# Practical 5

PRN: 23520005
Name: Manas Indrapal Gedam
Batch: B3

---

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

int main() {
    int n = 100;
    int i, j, k;

    // 'static' prevents stack overflo (int)100 e arrays
    static double A[100][100], B[100][100], C[100][100];
    static double x[100], y[100];
    double scalar = 2.5;

    // 1
    printf("1. Running Matrix-Matrix Multiplication...\n");

    #pragma omp parallel for private(i, j, k) // Each thread takes a set of rows
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    printf("   ...Done.\n\n");

    // 2
    printf("2. Running Matrix-Scalar Multiplication...\n");
```

```c
#pragma omp parallel for private(i, j)
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        B[i][j] = scalar * A[i][j];
    }
}
printf("   ...Done.\n\n");

// 3
printf("3. Running Matrix-Vector Multiplication...\n");

#pragma omp parallel for private(i, j) // Each thread takes a set of rows
for (i = 0; i < n; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
        y[i] += A[i][j] * x[j];
    }
}
printf("   ...Done.\n\n");

/    4
printf("4. Running Parallel Prefix Sum...\n");
int size = 10000;
int* in_arr = (int*)malloc(size * sizeof(int));
int* out_arr = (int*)malloc(size * sizeof(int));

for(i = 0; i < size; i++) in_arr[i] = 1; // Initialize array
```

```c
for (int d = 1; d < size; d *= 2) {
    #pragma omp parallel for private(i)
    for (i = d; i < size; i++) {
        out_arr[i] = in_arr[i] + in_arr[i - d];
    }
    #pragma omp parallel for private(i)
    for (i = d; i < size; i++) {
        in_arr[i] = out_arr[i];
    }
}
printf("   ...Done.\n\n");

free(in_arr);
free(out_arr);

return 0;
```

```
● manas@Manass-MacBook-Air HPCL 5 % gcc-15 -fopenmp Q.c -o Q
● manas@Manass-MacBook-Air HPCL 5 % ./Q
1. Running Matrix-Matrix Multiplication...
   ...Done.

2. Running Matrix-Scalar Multiplication...
   ...Done.

3. Running Matrix-Vector Multiplication...
   ...Done.

4. Running Parallel Prefix Sum...
   ...Done.

○ manas@Manass-MacBook-Air HPCL 5 %
```

Problem 1:
**Analysis:** The core task is to calculate each element `C[i][j]` in the output matrix. The key insight is that the calculation for any single element is **completely independent** of the calculation for any other element. For example, computing `C[0][0]` has no effect on the value of `C[3][5]`.

Problem 2:
**Analysis:** Each element in the new matrix is calculated as `B[i][j] = scalar * A[i][j]`. Just like in the previous problem, every calculation is **100% independent** of all others. The operation on `B[0][0]` doesn't affect `B[0][1]` or any other element.

Problem 3:
**Analysis:** The goal is to compute each element `y[i]` of the resulting vector. Each `y[i]` is the dot product of a single row from matrix **A** and the vector **x**. The calculation of `y[0]` is **independent** of the calculation of `y[1]`, and so on.

**Parallel Strategy:** The outer loop over the rows (`i`) is parallelized. This means each thread is responsible for calculating one or more elements of the output vector `y`. Since each thread writes to a different location in `y` (thread 0 might write to `y[0]...y[24]`, while thread 1 writes to `y[25]...y[49]`), there are no **race conditions** or conflicts.

Problem 4:
**Analysis:** A simple, sequential prefix sum has a **loop-carried dependency**. To calculate `output[i]`, you *must* know the result of `output[i-1]`. This makes a naive `for` loop impossible to parallelize directly, as each iteration depends on the one before it.

**Parallel Strategy:** To solve this, a more advanced parallel algorithm is needed. The code uses an algorithm that works in `log(n)` steps. In each step, it performs calculations that *can* be done in parallel (e.g., adding elements that are a certain distance `d` apart). The outer loop manages these sequential steps, while the inner `for` loop within each step is parallelized with OpenMP. This approach has more overhead (synchronization between steps, data copying) but is much faster than the sequential method for large arrays.