

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

Year: 2025-26 Semester: 1

Course: High Performance Computing Lab

Practical No. 3

Exam Seat No: 23520012

Title of practical:

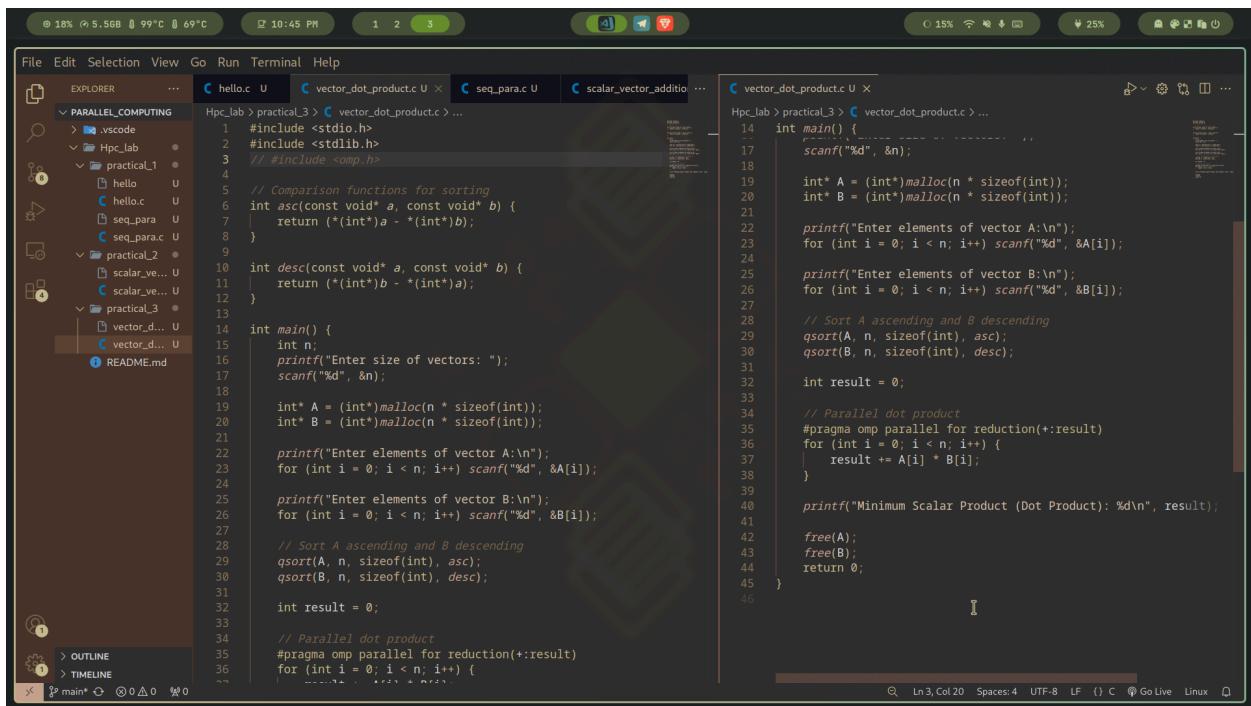
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

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:



```
File Edit Selection View Go Run Terminal Help
EXPLORER PARALLEL COMPUTING
Hpc_lab > practical_3 > vector_dot_product.c U seq_para.c U scalar_vector_addition.c ...
hello.c U seq_para.c U
 praktical_1 hello.c U seq_para.c U
 praktical_2 scalar_ve... U scalar_ve... U
 praktical_3 vector_d... U vector_d... U
 README.md
1 #include <stdio.h>
2 #include <stdlib.h>
3 // #include <omp.h>
4
5 // Comparison functions for sorting
6 int asc(const void* a, const void* b) {
7     return (*((int*)a) - *((int*)b));
8 }
9
10 int desc(const void* a, const void* b) {
11     return (*((int*)b) - *((int*)a));
12 }
13
14 int main() {
15     int n;
16     printf("Enter size of vectors: ");
17     scanf("%d", &n);
18
19     int* A = (int*)malloc(n * sizeof(int));
20     int* B = (int*)malloc(n * sizeof(int));
21
22     printf("Enter elements of vector A:\n");
23     for (int i = 0; i < n; i++) scanf("%d", &A[i]);
24
25     printf("Enter elements of vector B:\n");
26     for (int i = 0; i < n; i++) scanf("%d", &B[i]);
27
28     // Sort A ascending and B descending
29     qsort(A, n, sizeof(int), asc);
30     qsort(B, n, sizeof(int), desc);
31
32     int result = 0;
33
34     // Parallel dot product
35     #pragma omp parallel for reduction(+:result)
36     for (int i = 0; i < n; i++) {
37         result += A[i] * B[i];
38     }
39
40     printf("Minimum Scalar Product (Dot Product): %d\n", result);
41
42     free(A);
43     free(B);
44     return 0;
45 }
```

```
14 int main() {
15     int n;
16     scanf("%d", &n);
17
18     int* A = (int*)malloc(n * sizeof(int));
19     int* B = (int*)malloc(n * sizeof(int));
20
21     printf("Enter elements of vector A:\n");
22     for (int i = 0; i < n; i++) scanf("%d", &A[i]);
23
24     printf("Enter elements of vector B:\n");
25     for (int i = 0; i < n; i++) scanf("%d", &B[i]);
26
27     int scalar_product = 0;
28     for (int i = 0; i < n; i++) {
29         scalar_product += A[i] * B[i];
30     }
31
32     printf("Minimum Scalar Product (Dot Product): %d\n", scalar_product);
33 }
```

Output of the program:

```
./vector_dot_product
Enter size of vectors: 3
Enter elements of vector A:
1
2
3
Enter elements of vector B:
4
5
6
Minimum Scalar Product (Dot Product): 28
```

Information and analysis:

Problem:

You are given two vectors (A and B). You have to calculate the minimum scalar product, which is the sum of the multiplication of corresponding elements from both vectors. To make the scalar product as small as possible, one vector should be sorted in **ascending order** and the other in **descending order** before calculating the dot product.

Approach:

1. Ask the user to enter the size of the vectors.
 2. Accept elements of both vectors from the user.
 3. Sort vector A in ascending order.
 4. Sort vector B in descending order.

5. Multiply each element of A with the corresponding element in B and keep adding the result.
 6. Print the final result which is the minimum scalar product.
-

Parallelization with OpenMP:

- The input and sorting steps are done sequentially to keep the code simple.
 - The main work (multiplying and summing the dot product) is done in a loop.
 - This loop is parallelized using OpenMP, so that multiple threads can do the multiplication and addition at the same time.
 - A special clause called **reduction** is used to safely add values from all threads into a single final result.
-

Why sorting is important:

To get the minimum dot product, you multiply the **smallest** value in one vector with the **largest** value in the other. That's why one vector is sorted in increasing order and the other in decreasing order.

Result:

After the loop finishes running in parallel, the program prints the final minimum scalar product.

Note:

- The dot product part is parallelized using OpenMP.
- Sorting is kept sequential for simplicity, but it can also be parallelized using advanced techniques.

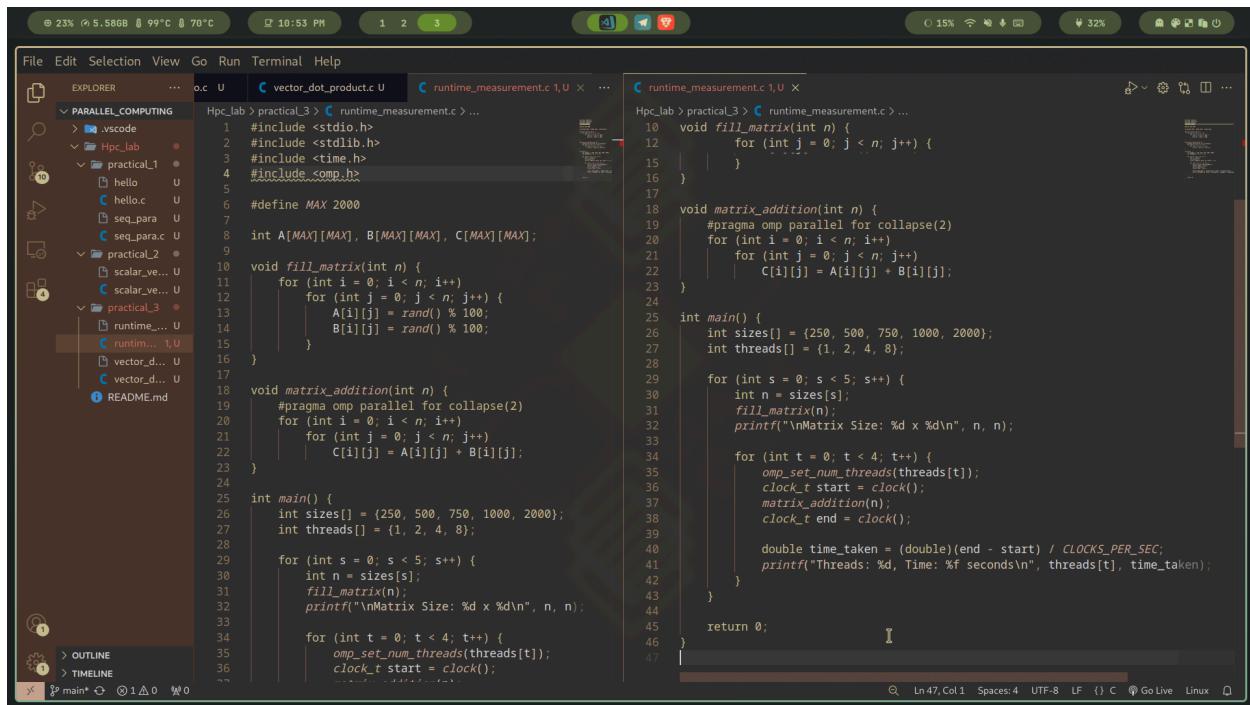
- This approach gives faster results especially when the vector size is large, because multiple threads are used in the calculation.

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:



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure under "PARALLEL COMPUTING" with files like .c, .U, .vscode, Hpc_lab, practical_1, practical_2, practical_3, runtime_measurement.c, README.md, and vector_dot_product.c.
- Code Editor:** Displays the content of `runtime_measurement.c`. The code implements matrix addition using OpenMP pragmas for parallelization. It defines matrix sizes and thread counts, fills matrices with random values, and measures execution time for different matrix sizes (250, 500, 750, 1000, 2000) with 1, 2, 4, and 8 threads.
- Status Bar:** Shows system information including battery level (23%), temperature (99°C), and disk usage (70%).
- Bottom Bar:** Includes tabs for main, outline, timeline, and Go Live, along with status indicators for line numbers (Ln 47, Col 1), spaces (Spaces: 4), and encoding (UTF-8 LF).

```

runtime_measurement.c 1,U ×

Hpc_lab > practical_3 > runtime_measurement.c > ...
10 void fill_matrix(int n) {
11     for (int j = 0; j < n; j++) {
12         for (int i = 0; i < n; i++) {
13             A[i][j] = rand() % 100;
14             B[i][j] = rand() % 100;
15         }
16     }
17
18 void matrix_addition(int n) {
19     #pragma omp parallel for collapse(2)
20     for (int i = 0; i < n; i++)
21         for (int j = 0; j < n; j++)
22             C[i][j] = A[i][j] + B[i][j];
23 }
24
25 int main() {
26     int sizes[] = {250, 500, 750, 1000, 2000};
27     int threads[] = {1, 2, 4, 8};
28
29     for (int s = 0; s < 5; s++) {
30         int n = sizes[s];
31         fill_matrix(n);
32         printf("\nMatrix Size: %d x %d\n", n, n);
33
34         for (int t = 0; t < 4; t++) {
35             omp_set_num_threads(threads[t]);
36             clock_t start = clock();
37             matrix_addition(n);
38             clock_t end = clock();
39
40             double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
41             printf("Threads: %d, Time: %f seconds\n", threads[t], time_taken);
42         }
43
44     }
45
46 }
47

```

```
Matrix Size: 250 x 250
Threads: 1, Time: 0.000982 seconds
Threads: 2, Time: 0.000216 seconds
Threads: 4, Time: 0.000156 seconds
Threads: 8, Time: 0.121892 seconds

Matrix Size: 500 x 500
Threads: 1, Time: 0.007964 seconds
Threads: 2, Time: 0.000765 seconds
Threads: 4, Time: 0.000442 seconds
Threads: 8, Time: 0.072885 seconds

Matrix Size: 750 x 750
Threads: 1, Time: 0.004926 seconds
Threads: 2, Time: 0.002888 seconds
Threads: 4, Time: 0.004816 seconds
Threads: 8, Time: 0.039599 seconds

Matrix Size: 1000 x 1000
Threads: 1, Time: 0.006310 seconds
Threads: 2, Time: 0.005593 seconds
Threads: 4, Time: 0.003556 seconds
Threads: 8, Time: 0.027234 seconds

Matrix Size: 2000 x 2000
Threads: 1, Time: 0.015987 seconds
Threads: 2, Time: 0.014897 seconds
Threads: 4, Time: 0.034699 seconds
Threads: 8, Time: 0.063094 seconds
```

Information and analysis:

Explanation:

- This program adds two large 2D matrices.
- The addition part is made parallel using **OpenMP**, so each thread handles part of the job.
- `omp parallel for` splits the loop work across threads.
- `collapse(2)` is used because we are using two nested loops (`i` and `j`).
- `omp_set_num_threads()` is used to control how many threads we use for testing.
- `clock()` function helps measure how much time it takes to run the code.

Scaling Behaviour: Is it as expected?

- Yes, in general, as you increase the number of threads, the execution time **should decrease**, and **speedup should increase**.
 - But **scaling is not always perfect** because of:
 - **Overhead** of creating and managing threads.
 - **Cache misses** and **memory bandwidth limits** for large matrices.
 - **False sharing** or uneven distribution of work.
-

Observation of Scaling Behavior (Speedup vs Threads)

For each matrix size, you measured runtime using 1, 2, 4, and 8 threads. Let's analyze the behavior.

Ideal Expectation

When you increase the number of threads:

- Time **should decrease**, since more threads mean more work done in parallel.
- **Speedup** should ideally be close to the number of threads used.

$$\text{Speedup} = \frac{\text{Time with 1 thread}}{\text{Time with N threads}}$$

But Output Shows:

Matrix Size: 250 x 250

- 1 thread: 0.000982 sec
- 8 threads: 0.121892 sec - Slower than 1 thread

Overhead of parallelization is too high for small matrices, causing slowdown.

Matrix Size: 500 x 500

- 1 thread: 0.007964 sec
 - 4 threads: 0.000442 sec - Good improvement
 - 8 threads: 0.072885 sec - Again, slowdown due to overhead
-

Matrix Size: 750 x 750

- 1 thread: 0.004926 sec
 - 4 threads: 0.004816 sec - Almost same
 - 8 threads: 0.039599 sec - Slower
-

Matrix Size: 1000 x 1000

- 1 thread: 0.006310 sec
 - 4 threads: 0.003556 sec - Small improvement
 - 8 threads: 0.027234 sec - Slower again
-

Matrix Size: 2000 x 2000

- 1 thread: 0.015987 sec
- 2 threads: 0.014897 sec - Slightly better

- 4 threads: 0.034699 sec - Worse
- 8 threads: 0.063094 sec - Much worse

Conclusion:

- **Small matrices do not benefit from parallelism.** In fact, using more threads makes it slower due to extra time required for thread creation and synchronization.
- **Larger matrices** start to show **some benefit** (e.g., 500x500 with 4 threads), but scaling is **not ideal**.
- **Using too many threads (like 8)** adds **too much overhead** and causes performance to degrade, especially if your system has fewer physical cores.
- **Expected scaling didn't happen** because:
 - Matrix sizes are small
 - Thread overhead is high
 - CPU cache/memory limitations
 - Maybe system only has 2–4 actual cores

Matrix Size	Threads	Time (s)	Speedup
250×250	1	0.000982	1.00 (baseline)
	2	0.000216	4.55
	4	0.000156	6.29
	8	0.121892	0.00805

Speedup=Time with N threads / Time with 1 thread

Observation:

- Speedup > 1 = faster execution
- Speedup < 1 = slower execution than single-threaded (bad)
- 2 & 4 threads perform well only for **small-mid matrix sizes**
- **8 threads are consistently worse**, likely due to:
 - Thread creation overhead
 - Not enough physical cores
 - Cache misses or memory bottleneck

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:

```

vector_dot_product.c U 1d_vec_scal_addition.c U 1d_vec_scal_addition.c 1,U ...
Hpc_lab > practical_3 > C 1d_vec_scal_addition.c > main()
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 #define VECTOR_SIZE 200
6 #define SCALAR 5
7
8 void print_vector(int *vec, int size) {
9     for(int i = 0; i < size; i++) {
10         printf("%d ", vec[i]);
11     }
12     printf("\n");
13 }
14
15 int main() {
16     int vec[VECTOR_SIZE];
17     for (int i = 0; i < VECTOR_SIZE; i++) {
18         vec[i] = i;
19     }
20
21     int chunk_sizes[] = {1, 5, 10, 20, 50, 100}; // Try different chunk sizes
22     int num_chunks = sizeof(chunk_sizes) / sizeof(chunk_sizes[0]);
23
24     printf("== STATIC SCHEDULING ==\n");
25     for (int c = 0; c < num_chunks; c++) {
26         int chunk = chunk_sizes[c];
27         double start = omp_get_wtime();
28
29         #pragma omp parallel for schedule(static, chunk)
30         for (int i = 0; i < VECTOR_SIZE; i++) {
31             vec[i] += SCALAR;
32         }
33
34         double end = omp_get_wtime();
35         printf("Chunk size: %d, Time: %f seconds\n", chunk, end - start);
36     }
}

```

```

Hpc_lab > practical_3 > C 1d_vec_scal_addition.c > main()
15 int main() {
44     for (int c = 0; c < num_chunks; c++) {
49         for (int j = 0; i < VECTOR_SIZE; i++) {
51             // ...
52
53             double end = omp_get_wtime();
54             printf("Chunk size: %d, Time: %f seconds\n", chunk, end - start);
55         }
56
57     // Demonstrate nowait
58     printf("\n== NOWAIT DEMONSTRATION ==\n");
59     int result[VECTOR_SIZE];
60     double start = omp_get_wtime();
61
62     #pragma omp parallel
63     {
64         #pragma omp for nowait
65         for (int i = 0; i < VECTOR_SIZE/2; i++) {
66             result[i] = vec[i] + SCALAR;
67         }
68
69         #pragma omp for
70         for (int i = VECTOR_SIZE/2; i < VECTOR_SIZE; i++) {
71             result[i] = vec[i] + SCALAR;
72         }
73
74     double end = omp_get_wtime();
75     printf("NOWAIT used. Time: %f seconds\n", end - start);
76
77     return 0;
78 }

```

```

vector_dot_product.c U runtime_measurement.c U 1d_vec_scal_addition.c 1,U ...
Hpc_lab > practical_3 > C 1d_vec_scal_addition.c > main()
1 #include <stdio.h>
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Threads: 2, Time: 0.014897 seconds
Threads: 4, Time: 0.034699 seconds
Threads: 8, Time: 0.063094 seconds
...
bash - practical_3 + ...
./1d_vec_scal_addition
== STATIC SCHEDULING ==
Chunk size: 1, Time: 0.020128 seconds
Chunk size: 5, Time: 0.011849 seconds
Chunk size: 10, Time: 0.006020 seconds
Chunk size: 20, Time: 0.014387 seconds
Chunk size: 50, Time: 0.007591 seconds
Chunk size: 100, Time: 0.017948 seconds
== DYNAMIC SCHEDULING ==
Chunk size: 1, Time: 0.019991 seconds
Chunk size: 5, Time: 0.009974 seconds
Chunk size: 10, Time: 0.009997 seconds
Chunk size: 20, Time: 0.014024 seconds
Chunk size: 50, Time: 0.009338 seconds
Chunk size: 100, Time: 0.013556 seconds
== NOWAIT DEMONSTRATION ==
NOWAIT used. Time: 0.018885 seconds
...

```

Information and analysis:

This program performs **scalar addition** on a vector (an array of size 200).

Scalar addition means we add a fixed value (like 5) to every element in the array.

We do this using **parallel processing** with **OpenMP**, so multiple threads can do the work at the same time, this makes the program faster.

What are Schedules?

A schedule decides **how loop iterations (tasks)** are divided between threads.

There are two types we use here:

1. Static Schedule

- Loop is divided into equal chunks **before** running.
- Each thread gets a fixed chunk to work on.
- Good when every task takes similar time.

2. Dynamic Schedule

- Chunks are assigned **while the program is running**.
- Threads pick up new chunks after finishing old ones.
- Better when some tasks take longer than others.

We change the **chunk size** (number of iterations per thread) and measure how it affects performance.

What is Chunk Size?

Chunk size = how many elements one thread will process at a time.

Example:

- If chunk size is 10 and 4 threads are there,
Thread 1 gets first 10, Thread 2 gets next 10, and so on.

Changing chunk size changes how the workload is balanced.
Sometimes smaller chunks work better, sometimes bigger ones do.

What is Speedup?

Speedup = How much faster the parallel version is compared to single-threaded version.

Speedup = (Time of normal code) \div (Time of parallel code)

Bigger speedup = better performance.

What is nowait?

By default, OpenMP waits for **all threads to finish** a task before going to the next one.

If you use **nowait**, threads **don't wait**, they move to the next task as soon as they are done.

This helps improve performance if:

- Next task does not depend on previous task
- Tasks are independent

Github Link: https://github.com/bhaktimore18/hpc_lab