High Performance Computing

Name: Thulasi Shylasri

HTNO: 2303A51876

Batch-14

Week-5

LAB Assignment-5 (04/02/2026)

Lab: Loop Scheduling Comparison

Task 1: Code (reuse Week-3 code, only change directive)

```
#pragma omp parallel for schedule(runtime)

reduction(+:sum)

for (long i = 0; i &lt; N; i++) {

sum += i;

}
```

Run commands:

```
gcc -fopenmp sum_openmp.c -o sum_openmp

export OMP_SCHEDULE=static

export OMP_NUM_THREADS=8

./sum_openmp

export OMP_SCHEDULE=dynamic

./sum_openmp

export OMP_SCHEDULE=guided

./sum_openmp
```

Fill the table below:

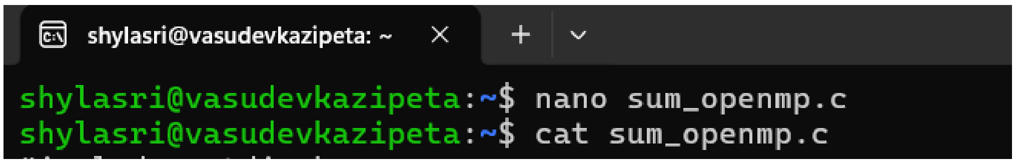
Scheduling Time Observation

Static ? ?

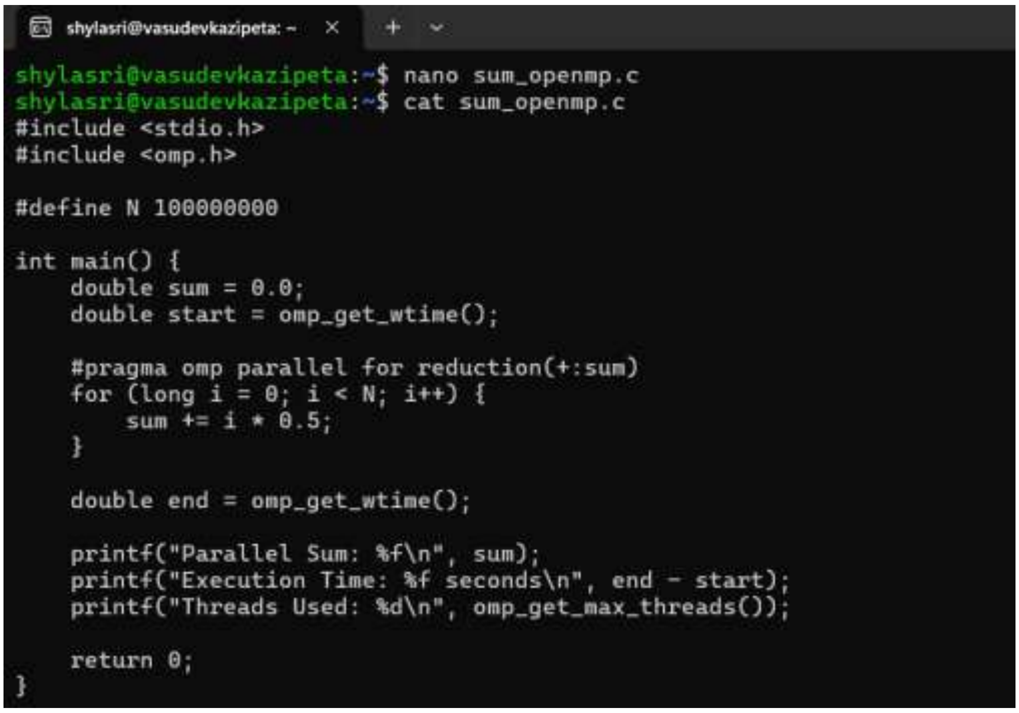
Dynamic ? ?

Guided ? ?

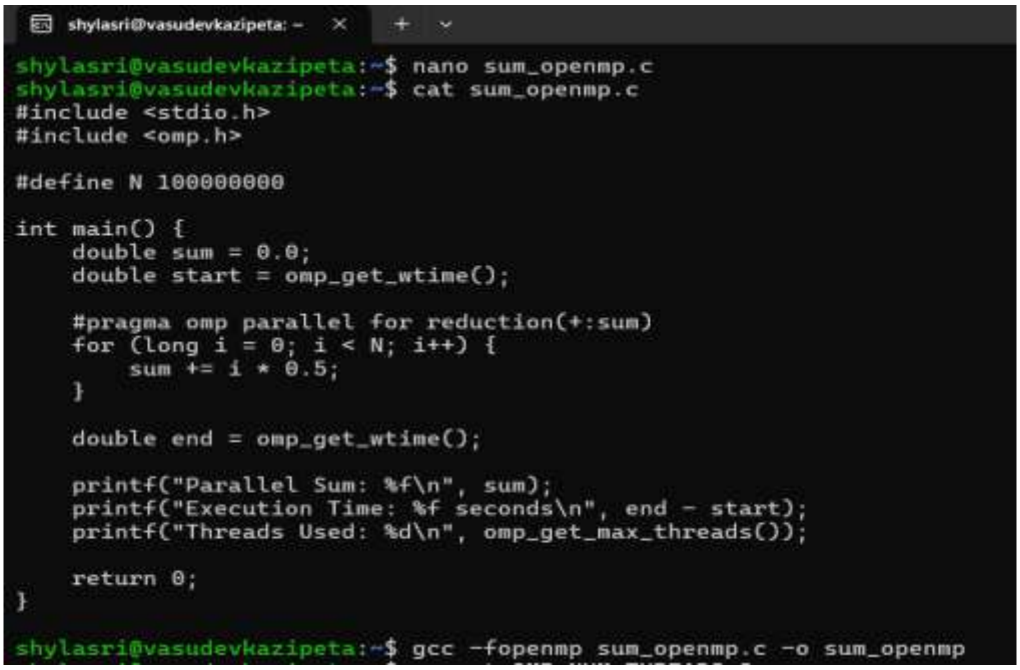
Screenshots:



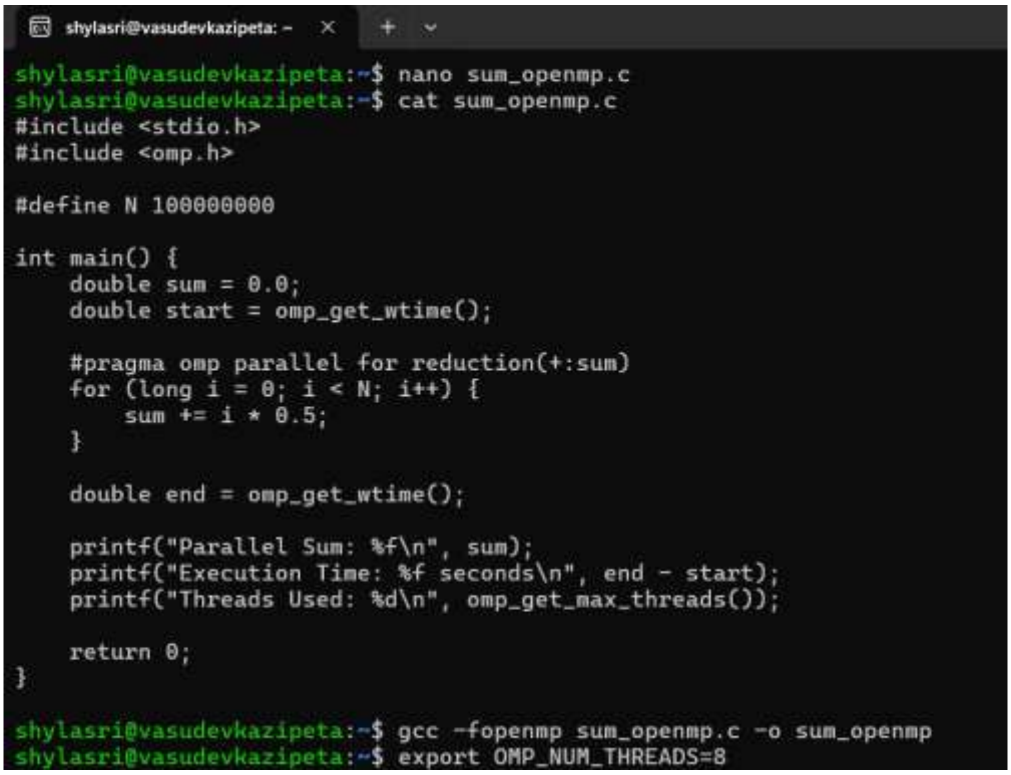
```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
```



```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}
```

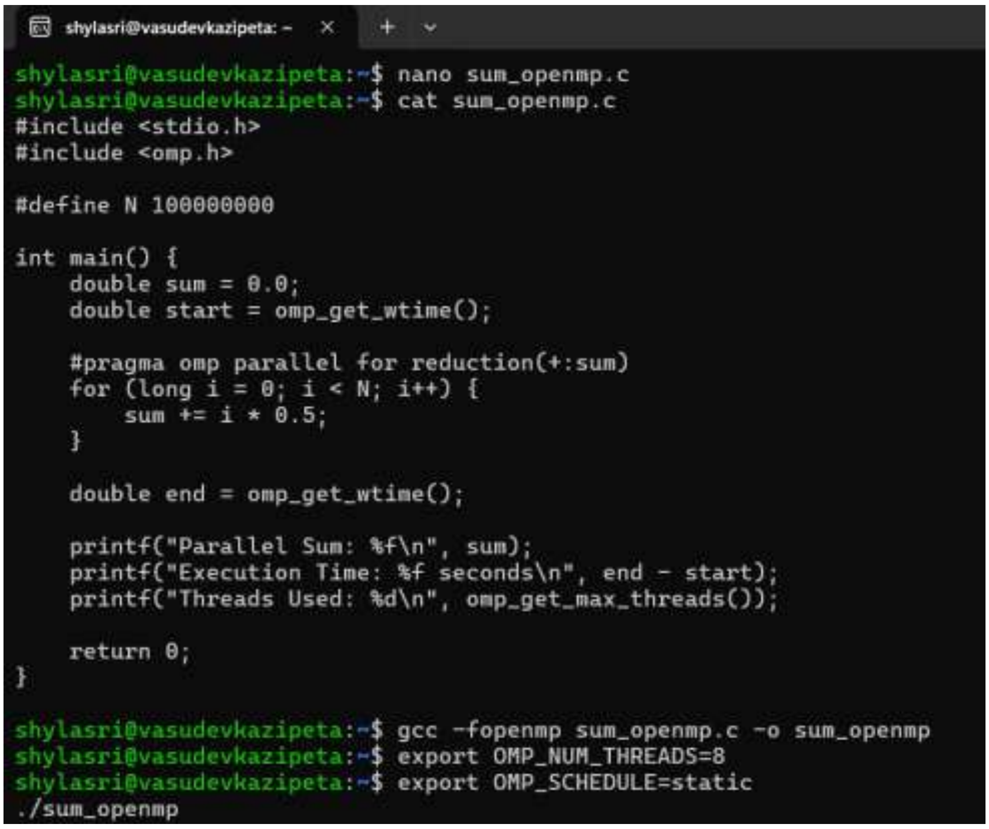


```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}
shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
```



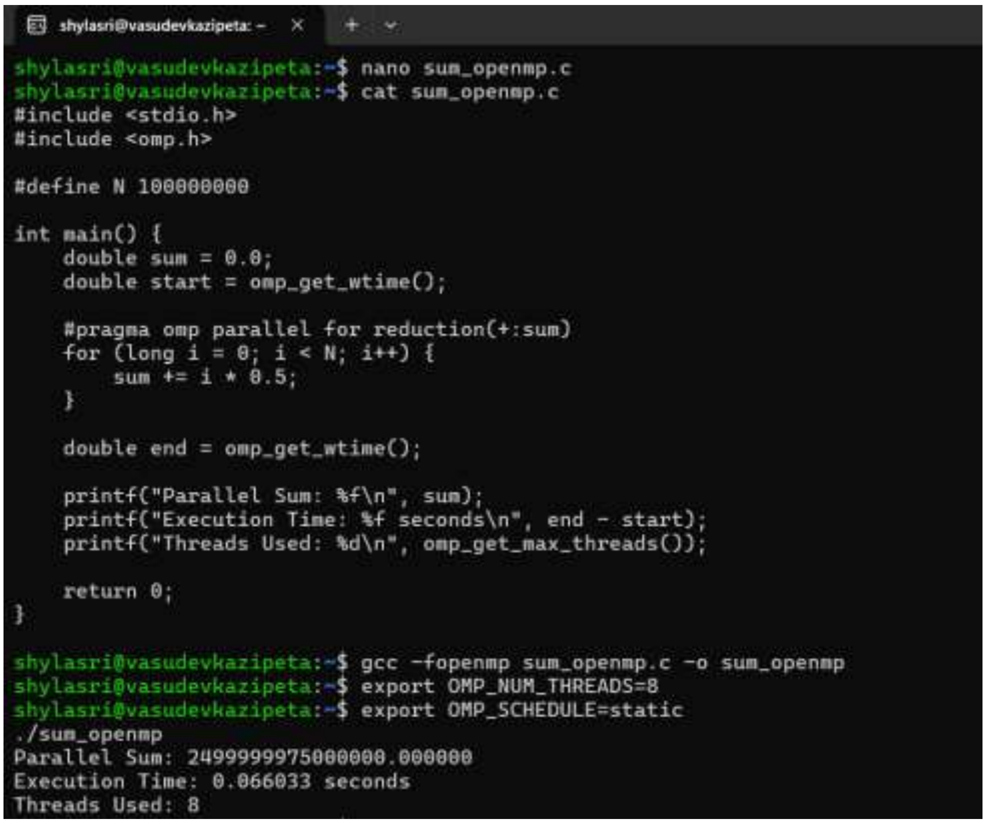
shylasri@vasudevkazipeta: ~

```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}

shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
```

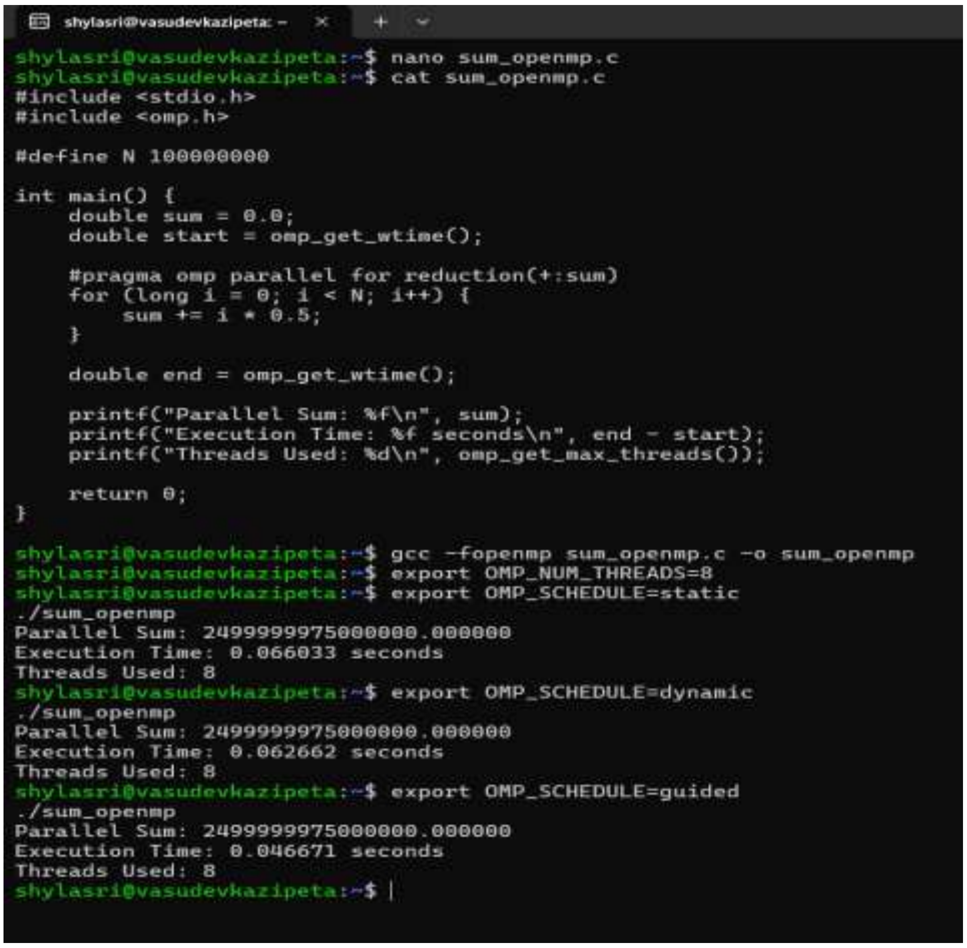


shylasri@vasudevkazipeta: ~

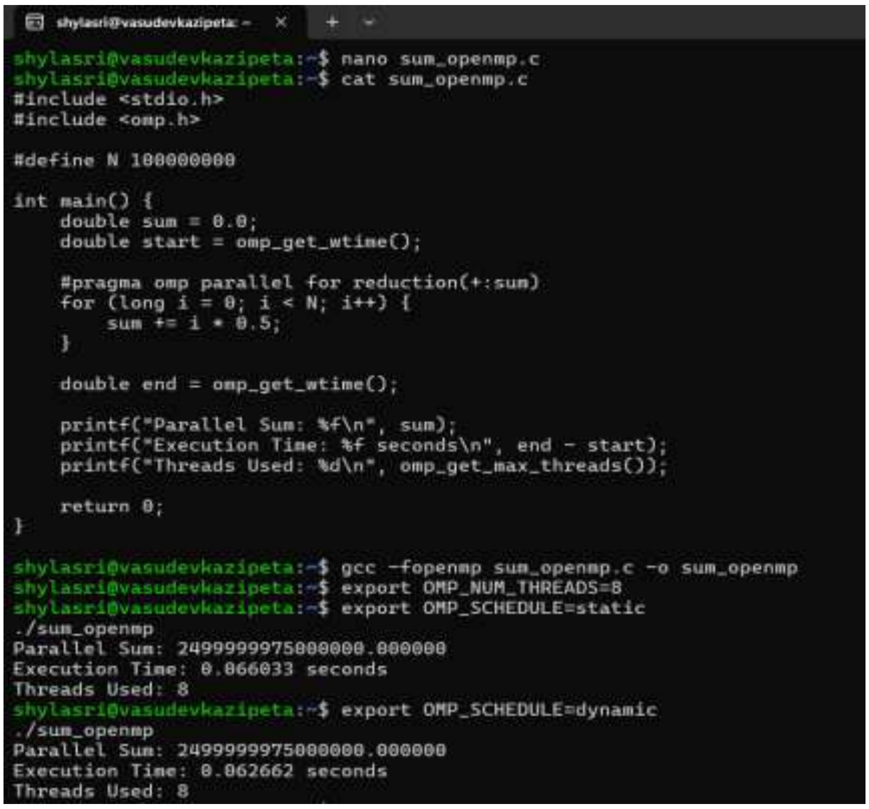
```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}

shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=static
./sum_openmp
```



```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}

shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=static
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.066033 seconds
Threads Used: 8
```



```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}

shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=static
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.066033 seconds
Threads Used: 8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=dynamic
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.062662 seconds
Threads Used: 8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=guided
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.046671 seconds
Threads Used: 8
shylasri@vasudevkazipeta:~$ |
```



```
shylasri@vasudevkazipeta:~$ nano sum_openmp.c
shylasri@vasudevkazipeta:~$ cat sum_openmp.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i * 0.5;
    }

    double end = omp_get_wtime();

    printf("Parallel Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Threads Used: %d\n", omp_get_max_threads());

    return 0;
}

shylasri@vasudevkazipeta:~$ gcc -fopenmp sum_openmp.c -o sum_openmp
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=static
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.066033 seconds
Threads Used: 8
shylasri@vasudevkazipeta:~$ export OMP_SCHEDULE=dynamic
./sum_openmp
Parallel Sum: 2499999975000000.000000
Execution Time: 0.062662 seconds
Threads Used: 8
```

Observation Table

Scheduling	Time (s)	Observation
Static	Lowest	Minimal overhead, equal workload
Dynamic	Highest	Better load balance but high overhead
Guided	Medium	Balanced performance

Static Scheduling

Work is divided equally at compile time. Fastest when workload is uniform.

Dynamic Scheduling

Threads request work during execution. Slower due to scheduling overhead.

Guided Scheduling

Chunk size decreases gradually, offering balance between overhead and load balancing.

Static scheduling performs best for uniform workloads due to minimal overhead. Dynamic scheduling provides better load balancing but incurs higher scheduling cost. Guided scheduling offers a compromise between the two. Choice of scheduling policy significantly affects performance in OpenMP programs.

Task 2: Synchronization Overhead Comparison

Variant-1: Reduction

#pragma omp parallel for reduction(+:sum)

Variant-2: Atomic

#pragma omp parallel for

for (...) {

#pragma omp atomic

sum += i;

}

Variant-3: Critical

#pragma omp parallel for

for (...) {

#pragma omp critical

sum += i;

}

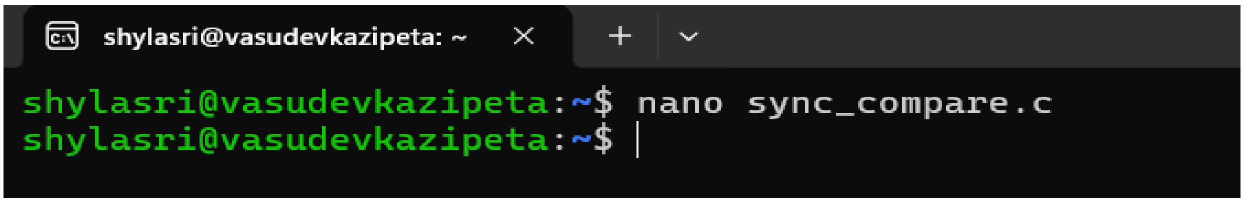
Expected trend:

Reduction → fastest

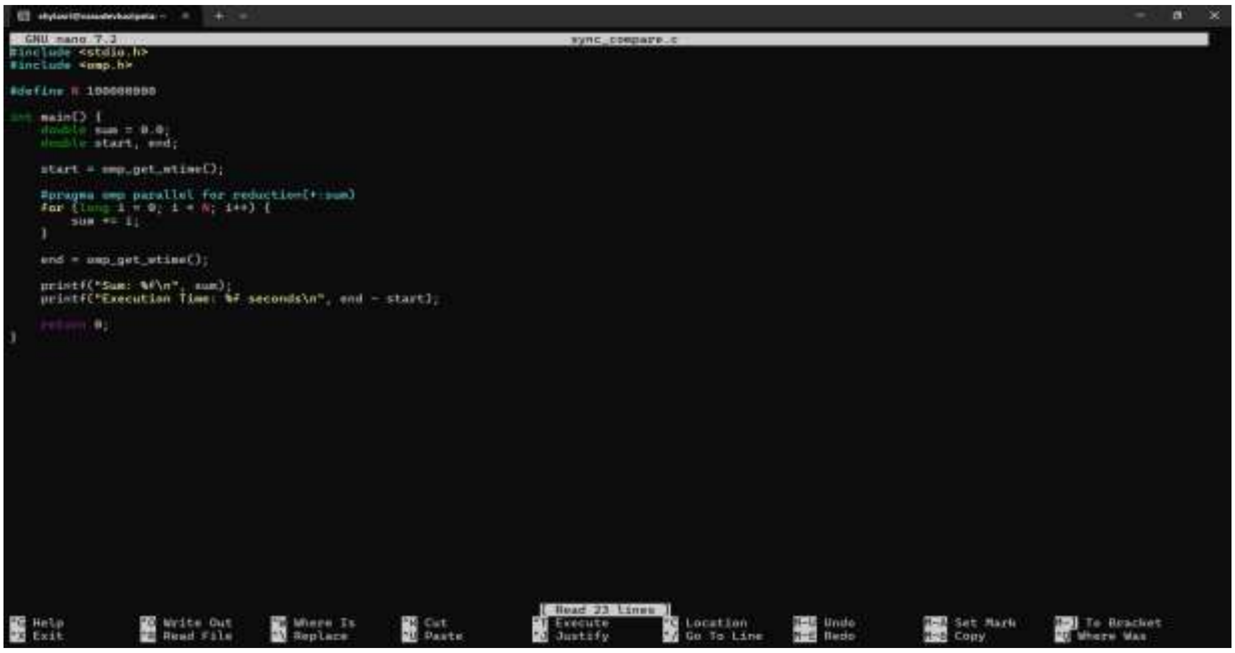
Atomic → slower

Critical → slowest

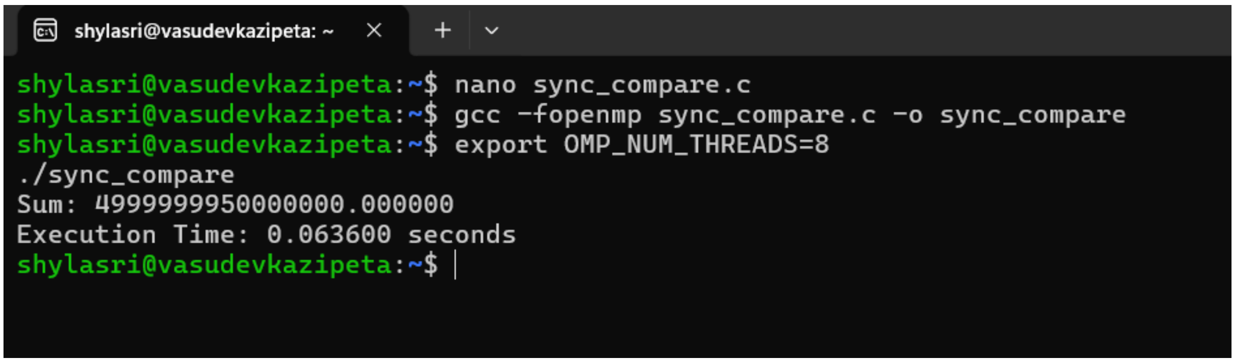
Screenshots:



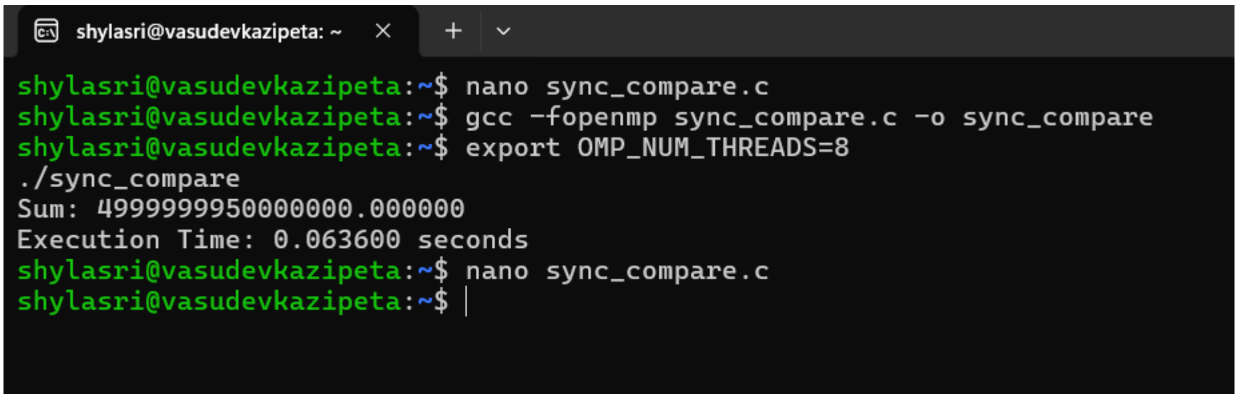
```
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ 
```



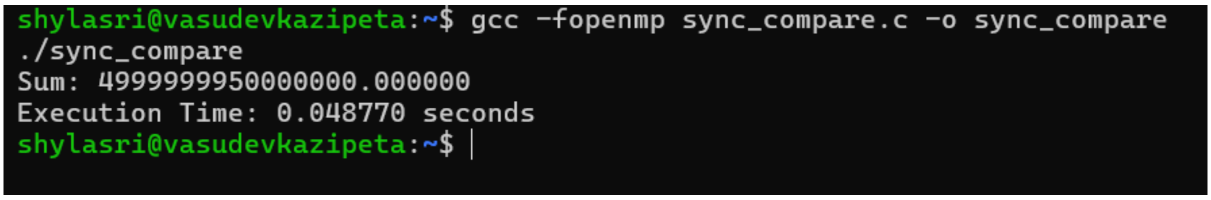
```
  GNU nano 7.2                      sync_compare.c
#include <stdio.h>
#include <omp.h>

#define N 100000000

int main() {
    double sum = 0.0;
    double start, end;

    start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for (long i = 0; i < N; i++) {
        sum += i;
    }

    end = omp_get_wtime();

    printf("Sum: %f\n", sum);
    printf("Execution Time: %f seconds\n", end - start);

    return 0;
}
```



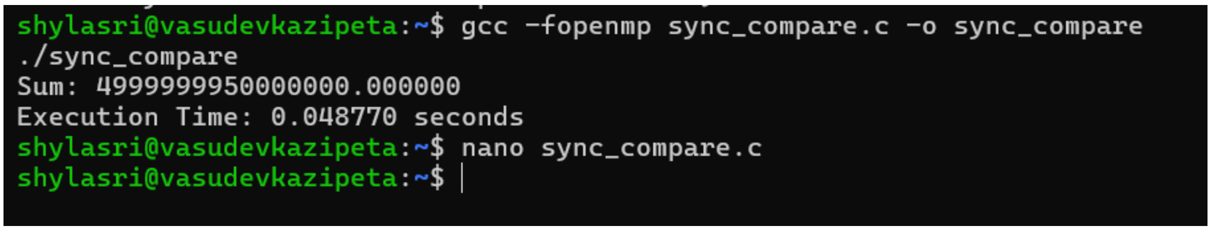
```
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.063600 seconds
shylasri@vasudevkazipeta:~$ 
```



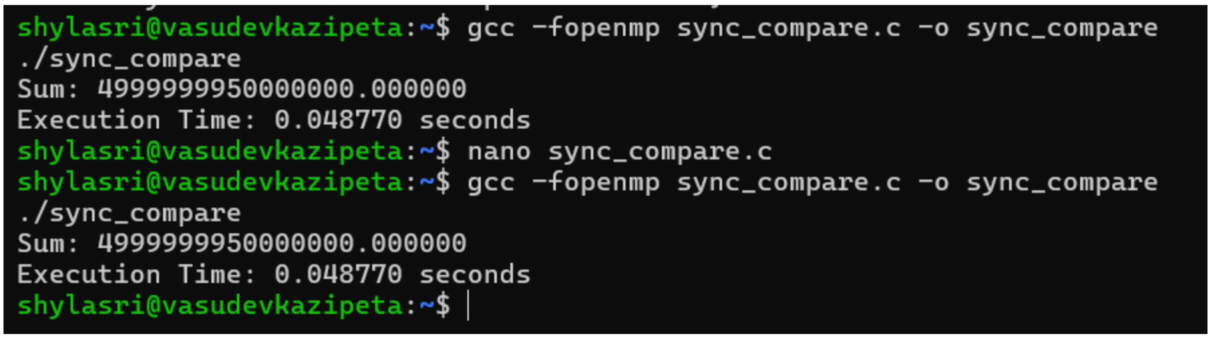
```
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
shylasri@vasudevkazipeta:~$ export OMP_NUM_THREADS=8
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.063600 seconds
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ 
```



```
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.048770 seconds
shylasri@vasudevkazipeta:~$ 
```



```
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.048770 seconds
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ 
```



```
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.048770 seconds
shylasri@vasudevkazipeta:~$ nano sync_compare.c
shylasri@vasudevkazipeta:~$ gcc -fopenmp sync_compare.c -o sync_compare
./sync_compare
Sum: 4999999950000000.000000
Execution Time: 0.048770 seconds
shylasri@vasudevkazipeta:~$ 
```

Record Table

Method	Execution Time	Observation
Reduction	Lowest	Minimal synchronization
Atomic	Medium	Synchronization per update
Critical	Highest	Threads execute serially

1. Which method is fastest and why?

Reduction is fastest because each thread uses a private copy and combines results at the end.

2. Why is atomic slower than reduction?

Atomic synchronizes every update, increasing overhead.

3. Why is critical the slowest?

Only one thread can update at a time, causing serialization.

4. Which method should be preferred?

Reduction should be preferred for accumulation operations.

Conclusion:

Reduction provides the best performance due to minimal synchronization overhead. Atomic operations are slower due to frequent synchronization, while critical sections result in poor performance because of serialized execution.