HPC Tutorial 2 Report

CS22B2012
K Aditya Sai

The code for Reduction and Critical Section is implemented in C++ with OpenMP

1. **Thread Allocation**: The code is executed with thread counts ranging from 1 to 64

2. **Dataset Generation**: A C++ code generates 10 million double-precision floating-point values randomly which are stored in input.txt.

3. **Performance Analysis**: Execution times for reduction and critical methods are recorded and speedup/parallelization fractions are calculated.

4. **Visualization**: Python scripts generate plots for execution time and speedup.

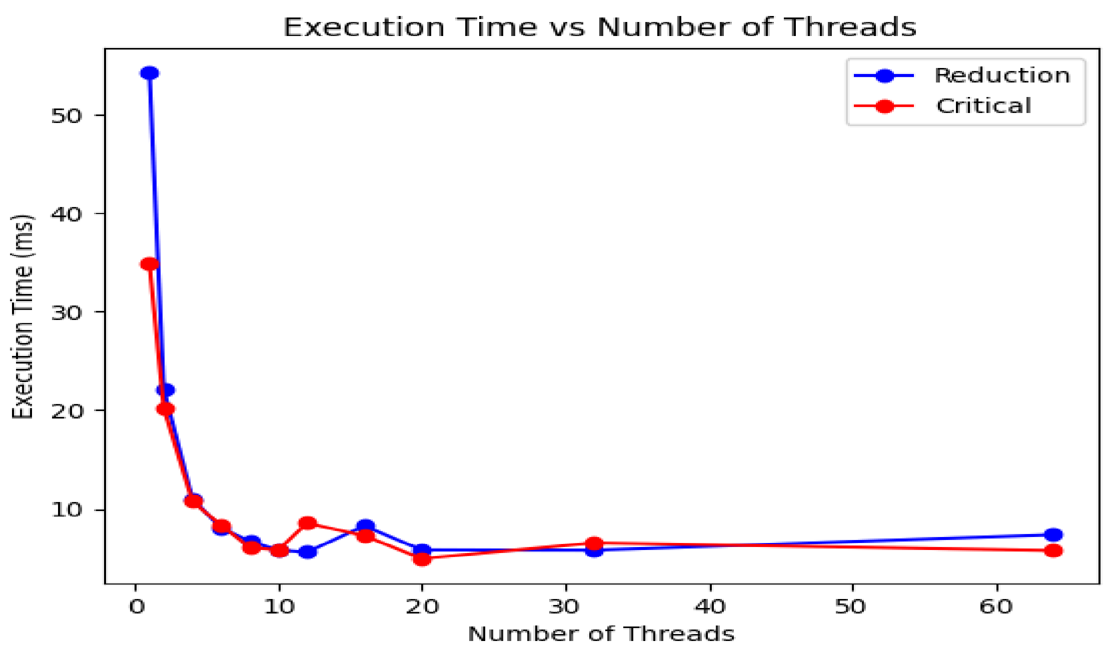
Parallel Code for Reduction

```
double sum_reduction(const vector<double> &data)
{
    double sum = 0.0;
    #pragma omp parallel for reduction(+ : sum) shared(data)
    for (size_t i = 0; i < data.size(); i++)
    {
        sum += data[i];
    }
    return sum;
}
```

Parallel Code for Critical Section

```
double critical_section(const vector<double> &data)
{
    double sum = 0.0;
    #pragma omp parallel
    {
        double local_sum = 0.0;
        #pragma omp for
        for (size_t i = 0; i < data.size(); i++)
        {
            local_sum += data[i];
        }
        #pragma omp critical
        {
            sum += local_sum;
        }
    }
    return sum;
}
```

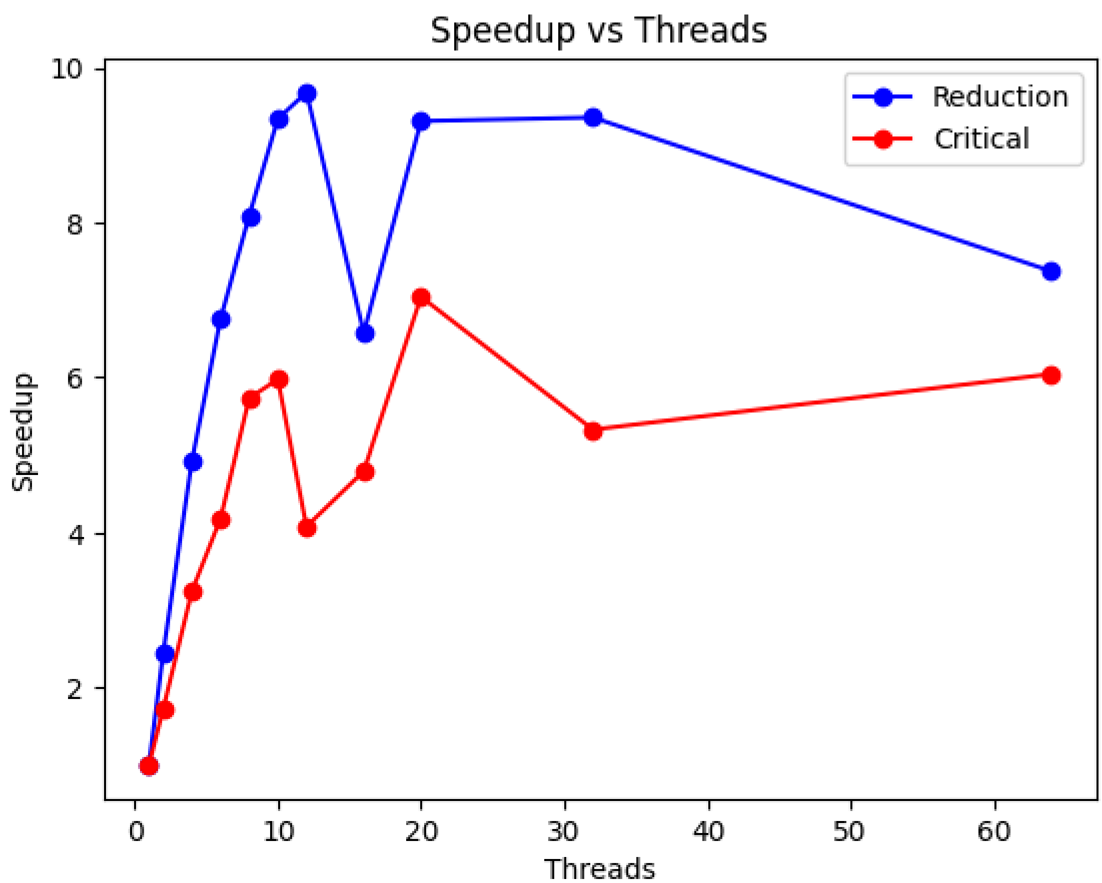
Plot Threads v/s Time :



Observations :

1. The execution time decreases at first but then we see a sharp increase as the number of context switches between the threads increases especially in the case of critical section parallelization.
2. Optimal execution time is achieved at 12 Threads for reduction and 20 threads for critical section.

Plot Threads v/s Speedup:



Observations :

1. Speedup increases up-to 12 threads in case of reduction and we see a slow decline of the speedup in case of critical section.
2. The peak speedup for reduction is reached when 12 threads are used.

Inferences :

1. Since speedup is calculated by Amdahl's law, the speedup is inversely proportional to parallelized execution time.
2. Parallelized execution time is negatively affected by increase in number of threads after some number of threads due to the overhead of context switch.
3. This overhead is existent is all cases but becomes dominant as the number of threads keeps increasing.

Estimated Parallelization Fraction :

=== Parallelization Fraction Table ===

Threads	P. Fraction (Reduction)	P. Fraction (Critical)
2	1.185072	0.840607
4	1.062680	0.922283
6	1.022396	0.913202
8	1.001597	0.943467
10	0.992167	0.925171
12	0.978257	0.823506
16	0.904511	0.844219
20	0.939601	0.903150
32	0.921966	0.838429
64	0.878089	0.847790

Observations :

1. The parallelization fraction reaches its peak at 10 threads.
2. Parallel fraction initially increases due to efficient distribution among threads.
3. Ideally, PF should be in the range [0, 1], but values **greater than 1** indicate an **overestimation of speedup** (or underestimation of serial execution time).
4. This can happen due to **measurement inaccuracies, system noise, or CPU turbo-boosting effects** when running with fewer threads.
5. Parallel efficiency reduces at higher thread counts, likely due to: Overhead of thread management, Memory access contention.

Conclusion:

- **Reduction outperforms Critical Section**: The reduction method achieves higher speedup and parallelization efficiency, confirming its superiority for summation tasks by minimizing synchronization overhead.
- **Critical section suffers from synchronization bottlenecks**: Performance fluctuates more in the critical section approach, showing its inefficiency for large thread counts.
- **Negative parallelization fraction anomalies**: Values slightly above 1 indicate system noise, turbo-boosting, or measurement inaccuracies, but overall trends align with Amdahl's Law.
- **Optimal thread count is around 8-12** for balancing performance and efficiency.