

Class: Final Year B.Tech(Computer Science and Engineering)

Year: 2025-26

Semester: 1

Course: High Performance Computing Lab

PRN : 22510021

Batch : B7

Practical No. 3

Exam Seat No:

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)

Code :

```
#include <iostream>
#include <vector>
#include <algorithm> // for sort
#include <omp.h>      // for OpenMP

using namespace std;

int main() {
    // Example vectors
    vector<int> a = {3, 9, 6};
    vector<int> b = {5, 21, 11};

    int n = a.size();

    // Sort a ascending
    sort(a.begin(), a.end());

    // Sort b descending
    sort(b.begin(), b.end(), greater<int>());
```

```
// Calculate minimum scalar product in parallel
int result = 0;

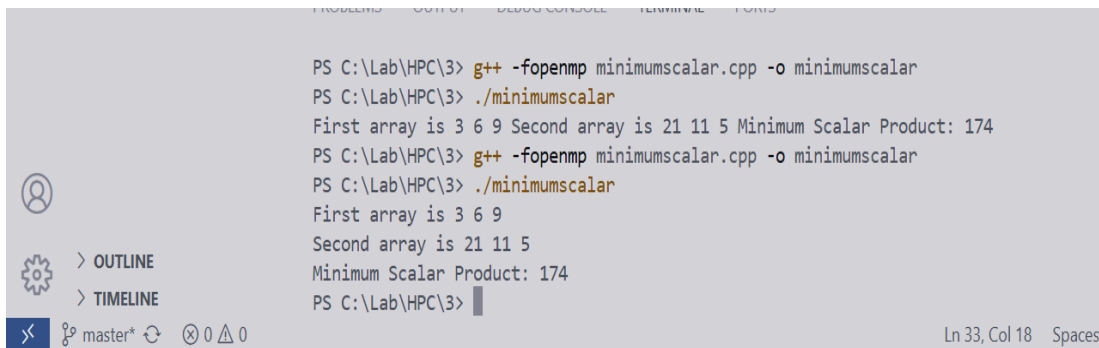
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < n; i++) {
    result += a[i] * b[i];
}

cout << "First array is ";
for(int i: a){
    cout << i << " ";
}
cout << endl;

cout << "Second array is ";
for(int i: b){
    cout << i << " ";
}
cout << endl;
cout << "Minimum Scalar Product: " << result << endl;

return 0;
}
```

Screenshots:



```
PS C:\Lab\HPC\3> g++ -fopenmp minimumscalar.cpp -o minimumscalar
PS C:\Lab\HPC\3> ./minimumscalar
First array is 3 6 9 Second array is 21 11 5 Minimum Scalar Product: 174
PS C:\Lab\HPC\3> g++ -fopenmp minimumscalar.cpp -o minimumscalar
PS C:\Lab\HPC\3> ./minimumscalar
First array is 3 6 9
Second array is 21 11 5
Minimum Scalar Product: 174
PS C:\Lab\HPC\3>
```

Information and analysis:

- This program find the minimum scalar product of 2 arrays.
- First array is sorted in ascending order and second in descending order.
- Then it multiply each element and add them to get the result.
- OpenMP is used to make the adding faster using multiple threads.
- It prints the sorted arrays and the final minimum scalar product.

Observations:

- Sorting in opposite way makes the scalar product minimum.
- Parallel loop makes the calculation faster for big arrays, but for small arrays not much difference.

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)

- 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.

Code: `#include <stdio.h>`

`#include <stdlib.h>`

`#include <omp.h>`

`// Macro for 2D indexing in 1D array`

`#define IDX(i, j, n) ((i) * (n) + (j))`

`// Fill matrices with simple values`

`void fill_matrix(double *A, double *B, int n) {`

`for (int i = 0; i < n; i++) {`

`for (int j = 0; j < n; j++) {`

`A[IDX(i, j, n)] = i + j; // Example: A(i,j) = i+j`

`B[IDX(i, j, n)] = i - j; // Example: B(i,j) = i-j`

`}`

```
    }  
}  
  
// Matrix addition using OpenMP  
void add_matrix(double *A, double *B, double *C, int n) {  
    #pragma omp parallel for collapse(2)  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            C[IDX(i, j, n)] = A[IDX(i, j, n)] + B[IDX(i, j, n)];  
        }  
    }  
}  
  
int main() {  
    // Different matrix sizes  
    int sizes[] = {250, 500, 750, 1000, 2000};  
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);  
  
    // Different thread counts  
    int threads[] = {1, 2, 4, 8};  
    int num_threads = sizeof(threads) / sizeof(threads[0]);  
  
    for (int s = 0; s < num_sizes; s++) {  
        int n = sizes[s];  
        printf("\nMatrix Size: %d x %d\n", n, n);  
  
        // Allocate matrices  
        double *A = (double*) malloc(n * n * sizeof(double));  
        double *B = (double*) malloc(n * n * sizeof(double));  
        double *C = (double*) malloc(n * n * sizeof(double));  
  
        if (!A || !B || !C) {  
            printf("Memory allocation failed for %d\n", n);  
            return 1;  
        }  
  
        // Fill input matrices  
        fill_matrix(A, B, n);  
  
        double base_time = 0.0;
```

```
// Run with different thread counts
for (int t = 0; t < num_threads; t++) {
    omp_set_num_threads(threads[t]);

    double start = omp_get_wtime();
    add_matrix(A, B, C, n);
    double end = omp_get_wtime();

    double elapsed = end - start;
    if (threads[t] == 1) base_time = elapsed;

    double speedup = base_time / elapsed;

    printf("Threads: %d, Time: %.6f sec, Speedup: %.2f\n",
           threads[t], elapsed, speedup);
}

free(A);
free(B);
free(C);
}

return 0;
}
```

Screenshots:

```
Matrix Size: 250 x 250
Threads: 1, Time: 0.000000 sec, Speedup: nan
Threads: 2, Time: 0.004000 sec, Speedup: 0.00
Threads: 4, Time: 0.000000 sec, Speedup: nan
Threads: 8, Time: 0.000000 sec, Speedup: nan

Matrix Size: 500 x 500
Threads: 1, Time: 0.000000 sec, Speedup: nan
Threads: 2, Time: 0.001000 sec, Speedup: 0.00
Threads: 4, Time: 0.000000 sec, Speedup: nan
Threads: 8, Time: 0.000000 sec, Speedup: nan

Matrix Size: 750 x 750
Threads: 1, Time: 0.002000 sec, Speedup: 1.00
Threads: 2, Time: 0.007000 sec, Speedup: 0.29
Threads: 4, Time: 0.001000 sec, Speedup: 2.00
Threads: 8, Time: 0.000000 sec, Speedup: inf

Matrix Size: 1000 x 1000
Threads: 1, Time: 0.000000 sec, Speedup: nan
Threads: 2, Time: 0.016000 sec, Speedup: 0.00
Threads: 4, Time: 0.005000 sec, Speedup: 0.00
Threads: 8, Time: 0.005000 sec, Speedup: 0.00

Matrix Size: 2000 x 2000
Threads: 1, Time: 0.024000 sec, Speedup: 1.00
Threads: 2, Time: 0.000000 sec, Speedup: inf
Threads: 4, Time: 0.015000 sec, Speedup: 1.60
Threads: 8, Time: 0.014000 sec, Speedup: 1.71
```

Information and analysis:

- For smaller matrices (e.g., 250×250), execution is very fast, so times may appear close to zero.
- With larger matrices (e.g., 2000×2000), parallel execution shows clear improvement over single-thread.
- Increasing thread count reduces time, giving higher speedup, but improvement is not perfectly linear because of memory bandwidth and thread overhead.
- Speedup is most noticeable at larger sizes, where computation dominates.
- Using collapse(2) distributes the nested loops across threads more evenly.

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.

Code:

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

#define SIZE 200
#define SCALAR 5

int main() {

    omp_set_num_threads(8);

    double A[SIZE];
    int i;
    double start, end;

    // initialize array
    for (i = 0; i < SIZE; i++) {
        A[i] = i;
    }

    // ----- static schedule -----
    start = omp_get_wtime();
    #pragma omp parallel for schedule(static, 10)
    for (i = 0; i < SIZE; i++) {
        A[i] = A[i] + SCALAR;
    }
    end = omp_get_wtime();
    printf("Static schedule (chunk=10) time: %f sec\n", end - start);

    // reset
    for (i = 0; i < SIZE; i++) {
        A[i] = i;
    }

    // ----- dynamic schedule -----
```

```
start = omp_get_wtime();
#pragma omp parallel for schedule(dynamic, 10)
for (i = 0; i < SIZE; i++) {
    A[i] = A[i] + SCALAR;
}
end = omp_get_wtime();
printf("Dynamic schedule (chunk=10) time: %f sec\n", end - start);

// reset
for (i = 0; i < SIZE; i++) {
    A[i] = i;
}

// ----- nowait example -----
start = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < SIZE; i++) {
        A[i] = A[i] + SCALAR;
    }

    #pragma omp single
    {
        printf("nowait function\n");
    }
}
end = omp_get_wtime();
printf("Nowait example time: %f sec\n", end - start);

return 0;
}
```

Screenshots:


```
collect2.exe: error: ld returned 1 exit status
PS C:\Lab\HPC\3> gcc -fopenmp 3chunksize.c -o chunksize
PS C:\Lab\HPC\3> .\chunksize
PS C:\Lab\HPC\3> gcc -fopenmp 3chunksize.c -o chunksize
PS C:\Lab\HPC\3> .\chunksize
PS C:\Lab\HPC\3> gcc -fopenmp 3chunksize.c -o chunksize
PS C:\Lab\HPC\3> .\chunksize
Static schedule (chunk=10) time: 0.000000 sec
Dynamic schedule (chunk=10) time: 0.000000 sec
nowait function
Nowait example time: 0.000000 sec
PS C:\Lab\HPC\3>
```

Information and analysis:

- For small array sizes, times showed 0.000000 sec because the work was too small; execution finished in microseconds.
- With large sizes (e.g., 100 million), measurable times appeared (~0.06 sec sequential vs ~0.04 sec parallel).
- Parallel was faster but speedup was limited since memory access, not computation, is the main bottleneck.
- Chunk size affects overhead and load balance: smaller chunks give balance but more overhead, larger chunks reduce overhead.
- `nowait` removes the barrier after a loop, so threads don't wait for others before moving ahead.

Github Link: <https://github.com/22510021-Shrikrishna/HPC.git>