

HPC

lab 1

```
gcc -fopenmp hello.c  
./a.out  
esport OMP_NUM_THREADS=7
```

The objective of this lab is to learn how to run **multiple threads** on a **shared-memory** machine and see how OpenMP splits work.

→ **OpenMP** is a library/API for **parallel programming on one machine** (shared memory).

Task 1:

- We implemented a basic OpenMP program to demonstrate parallel execution on a shared-memory system.
- Using the directive `#pragma omp parallel`, the program creates multiple threads inside a single process. Each thread executes the same code concurrently.
- The function `omp_get_thread_num()` is used to identify which thread is running. Each thread prints its own ID, showing that multiple threads are active at the same time.
- The parallel region is executed repeatedly using a loop in order to clearly observe thread behavior. The order of the output changes between runs, which demonstrates the non-deterministic nature of parallel execution.
- The number of threads is controlled using the environment variable `OMP_NUM_THREADS`. This confirms that OpenMP enables parallelism through threads rather than multiple processes.

```
#include <stdio.h>  
#include <omp.h>  
  
int main() {
```

```

#pragma omp parallel
{
    int tid = omp_get_thread_num();

    for (long long i = 0; i < 1000000000LL; i++) {
        if (i % 200000000 == 0) {
            printf("Thread %d at iteration %lld\\n", tid,
i);
        }
    }

    printf("Thread %d finished\\n", tid);
}

return 0;
}

```

Task 2: Parallel vector computation

```

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

int main() {
    const long N = 100000000; // 100 million
    double *A = (double*)malloc(N * sizeof(double));
    double *B = (double*)malloc(N * sizeof(double));
    double *C = (double*)malloc(N * sizeof(double));
    if (!A || !B || !C) {
        printf("Memory allocation failed!\\n");
        return 1;
    }

    // Initialize arrays

```

```

for (long i = 0; i < N; i++) {
    A[i] = (double)i;
    B[i] = (double)(N - i);
}

// ----- SERIAL -----
double t1 = omp_get_wtime();
for (long i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
double t2 = omp_get_wtime();
double serial_time = t2 - t1;

// ----- PARALLEL (MANUAL SPLIT) -----
-
double t3 = omp_get_wtime();

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    long chunk = N / nthreads;
    long start = tid * chunk;
    long end = (tid == nthreads - 1) ? N : start + chunk;
    // last thread takes remainder

    for (long i = start; i < end; i++) {
        C[i] = A[i] + B[i];
    }
}

double t4 = omp_get_wtime();
double parallel_time = t4 - t3;

printf("N = %ld\n", N);
printf("Serial time: %f seconds\n", serial_time);
printf("Parallel time: %f seconds\n", parallel_time);

```

```

printf("Speedup:      %fx\n", serial_time / parallel_time);
printf("Check: C[0]=%f, C[N-1]=%f\n", C[0], C[N-1]);

free(A);
free(B);
free(C);
return 0;
}

```

- In this experiment, vector addition was executed using different numbers of OpenMP threads. The work was manually divided among threads using

`#pragma omp parallel`.

The correctness check confirms that the results are correct for all runs.

<pre> root@Youa-Laptop:~# export OMP_NUM_THREADS=1 root@Youa-Laptop:~# ./a.out N = 100000000 Serial time: 1.487618 seconds Parallel time: 0.693912 seconds Speedup: 2.143814x Check: C[0]=100000000.000000, C[N-1]=100000000.000000 </pre>	<pre> root@Youa-Laptop:~# export OMP_NUM_THREADS=2 root@Youa-Laptop:~# ./a.out N = 100000000 Serial time: 1.619473 seconds Parallel time: 0.386560 seconds Speedup: 4.189445x Check: C[0]=100000000.000000, C[N-1]=100000000.000000 </pre>
--	--

<pre> root@Youa-Laptop:~# export OMP_NUM_THREADS=4 root@Youa-Laptop:~# ./a.out N = 100000000 Serial time: 1.594763 seconds Parallel time: 0.214382 seconds Speedup: 7.438892x Check: C[0]=100000000.000000, C[N-1]=100000000.000000 </pre>	<pre> root@Youa-Laptop:~# export OMP_NUM_THREADS=7 root@Youa-Laptop:~# ./a.out N = 100000000 Serial time: 2.720208 seconds Parallel time: 0.185859 seconds Speedup: 14.635832x Check: C[0]=100000000.000000, C[N-1]=100000000.000000 </pre>
--	---

Note 1:

Using `omp parallel` allows manual control over how work is divided among threads, which can be more efficient than `omp for` when the programmer understands the task and workload distribution well.

Note 2:

The speedup cannot increase infinitely when adding more threads. According to Amdahl's Law, the maximum possible speedup is limited by the serial part of the program and is given by $\text{Speedup} \leq 1 / s$, where s is the fraction of the code that must run serially and cannot be parallelized.

Task 3: Parallel Matrix Computation

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

/*
    Matrix multiplication: C = A * B
    A, B, C are N x N (stored in 1D arrays, row-major).
*/

int main(int argc, char *argv[]) {
    int N = 800;                      // default size (change later
    r or pass as argument)
    if (argc >= 2) N = atoi(argv[1]);

    // Allocate memory
    double *A = (double*)malloc((size_t)N * N * sizeof(double));
    double *B = (double*)malloc((size_t)N * N * sizeof(double));
    double *C = (double*)malloc((size_t)N * N * sizeof(double));

    if (!A || !B || !C) {
        printf("Memory allocation failed. Try smaller
N.\n");
        return 1;
    }

    // Initialize A and B with simple values
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i*N + j] = (double)(i + j);
            B[i*N + j] = (double)(i - j);
            C[i*N + j] = 0.0;
        }
    }
}
```

```

}

// ----- SERIAL -----
double t1 = omp_get_wtime();

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        double sum = 0.0;
        for (int k = 0; k < N; k++) {
            sum += A[i*N + k] * B[k*N + j];
        }
        C[i*N + j] = sum;
    }
}

double t2 = omp_get_wtime();
double serial_time = t2 - t1;

// Reset C
for (int i = 0; i < N*N; i++) C[i] = 0.0;

// ----- PARALLEL (MANUAL SPLIT WITH omp parallel) -----
double t3 = omp_get_wtime();

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    int nthreads = omp_get_num_threads();

    // Split rows among threads
    int chunk = N / nthreads;
    int start = tid * chunk;
    int end = (tid == nthreads - 1) ? N : start + chunk;

    for (int i = start; i < end; i++) {
        for (int j = 0; j < N; j++) {

```

```

        double sum = 0.0;
        for (int k = 0; k < N; k++) {
            sum += A[i*N + k] * B[k*N + j];
        }
        C[i*N + j] = sum;
    }

}

double t4 = omp_get_wtime();
double parallel_time = t4 - t3;

// Print results
printf("N = %d\n", N);
printf("Serial time: %f seconds\n", serial_time);
printf("Parallel time: %f seconds\n", parallel_time);
printf("Speedup: %fx\n", serial_time / parallel_time);

// Small correctness check (print a couple of values)
printf("Check: C[0][0]=%f, C[N-1][N-1]=%f\n", C[0], C[(N-1)*N + (N-1)]);

free(A);
free(B);
free(C);
return 0;
}

```

- In this task, I performed matrix multiplication of two square matrices using both serial and parallel implementations. The overall matrix size used was 600×600 , meaning each matrix had 600 rows and 600 columns.
- First, the serial version was executed using a single thread. This version computes all rows of the result matrix sequentially and serves as a baseline for performance comparison.
- Then, the parallel version was implemented using `#pragma omp parallel`. The rows of the result matrix were manually divided among the available

threads. Each thread computed a different subset of rows, allowing multiple parts of the matrix multiplication to be executed at the same time.

- Execution time was measured using `omp_get_wtime()`. The parallel version was significantly faster than the serial version because the workload was shared across multiple CPU cores.

As the number of threads increased, execution time decreased and speedup improved. However, the speedup did not increase infinitely due to thread overhead, memory access limitations, and the serial portion of the program, as explained by Amdahl's Law.

This experiment shows that matrix multiplication benefits from parallelization, especially for larger matrix sizes such as 600×600 .

```
root@Youa-Laptop:~# export OMP_NUM_THREADS=1
root@Youa-Laptop:~# ./a.out 600
N = 100000000
Serial time: 2.341996 seconds
Parallel time: 0.651068 seconds
Speedup: 3.597159x
Check: C[0]=100000000.000000, C[N-1]=100000000.000000
```

```
root@Youa-Laptop:~# export OMP_NUM_THREADS=7
root@Youa-Laptop:~# ./a.out 600
N = 100000000
Serial time: 2.317232 seconds
Parallel time: 0.338306 seconds
Speedup: 6.849506x
Check: C[0]=100000000.000000, C[N-1]=100000000.000000
```