma omp  for  nowait
    // num_threads(NUM_thread) reduction(+ : product)
    for ( i = 0; i < vector1.size(); i++) {
        product += vector1[i] * vector2[i];
    }

    std::cout << " Scalar Product: " << product << std::endl;
     // Record the ending time
    clock_t end_time = clock();

    // Calculate the elapsed time in seconds
    double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    // Print the execution time
    printf("Execution Time: %f seconds\n", elapsed_time);
  return 0;
}
```

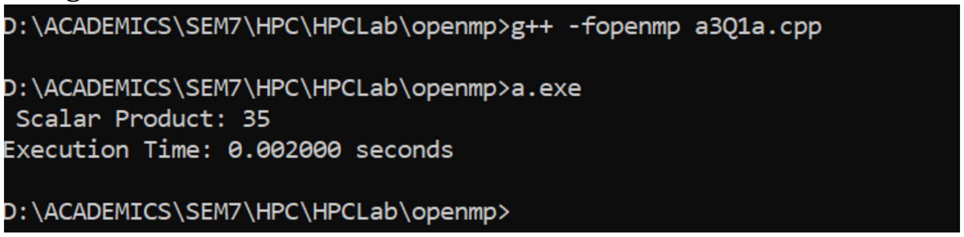
Screenshots:

Using reduction clause:



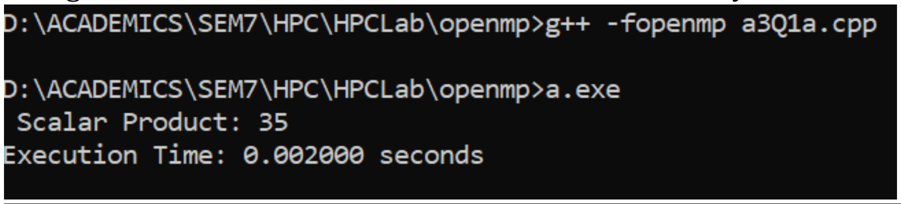
```
D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>g++ -fopenmp a3Q1.cpp

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>a.exe
 Scalar Product: 35
Execution Time: 0.021000 seconds
```

Using nowait clause:

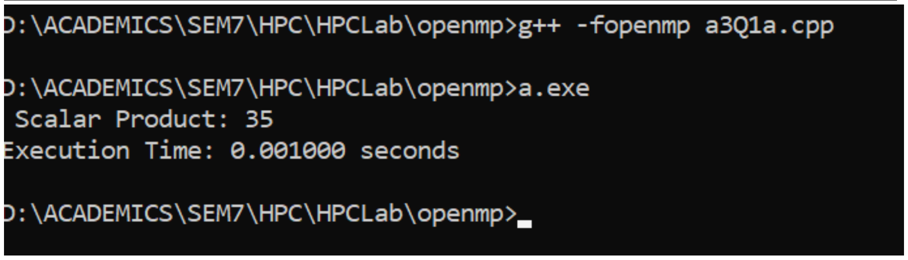


```
D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>g++ -fopenmp a3Q1a.cpp

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>a.exe
 Scalar Product: 35
Execution Time: 0.002000 seconds

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>
```

Using Reduction with Static schedule and then with Dynamic schedule:



```
D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>g++ -fopenmp a3Q1a.cpp

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>a.exe
 Scalar Product: 35
Execution Time: 0.002000 seconds
```



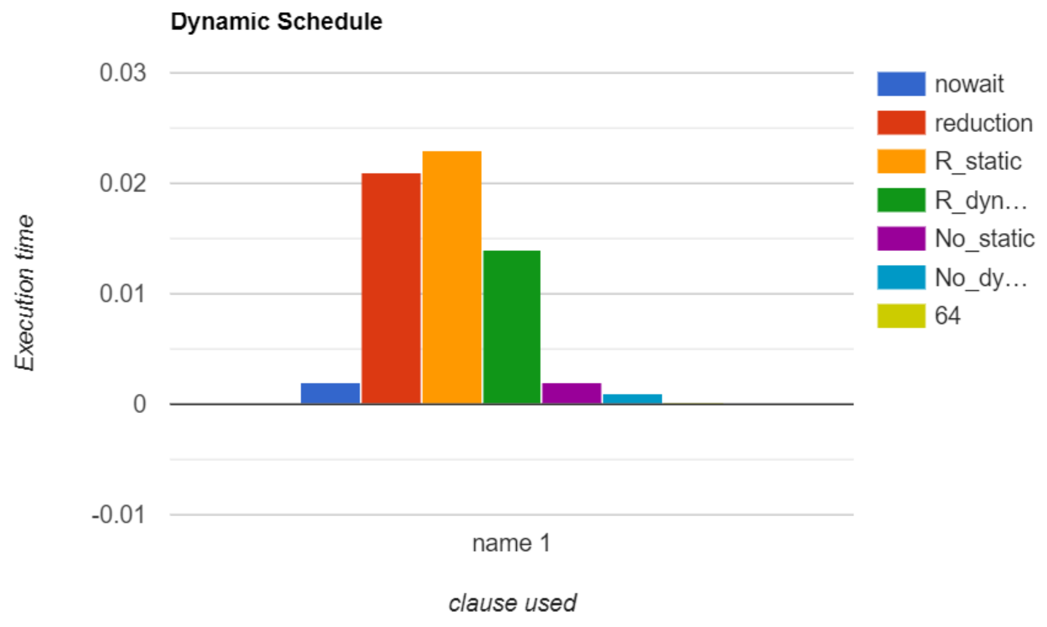
```
D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>g++ -fopenmp a3Q1a.cpp

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>a.exe
 Scalar Product: 35
Execution Time: 0.001000 seconds

D:\ACADEMICS\SEM7\HPC\HPCLab\openmp>
```

Information and analysis:
The reason for the code using the `reduction` clause taking more time for execution compared to the one with only the `nowait` clause is likely due to the nature of the work being performed and how it interacts with parallelism.

1. **Reduction Clause Overhead**: When you use the `reduction` clause in OpenMP, the compiler and runtime system need to perform additional operations to combine the partial results from each thread. This can introduce some overhead, especially for complex reduction operations. The `reduction` clause is intended for situations where you need to compute a reduction operation accurately, such as summing values, and it ensures the correct result at the end of the parallel region. This overhead can make the code slightly slower than a simple `nowait` clause.

2. **Parallelization Granularity**: The effectiveness of parallelization depends on the granularity of the parallel tasks. If the work done within the loop is relatively small compared to the overhead of thread creation and synchronization, the parallel version of the code might not show a significant speedup compared to the serial version. In such cases, a simple `nowait` clause may make the code faster because it avoids some of the synchronization overhead.

3. **Additional Work**: In the code with the `nowait` clause, there is "additional work" done after the loop. If this additional work is substantial and can be efficiently parallelized, it may offset the difference in execution time. The `nowait` clause allows the program to overlap this additional work with the loop execution, potentially making the overall execution time shorter.

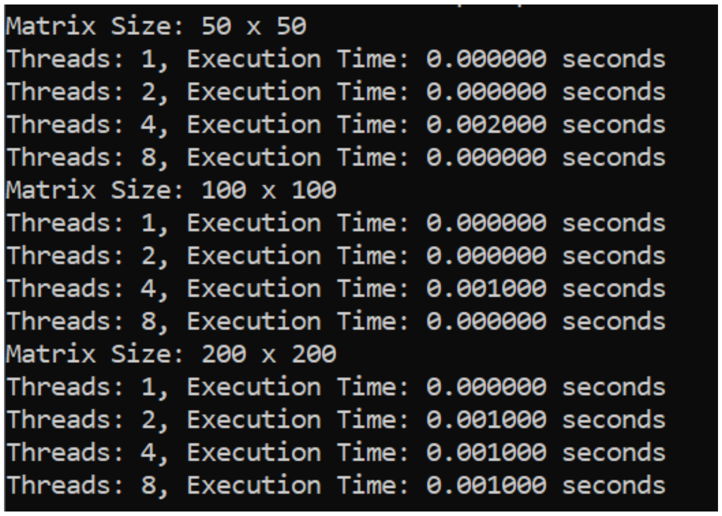


Problem Statement 2:

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 <stdlib.h>
#include <omp.h>
#include <time.h>



int main() {
    // Varying matrix sizes
    srand(0);
    int sizes[] = {50, 100, 200, 300};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    // Varying thread counts
    int num_threads[] = {1, 2, 4, 8};
    int num_thread_counts = sizeof(num_threads) /
sizeof(num_threads[0]);

    for (int size_idx = 0; size_idx < num_sizes; size_idx++) {
        int size = sizes[size_idx];
        int A[size][size];
        int B[size][size];
         for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                A[i][j] = rand() % 100;
                            }
        }
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                B[i][j] = rand() * 13;
                            }
        }

       int C[size][size];

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

        for (int thread_idx = 0; thread_idx < num_thread_counts;
thread_idx++) {
            int num_thread = num_threads[thread_idx];
            omp_set_num_threads(num_thread);
            clock_t start_time = clock();

            #pragma omp parallel for
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    C[i][j] = A[i][j] + B[i][j];
                }
            }

            clock_t end_time = clock();
            double execution_time =(double)( end_time -
start_time)/CLOCKS_PER_SEC;

            printf("Threads: %d, Execution Time: %f seconds\n",
num_thread, execution_time);
        }

        for (int i = 0; i < size; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
        }
        free(A);
        free(B);
        free(C);
    }

    return 0;
}
```

Screenshots:



```
Matrix Size: 50 x 50
Threads: 1, Execution Time: 0.000000 seconds
Threads: 2, Execution Time: 0.000000 seconds
Threads: 4, Execution Time: 0.002000 seconds
Threads: 8, Execution Time: 0.000000 seconds
Matrix Size: 100 x 100
Threads: 1, Execution Time: 0.000000 seconds
Threads: 2, Execution Time: 0.000000 seconds
Threads: 4, Execution Time: 0.001000 seconds
Threads: 8, Execution Time: 0.000000 seconds
Matrix Size: 200 x 200
Threads: 1, Execution Time: 0.000000 seconds
Threads: 2, Execution Time: 0.001000 seconds
Threads: 4, Execution Time: 0.001000 seconds
Threads: 8, Execution Time: 0.001000 seconds
```

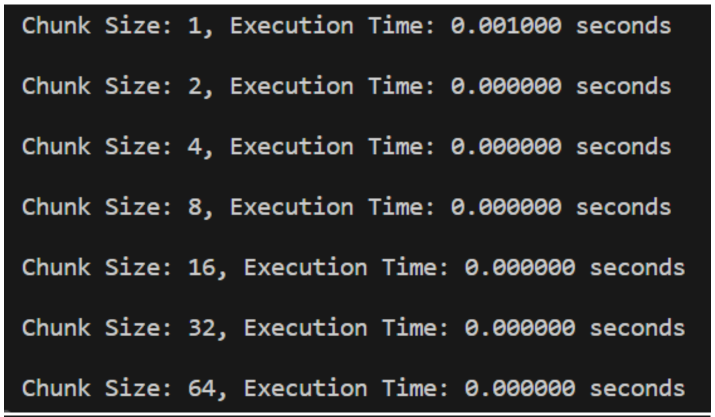
Information and analysis:

Threads / Input Size	1	2	4	8
50x50	**0.000 sec**	**0.001 sec**	**0.002 sec**	**0.000 sec**
100x100	**0.000 sec**	**0.000 sec**	**0.001 sec**	**0.000 sec**
200x200	**0.000 sec**	**0.001 sec**	**0.001 sec**	**0.001 sec**

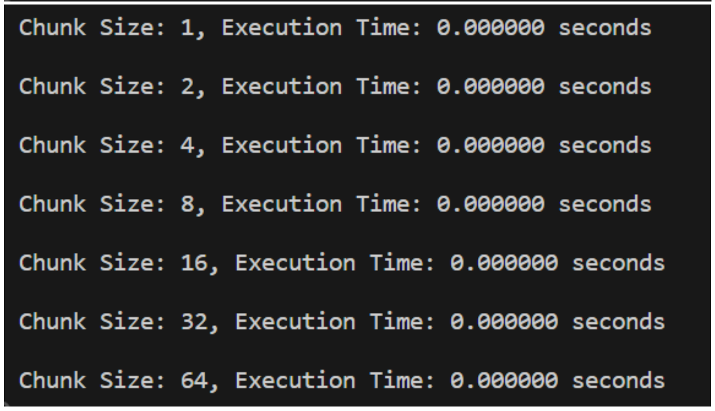
Problem Statement 3:

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:



```
Chunk Size: 1, Execution Time: 0.001000 seconds

Chunk Size: 2, Execution Time: 0.000000 seconds

Chunk Size: 4, Execution Time: 0.000000 seconds

Chunk Size: 8, Execution Time: 0.000000 seconds

Chunk Size: 16, Execution Time: 0.000000 seconds

Chunk Size: 32, Execution Time: 0.000000 seconds

Chunk Size: 64, Execution Time: 0.000000 seconds
```



```
Chunk Size: 1, Execution Time: 0.000000 seconds

Chunk Size: 2, Execution Time: 0.000000 seconds

Chunk Size: 4, Execution Time: 0.000000 seconds

Chunk Size: 8, Execution Time: 0.000000 seconds

Chunk Size: 16, Execution Time: 0.000000 seconds

Chunk Size: 32, Execution Time: 0.000000 seconds

Chunk Size: 64, Execution Time: 0.000000 seconds
```

Information and analysis:

Chunk Size	Execution Time for static	Execution Time for dynamic
1	0.001000 sec	0.000000 sec
2	0.000000 sec	0.000000 sec
4	0.000000 sec	0.000000 sec
8	0.000000 sec	0.000000 sec
16	0.000000 sec	0.000000 sec
32	0.000000 sec	0.000000 sec
64	0.000000 sec	0.000000 sec

Github Link: https://github.com/Nikita226/HPCL/tree/main/A3