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: 2020BTECS00042

Title of practical:

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

1) **Nowait:**
The nowait clause is used to indicate that a barrier at the end of a parallel loop construct should be omitted. This can help reduce synchronization overhead when multiple parallel loop constructs are present within the same parallel region and can run independently without the need to synchronize at the end of each loop. The absence of the barrier can potentially improve performance by allowing threads to continue executing without waiting for others to complete.
2) **Reduction:**
The reduction clause is used to perform a reduction operation on a variable across multiple threads, accumulating the results into a single value. This is useful for operations like summing, finding the maximum or minimum, etc., without race conditions.
3) **Ordered:**
The ordered clause ensures that specific portions of code within a parallel loop are executed in the original sequential order of the loop. It is often used when there are dependencies between loop iterations that must be maintained.
4) **Collapse:**
The collapse clause is used to collapse multiple nested loops into a single loop for parallelization. This can be useful when there are nested loops and you want to parallelize across both loop levels.

Problem Statement 1:

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

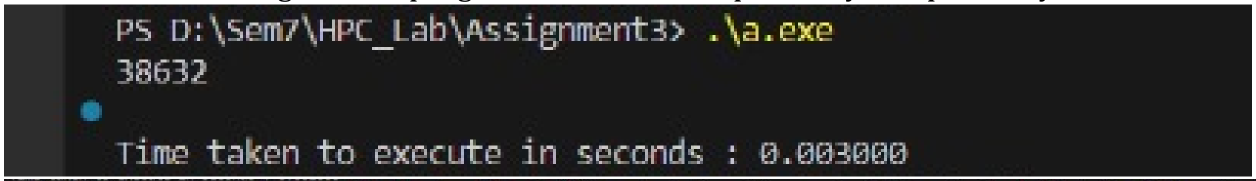
// C Program to find the minimum scalar product of two vectors (dot product)

Screenshots:

```
#include <stdio.h>
#include <algorithm>
#include <iostream>
```

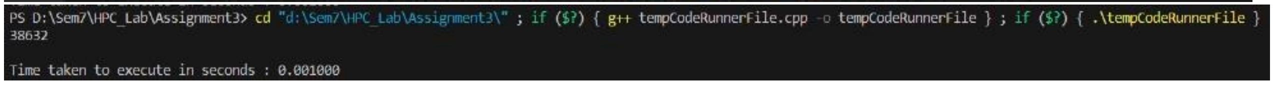
```
#include <omp.h>
#include <time.h>
using namespace std;
int compare(int x, int y)
{
    return x > y;
}
int main()
{
    clock_t start, end;
    start = clock();
    int v1[] = {3, 2, 5, 6, 1, 3, 3, 43, 54, 21, 43, 76, 21,
32, 43, 54, 21, 987, 21, 32, 54, 21, 32, 54, 21, 43, 21, 43,
21, 421, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 2, 6, 0, 1, 9, 43,
12, 5, 6, 2, 6, 0, 1, 9, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 5,
45, 12, 34, 54, 23, 32, 76, 15, 18, 42, 6, 14, 10, 3, 3, 43,
54, 21, 43, 76, 21, 32, 43, 54, 21, 987, 21, 32, 54, 21, 32,
54, 21, 2, 6, 0, 1, 9, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 5,
45, 12, 34, 54, 23, 32, 76, 15, 18, 42, 6, 14, 10, 3, 3, 43,
54, 21, 43, 76, 21, 32, 43, 54, 21, 987, 21, 32, 54, 21, 32,
54, 21, 3, 2, 5, 6, 1, 3, 3, 43, 54, 21, 43, 76, 21, 32, 43,
54, 21, 987, 21, 32, 54, 21, 32, 54, 21, 43, 21, 43, 21, 421,
43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 2, 6, 0, 1, 9, 43, 12, 5,
6};
    int v2[] = {2, 6, 0, 1, 9, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4,
2, 5, 45, 12, 34, 54, 23, 32, 76, 15, 18, 42, 6, 14, 10, 3, 3,
43, 54, 21, 43, 76, 21, 32, 43, 54, 21, 987, 21, 32, 54, 21,
32, 54, 21, 3, 2, 5, 6, 1, 3, 3, 43, 54, 21, 43, 76, 21, 32,
43, 54, 21, 987, 21, 32, 54, 21, 32, 54, 21, 43, 21, 43, 21,
421, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 2, 6, 0, 1, 9, 43, 12,
5, 6, 2, 6, 0, 1, 9, 43, 12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 5, 45,
12, 34, 54, 23, 32, 76, 15, 18, 42, 6, 14, 10, 3, 3, 43, 54,
21, 43, 76, 21, 32, 43, 54, 21, 987, 21, 32, 54, 21, 32, 54,
21, 3, 2, 5, 6, 1, 3, 3, 43, 54, 21, 43, 76, 21, 32, 43, 54,
21, 987, 21, 32, 54, 21, 32, 54, 21, 43, 21, 43, 21, 421, 43,
12, 5, 6, 7, 4, 2, 8, 9, 4, 2, 2, 6, 0, 1, 9, 43, 12, 5, 6};
    int n1 = sizeof(v1) / sizeof(int);
    int n2 = sizeof(v2) / sizeof(int);
    sort(v1, v1 + n1);
    sort(v2, v2 + n2, compare);
    int sum = 0;
#pragma omp parallel for num_threads(8) reduction(+ : sum)
    for (int i = 0; i < n1; i++)
    {
        sum += (v1[i] * v2[i]);
    }
    cout << sum << endl;
    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f",
duration);
    return 0;
}
```

Output:
Execution times we get when program is executed sequentially and parallelly:



```
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe
38632

Time taken to execute in seconds : 0.003000
```



```
PS D:\Sem7\HPC_Lab\Assignment3> cd "d:\Sem7\HPC_Lab\Assignment3\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
38632

Time taken to execute in seconds : 0.001000
```

Information and analysis:

In OpenMP, the **parallel for** construct is used to parallelize the execution of a loop across multiple threads. This construct combines the parallel construct (which creates a parallel region) with the for construct (which specifies a loop to be parallelized).

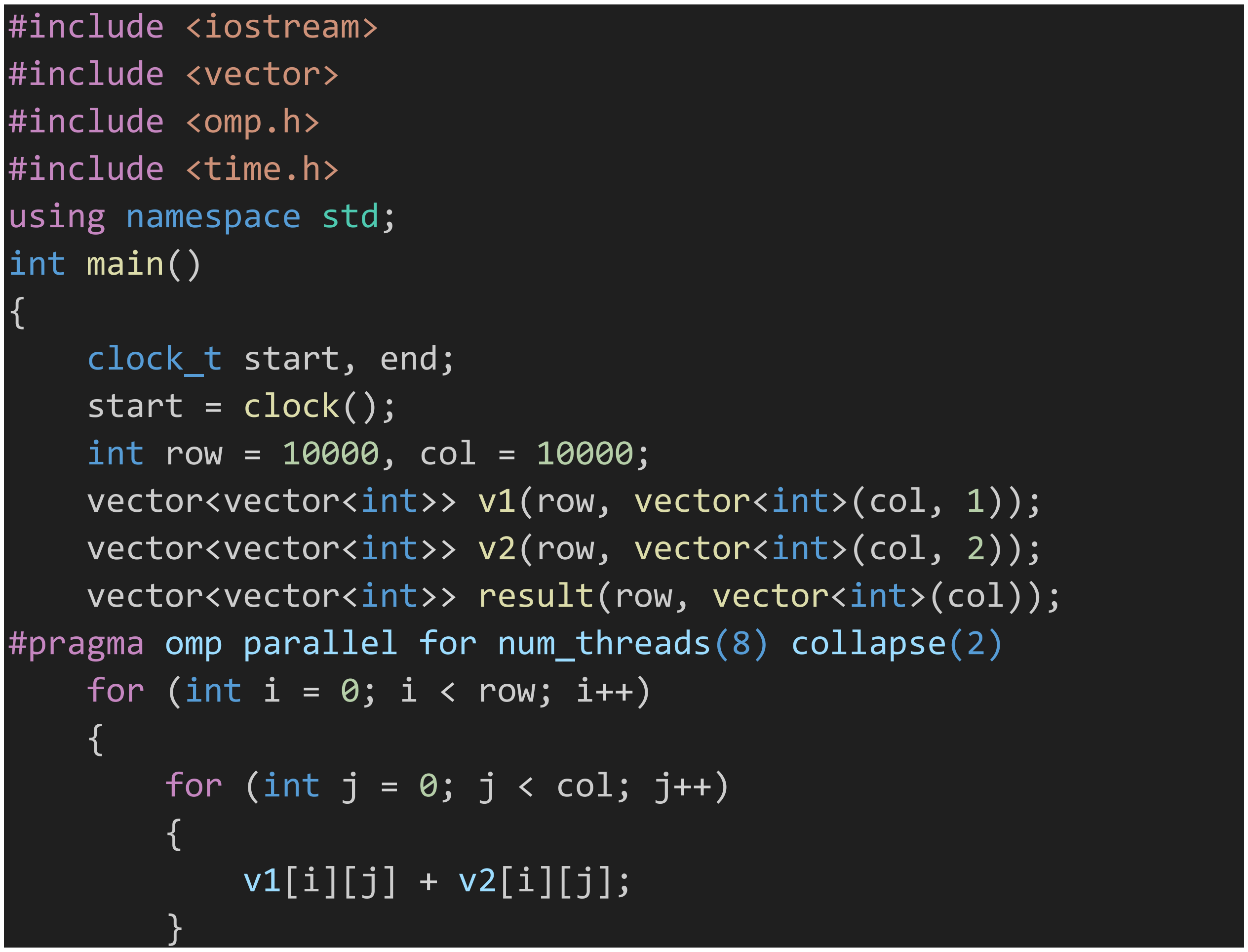
The **reduction** clause in OpenMP is used to perform a reduction operation on a variable across multiple threads in a parallel region. Reductions are commonly used to accumulate results from multiple threads into a single result, such as summing up values or finding the maximum value. The reduction clause simplifies the process of maintaining thread-local copies of the variable and combining the results at the end of the parallel region.

Number of Threads	Data Size	Sequential Time	Parallel Time
4	200	0.00100	0.00300
8	200	0.00100	0.00400
12	200	0.00100	0.00400
16	200	0.00100	0.00100
20	200	0.00100	0.00200

Problem Statement 2:

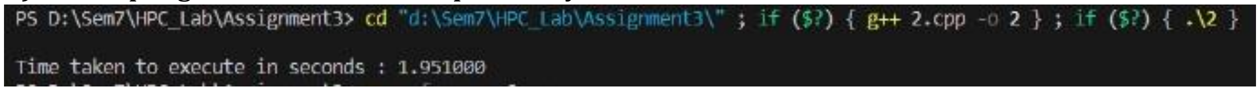
Write OpenMP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C in calculate the execution time or use GPROF)
i. For each matrix size, change the number of threads from 2,4,8., and plot the speedup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

Screenshots:



```
#include <iostream>
#include <vector>
#include <omp.h>
#include <time.h>
using namespace std;
int main()
{
    clock_t start, end;
    start = clock();
    int row = 10000, col = 10000;
    vector<vector<int>> v1(row, vector<int>(col, 1));
    vector<vector<int>> v2(row, vector<int>(col, 2));
    vector<vector<int>> result(row, vector<int>(col));
#pragma omp parallel for num_threads(8) collapse(2)
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            v1[i][j] + v2[i][j];
        }
    }
    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f",
duration);
    return 0;
}
```

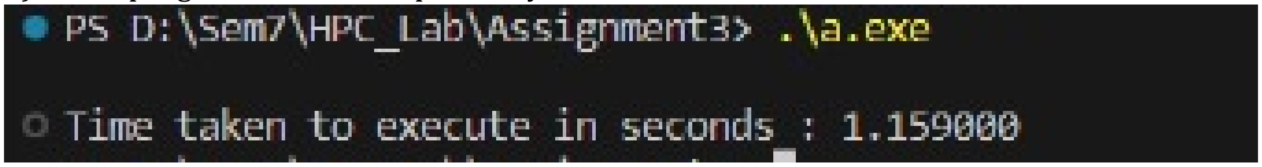
Output:
1) When program is executed sequentially:



```
PS D:\Sem7\HPC_Lab\Assignment3> cd "d:\Sem7\HPC_Lab\Assignment3\" ; if ($?) { g++ 2.cpp -o 2 } ; if ($?) { .\2 }

Time taken to execute in seconds : 1.951000
```

2) When program is executed parallelly:



```
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.159000
```

Information and analysis:
When we use the collapse with the parallel for, the nested loops also get parallelized. So, we get the speedup when we execute the nested loops parallelly and using the collapse clause.

Number of Threads	Data Size	Sequential Time	Parallel Time
4	Rows-10, Cols-10	0.00000	0.00100
8	Rows-10, Cols-10	0.00000	0.00300
4	Rows-100, Cols-100	0.00000	0.00200
8	Rows-100, Cols-100	0.00000	0.00100
4	Rows-1000, Cols-1000	0.00290	0.01600
8	Rows-1000, Cols-1000	0.00290	0.02400
4	Rows-10000, Cols-10000	2.39500	2.45400
8	Rows-10000, Cols-10000	2.39500	1.37700

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:

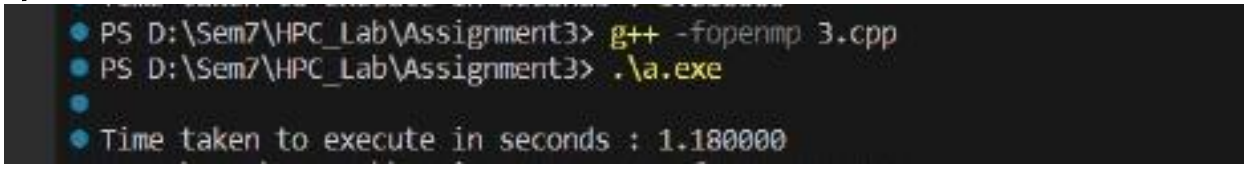
```
#include <iostream>
#include <vector>
#include <omp.h>
#include <time.h>

using namespace std;
int main()
{
    clock_t start, end;
    start = clock();
    int n = 200000000;
    vector<int> v(n, 3);
    int b = 6;
#pragma omp parallel for num_threads(8) schedule(static, 500)
firstprivate(b)
    for (int i = 0; i < n; i++)
    {
        v[i] = v[i] + b;
    }

    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f",
duration);
    return 0;
}
```

Output:
Execution times when chunk_size in scheduling is varied

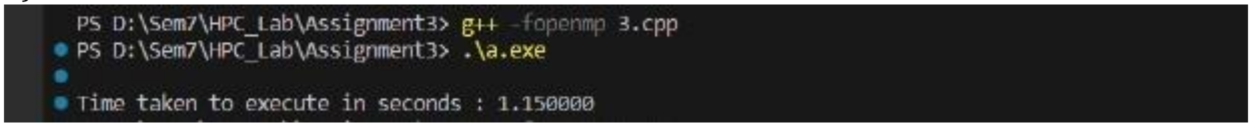
1) When chunk_size is 50



```
PS D:\Sem7\HPC_Lab\Assignment3> g++ -fopenmp 3.cpp
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.180000
```

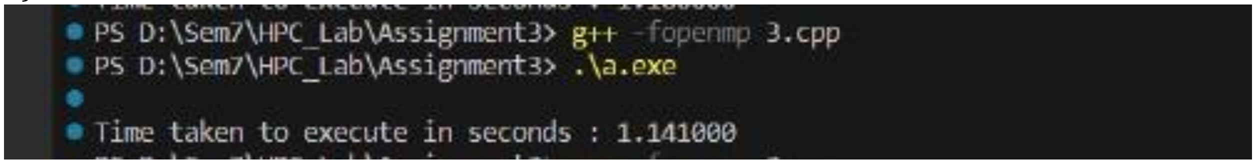
2) When chunk_size is 100



```
PS D:\Sem7\HPC_Lab\Assignment3> g++ -fopenmp 3.cpp
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.150000
```

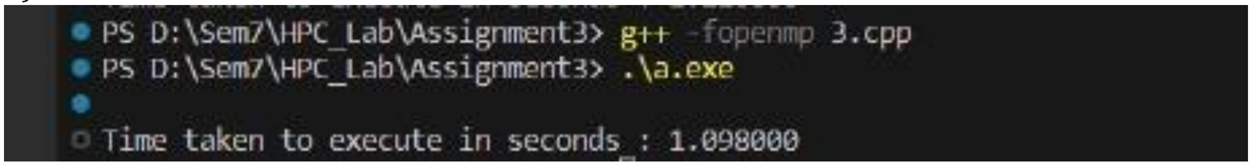
3) When chunk_size is 200



```
PS D:\Sem7\HPC_Lab\Assignment3> g++ -fopenmp 3.cpp
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.141000
```

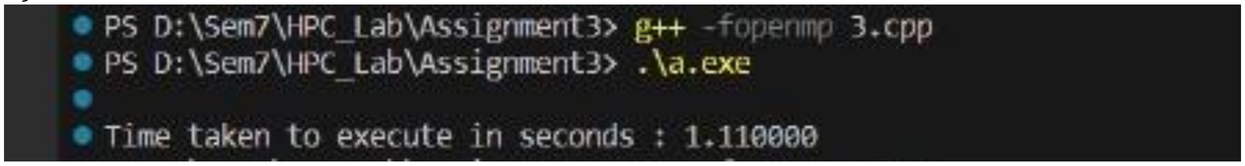
4) When chunk_size is 400



```
PS D:\Sem7\HPC_Lab\Assignment3> g++ -fopenmp 3.cpp
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.098000
```

5) When chunk_size is 500



```
PS D:\Sem7\HPC_Lab\Assignment3> g++ -fopenmp 3.cpp
PS D:\Sem7\HPC_Lab\Assignment3> .\a.exe

Time taken to execute in seconds : 1.110000
```

Information and analysis:
In static scheduling, as we increase the chunk_size of our task from 50 to 400 gradually, we get the execution time better each time but after that as we go for more chunk_size we don’t get much difference and execution time remains constant.

Number of Threads	Chunk Size	Sequential Time	Parallel Time
4	50	2.33600	1.18000
4	100	2.33600	1.15000
4	200	2.33600	1.14100
4	400	2.33600	1.09800
4	500	2.33600	1.11000

Github Link: