HW4: OpenMP

1. Objective

The objective of this assignment is to parallelize the HW1 program using OpenMP. The original HW1 code generated a random array and repeatedly applied a smoothing algorithm to reduce its standard deviation. In HW2, we:

- Introduced OpenMP parallel loops using #pragma omp parallel for
- Used default(none) to explicitly control data-sharing attributes
- Applied reduction clauses instead of critical sections
- Ensured thread-safe random number generation
- Measured execution time for each major function and overall runtime
- Performed strong scaling tests with different numbers of threads

2. Approach and Parallelization Strategy

Each independent for loop was parallelized individually, while the function calls remained serialized. This ensures clean and safe parallel execution without dependencies between functions.

Table 1 OpenMP implementation

Function	Parallelization Approach	OpenMP Features Used
random_array()	Parallel element-wise initialization with thread-local RNG	#pragma omp parallel for default(none) + rand_r()
sum()	Parallel reduction over array elements	reduction(+:s)
stdev()	Parallel variance computation using reduction	reduction(+:variance)
smooth()	Parallel neighbor averaging into temp array, then copy back	#pragma omp parallel for default(none)

All #pragma omp parallel for directives used default(none), enforcing explicit variable scoping. The number of threads is controlled externally using the environment variable *OMP_NUM_THREADS*.

3. Implementation

Th code is available in the .zip file, named *smooth_omp_timed.c.*

4. Result

Test Setup:

- System: SimCenter cluster
- Array size: 100,000,000 doubles
- Iterations: 5 smoothing passes
- Environment variable: OMP_NUM_THREADS={1,2,4,16,32}

```
OMP_NUM_THREADS=1 ./smooth_omp_timed

OMP_NUM_THREADS=2 ./smooth_omp_timed

OMP_NUM_THREADS=4 ./smooth_omp_timed

OMP_NUM_THREADS=16 ./smooth_omp_timed

OMP_NUM_THREADS=32 ./smooth_omp_timed
```

Table 2 presents data for execution time (sum of *random_array* (s), *stdev* (s), *smooth* (s)) for individual run with increasing number of threads. It is observed that the execution time is inversely proportional to the number of threads.

Table 2 Number of threads and execution time data

No. Threads	Total (s)
1	10.635
2	5.887
4	2.560
16	1.140
32	0.646

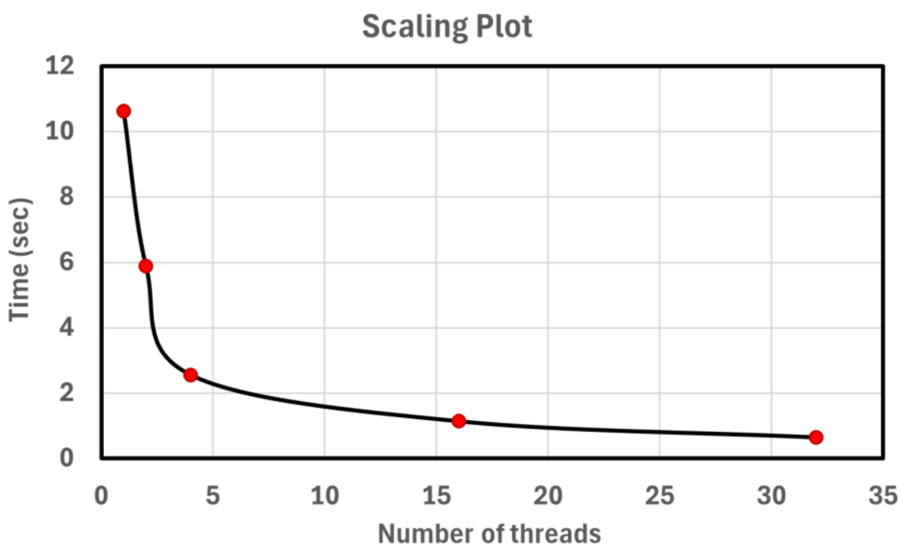

Figure 1 Scaling plot between execution time and number of threads

Figure 1 presents plot between total time and number of threads. The curve indicate strong scaling, approaching ideal 1/N scaling.