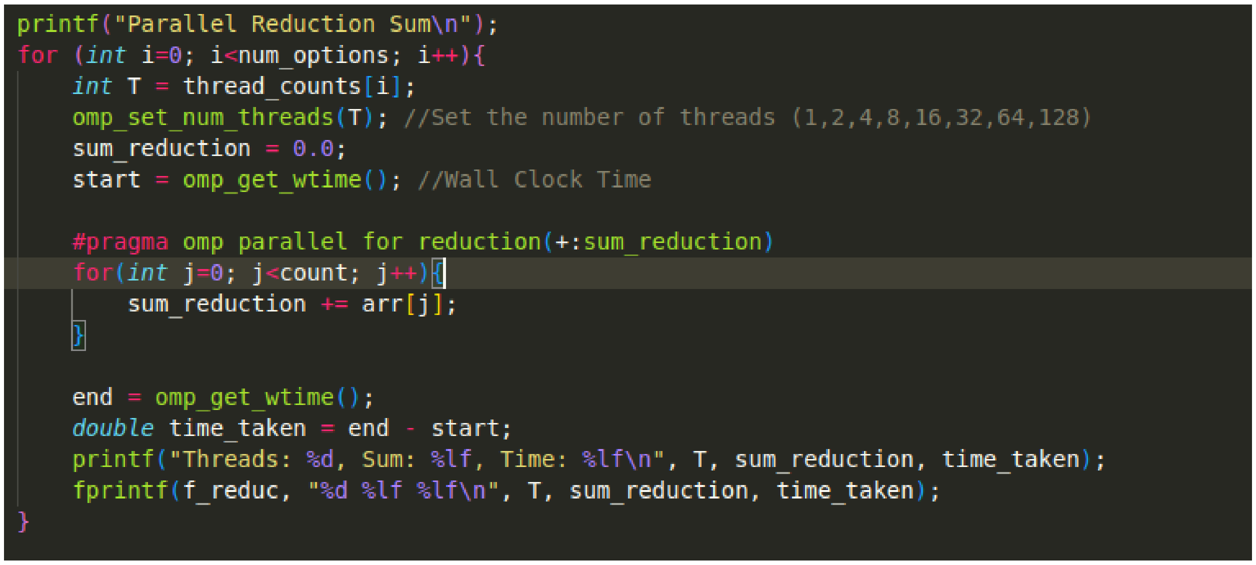
HPC OPENMP TUTORIAL 2 REPORT

CS22B2015 – HARSHITH B

The parallel code for reduction and critical sections is provided in the main.c file. The key section of the code performs the following steps:

1. Threads are allocated in powers of 2 (1, 2, 4, 8, 16, 32, 64, 128) within a loop, and the sum is calculated using the OpenMP pragma reduction.
2. A dataset consisting of approximately 1.5 million double-precision floating-point numbers is generated using Python and stored in the output.txt file.
3. The threads versus time data is recorded into a text file. This data is then processed, and the respective results are plotted and analyzed using results.py.

1. Parallel Code Segment for Reduction

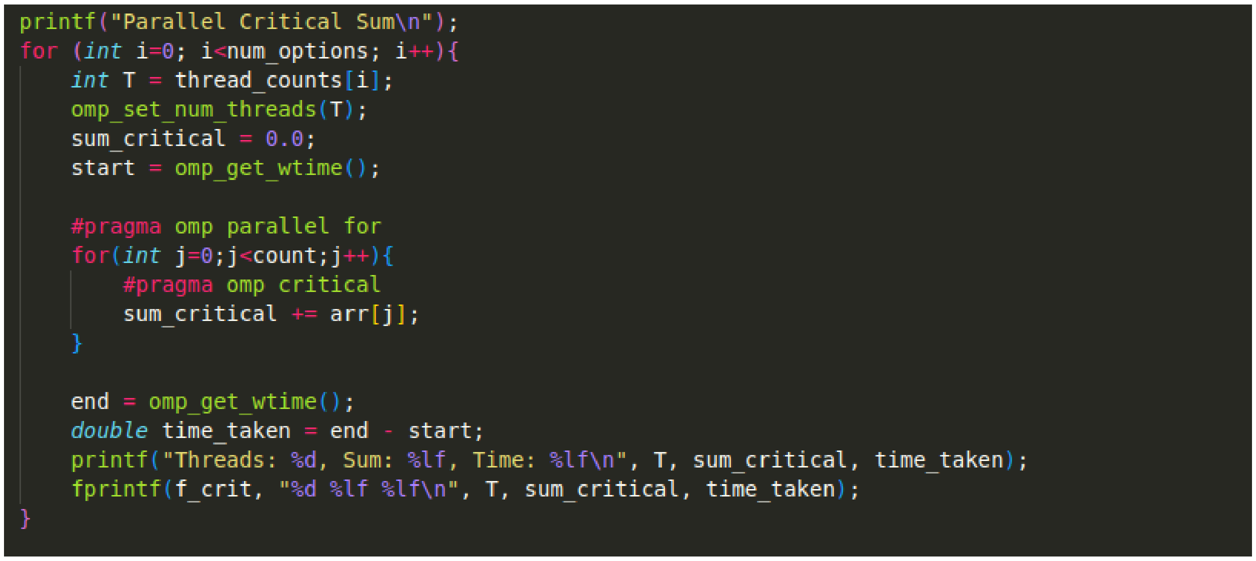


```c
printf("Parallel Reduction Sum\n");
for (int i=0; i<num_options; i++){
    int T = thread_counts[i];
    omp_set_num_threads(T); //Set the number of threads (1,2,4,8,16,32,64,128)
    sum_reduction = 0.0;
    start = omp_get_wtime(); //Wall Clock Time

    #pragma omp parallel for reduction(+:sum_reduction)
    for(int j=0; j<count; j++){
        sum_reduction += arr[j];
    }

    end = omp_get_wtime();
    double time_taken = end - start;
    printf("Threads: %d, Sum: %lf, Time: %lf\n", T, sum_reduction, time_taken);
    fprintf(f_reduc, "%d %lf %lf\n", T, sum_reduction, time_taken);
}
```

2. Parallel Code Segment for Critical Section

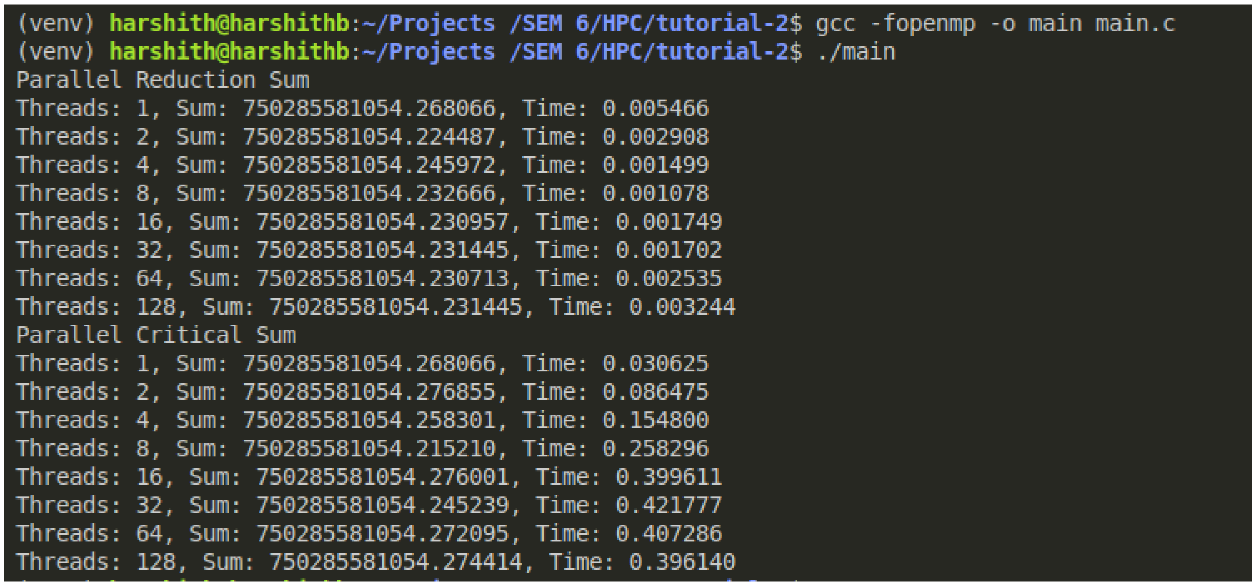


```c
printf("Parallel Critical Sum\n");
for (int i=0; i<num_options; i++){
    int T = thread_counts[i];
    omp_set_num_threads(T);
    sum_critical = 0.0;
    start = omp_get_wtime();

    #pragma omp parallel for
    for(int j=0;j<count;j++){
        #pragma omp critical
        sum_critical += arr[j];
    }

    end = omp_get_wtime();
    double time_taken = end - start;
    printf("Threads: %d, Sum: %lf, Time: %lf\n", T, sum_critical, time_taken);
    fprintf(f_crit, "%d %lf %lf\n", T, sum_critical, time_taken);
}
```

3. Terminal Output



```
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-2$ gcc -fopenmp -o main main.c
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-2$ ./main
Parallel Reduction Sum
Threads: 1, Sum: 750285581054.268066, Time: 0.005466
Threads: 2, Sum: 750285581054.224487, Time: 0.002908
Threads: 4, Sum: 750285581054.245972, Time: 0.001499
Threads: 8, Sum: 750285581054.232666, Time: 0.001078
Threads: 16, Sum: 750285581054.230957, Time: 0.001749
Threads: 32, Sum: 750285581054.231445, Time: 0.001702
Threads: 64, Sum: 750285581054.230713, Time: 0.002535
Threads: 128, Sum: 750285581054.231445, Time: 0.003244
Parallel Critical Sum
Threads: 1, Sum: 750285581054.268066, Time: 0.030625
Threads: 2, Sum: 750285581054.276855, Time: 0.086475
Threads: 4, Sum: 750285581054.258301, Time: 0.154800
Threads: 8, Sum: 750285581054.215210, Time: 0.258296
Threads: 16, Sum: 750285581054.276001, Time: 0.399611
Threads: 32, Sum: 750285581054.245239, Time: 0.421777
Threads: 64, Sum: 750285581054.272095, Time: 0.407286
Threads: 128, Sum: 750285581054.274414, Time: 0.396140
```

4. Thread vs Time Plots and Observations

Below are the plots for Thread vs Time for Reduction and Critical Section and Observations.

• Reduction Sum:

– The reduction results show a significant decrease in time as threads increase from

1 to 8.

– The best performance is achieved around 8 threads (0.001078 sec), after which further thread increases do not help and slowly increase the overhead and time required.

– This indicates that the reduction construct scales well until a certain point where coordination/overhead limits additional gains.

• Critical Sum:

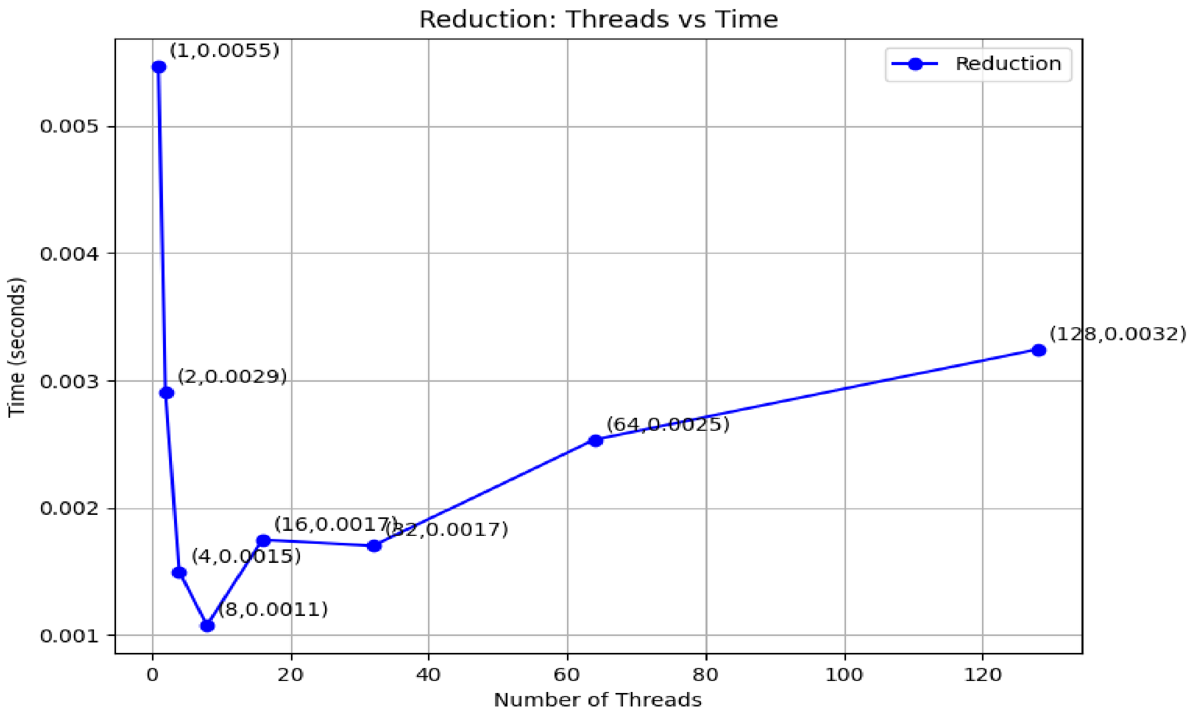
– The critical section times are much higher overall and increase monotonically with added threads – from 0.030625 sec at 1 thread to about 0.399–0.421 sec for 16–32 threads, with similar values for 64 and 128 threads.

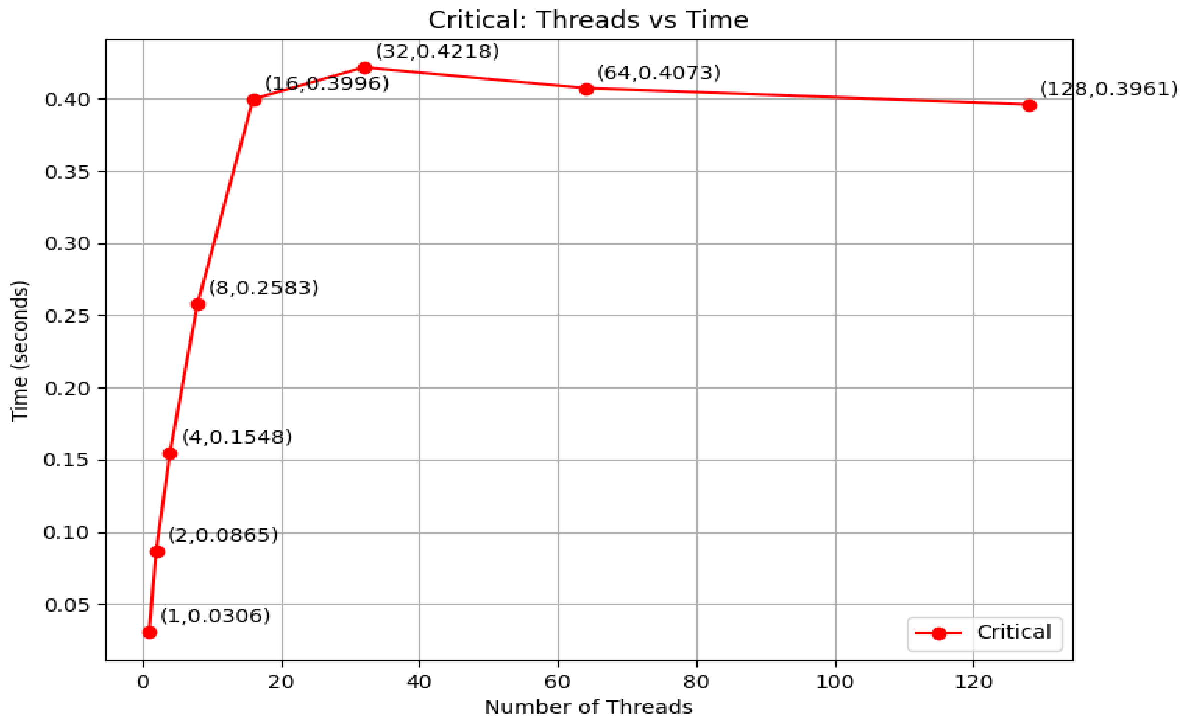
– Since the sum update is done inside a critical section, threads contend for access to the shared variable, leading to serialization of the update and poor scaling.

– The results highlight that the overhead from frequent locking in critical sections can dominate and even degrade performance in highly parallel configurations.

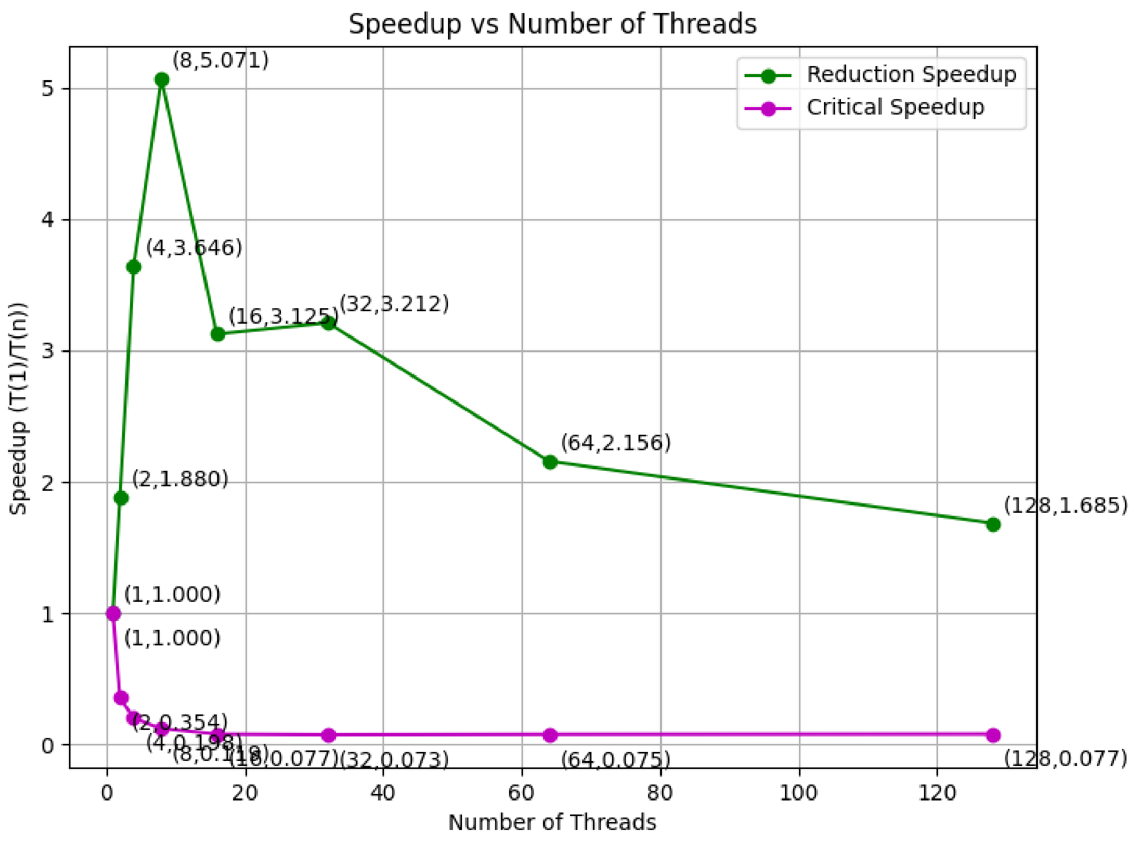
Inference :

From the above observations the reduction approach is far more efficient and scalable than the critical section approach for this summation task.





5. SpeedUp vs Processors (SpeedUp == T(1) / T(n))



6. Parallelization Fraction and Inference

We try to estimate the parallelization fraction (p) for two parallel summation implementations:

- **Parallel Reduction Sum**
- **Parallel Critical Sum**

The estimation uses Amdahl's Law to determine the effective parallel fraction for each method, selecting the best performance configuration—i.e., the configuration with the lowest execution time—to minimize the impact of overhead and better represent the inherent parallelism.

Amdahl's Law:

Amdahl's Law describes the maximum speedup achievable by parallelizing a fraction of a program. The formula is:

S(n) = 1 / [(1 - p) + (p / n)]

Where:

S(n) = Speedup using n threads

p = Fraction of the program that is parallelizable

n = Number of threads

Rearranging this formula to solve for p, we get:

p = [n * (S(n) - 1)] / [S(n) * (n - 1)] -> Implemented in Code to calculate the p value

Methodology and Best Performance:

For each method, the configuration with the lowest execution time is chosen as the best-case performance, as it minimizes the impact of overhead and more accurately reflects the method's inherent parallelism.

Results:

- **Best Reduction Performance**: Achieved at 8 threads with a speedup of 5.071.
    - **Estimated Parallelization Fraction (Reduction)**: p = 0.917
- **Best Critical Performance**: Achieved at 1 thread with a speedup of 1.000.
    - **Estimated Parallelization Fraction (Critical)**: p = 0.000

Parallel Reduction Sum:

- **Best configuration**: 8 threads
- **Measured Speedup (S_opt)**: 5.071

Using Amdahl's Law:

p = (8 * (5.071 - 1)) / (5.071 * (8 - 1)) = (8 * 4.071) / (5.071 * 7) ≈ 0.917

This indicates that approximately 91.7% of the code is parallelizable using the reduction method.

Parallel Critical Sum:

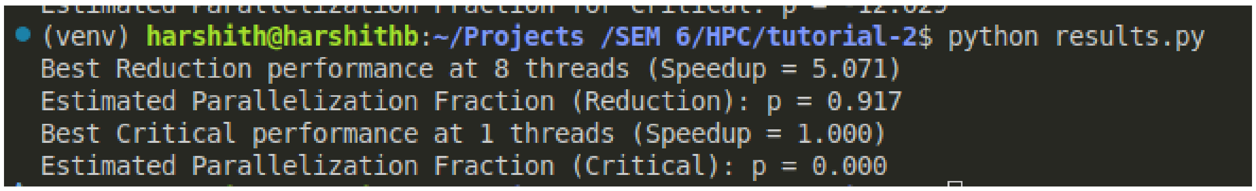
- **Best configuration**: 1 thread
- **Measured Speedup (S_opt)**: 1.000

Consequently, p = 0.000, meaning no effective parallelization is achieved due to critical section overhead.

Inference:

- The **Parallel Reduction Sum** method demonstrates significant scalability, with an estimated parallel fraction of 0.917. This suggests that most of the computation is parallelizable. The best performance was observed at 8 threads, yielding a speedup of 5.071. Beyond 8 threads, further parallelization may not provide additional benefits due to overhead and diminishing returns.
- The **Parallel Critical Sum** method, on the other hand, is limited by heavy locking overhead. Its best performance occurs with 1 thread, resulting in a speedup of 1.000 and an estimated parallel fraction of 0.000, indicating no meaningful parallelization.

Overall, the results highlight that the parallel reduction strategy is far more effective in utilizing multi-threading than the critical section approach, which fails to provide any significant parallel speedup.



```
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-2$ python results.py
Best Reduction performance at 8 threads (Speedup = 5.071)
Estimated Parallelization Fraction (Reduction): p = 0.917
Best Critical performance at 1 threads (Speedup = 1.000)
Estimated Parallelization Fraction (Critical): p = 0.000
```