

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

```
Enter size of array = 10

8. Thread = 4, Product = 8.000000
9. Thread = 4, Product = 8.000000
2. Thread = 1, Product = 14.000000
3. Thread = 1, Product = 32.000000
0. Thread = 0, Product = 0.000000
1. Thread = 0, Product = 8.000000
4. Thread = 2, Product = 20.000000
5. Thread = 2, Product = 40.000000
6. Thread = 3, Product = 18.000000
7. Thread = 3, Product = 32.000000
Product: 120.000000
Time taken: 6.000000 milliseconds
Time taken: 0.006000 seconds
```

Threads = 8, Array size = 500

```
479. Thread = 7, Product = 2583740.000000
494. Thread = 7, Product = 2586210.000000
495. Thread = 7, Product = 2588190.000000
485. Thread = 2, Product = 2588190.000000
Product: 20708500.000000
Time taken: 51.000000 milliseconds
Time taken: 0.051000 seconds
```

Threads = 8, Array size = 1000

```
981. Thread = 2, Product = 20765908.000000
996. Thread = 2, Product = 20768896.000000
997. Thread = 2, Product = 20770890.000000
Product: 166167000.000000
Time taken: 101.000000 milliseconds
Time taken: 0.101000 seconds
```

Keeping data constant and increasing number of threads.

Threads = 10, Array size = 500

```
451. Thread = 5, Product = 2057234.000000  
470. Thread = 5, Product = 2050924.000000  
471. Thread = 5, Product = 2064112.000000  
490. Thread = 5, Product = 2068522.000000  
491. Thread = 5, Product = 2072450.000000  
Product: 20708500.000000  
Time taken: 48.000000 milliseconds  
Time taken: 0.048000 seconds
```

Threads = 250, Array size = 500

```
355. Thread = 177, Product = 102450.000000  
372. Thread = 186, Product = 47244.000000  
373. Thread = 186, Product = 94242.000000  
Product: 20708500.000000  
Time taken: 72.000000 milliseconds  
Time taken: 0.072000 seconds
```

Threads = 500, Array size = 500

```
160. Thread = 80, Product = 54240.000000  
161. Thread = 80, Product = 108658.000000  
Product: 20708500.000000  
Time taken: 98.000000 milliseconds  
Time taken: 0.098000 seconds
```

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.
 - a. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
 - b. **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 to calculate the execution time or use GPROF)

- For each matrix size, change the number of threads from 2,4,8, and plot the speedup versus the number of threads.
- 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

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe  
Enter dimension for 2D matrix = 250  
  
Parallel Method Time: 0.008000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 300

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe  
Enter dimension for 2D matrix = 300  
  
Parallel Method Time: 0.008000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 350

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe  
Enter dimension for 2D matrix = 350  
  
Parallel Method Time: 0.011000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 415

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe  
Enter dimension for 2D matrix = 415  
  
Parallel Method Time: 0.016000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Threads = 4

Matrix size = 250

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe
Enter dimension for 2D matrix = 250

Parallel Method Time: 0.006000 seconds

C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 300

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe
Enter dimension for 2D matrix = 300

Parallel Method Time: 0.007000 seconds

C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 350

```
Enter dimension for 2D matrix = 350

Parallel Method Time: 0.009000 seconds

C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 415

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>a.exe
Enter dimension for 2D matrix = 415

Parallel Method Time: 0.012000 seconds

C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```


Threads = 8

Matrix size = 250

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>.exe  
Enter dimension for 2D matrix = 250  
  
Parallel Method Time: 0.006000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 300

```
Enter dimension for 2D matrix = 300  
  
Parallel Method Time: 0.007000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 350

```
Enter dimension for 2D matrix = 350  
  
Parallel Method Time: 0.009000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Matrix size = 415

```
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>.exe  
Enter dimension for 2D matrix = 415  
  
Parallel Method Time: 0.013000 seconds  
  
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

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;
}

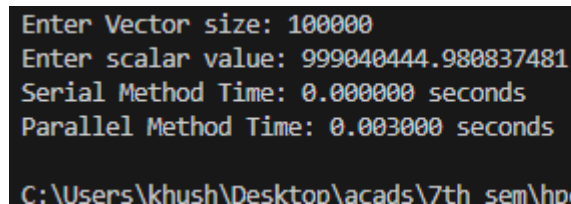
double end_time_parallel = omp_get_wtime();

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

return 0;
}
```

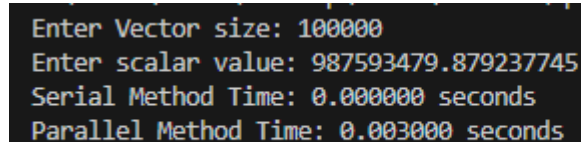
Screenshots:

Threads = 2 Vector Size= 100000



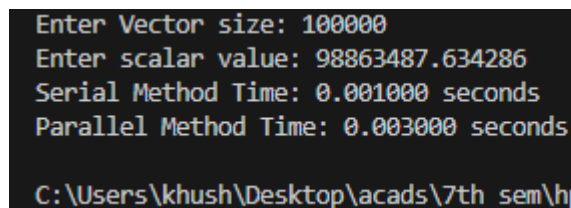
```
Enter Vector size: 100000
Enter scalar value: 999040444.980837481
Serial Method Time: 0.000000 seconds
Parallel Method Time: 0.003000 seconds
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Threads = 4 Vector Size= 100000



```
Enter Vector size: 100000
Enter scalar value: 987593479.879237745
Serial Method Time: 0.000000 seconds
Parallel Method Time: 0.003000 seconds
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

Threads = 8 Vector Size= 100000



```
Enter Vector size: 100000
Enter scalar value: 98863487.634286
Serial Method Time: 0.001000 seconds
Parallel Method Time: 0.003000 seconds
C:\Users\khush\Desktop\acads\7th sem\hpc1\p3>
```

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:

2. **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
3. **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.
 - a. **Static Schedule (schedule(static, chunk)):** Divides iterations into contiguous chunks, distributing them statically among threads. Useful when loop iterations have roughly uniform workload
 - b. **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.