Parallel Traffic Accident Analysis

K. Sai Sri Neharika (2023BCD0053)

A. Pranathi (2023BCD0065)

INTRODUCTION

- The number of vehicles on roads is increasing rapidly, leading to more traffic jams and accidents.
- These accidents not only result in loss of lives but also cause major economic damage.
- Studying and analyzing accident data helps in identifying accident-prone areas and understanding their causes.
- When the dataset becomes large, analyzing it using traditional (serial) methods takes a lot of time.
- Using parallel computing allows us to process data faster by dividing the work among multiple processors, improving both speed and efficiency.

PROBLEM STATEMENT

- With the growing amount of traffic accident data, it’s important to find a faster way to analyze and extract meaningful information from it.
- However, the serial approach becomes slow and inefficient when handling millions of records.
- To solve this issue, the project focuses on developing a parallel version using OpenMP to speed up the data processing.

Traditional Serial Approach

- The program reads accident data from a CSV file and stores it in a structured format.
- Each record includes details like severity, cause, weather, traffic density, and number of vehicles.
- It processes all data sequentially i.e., one record at a time, without using multiple threads.
- Performs calculations for total accidents, injuries, fatalities, and average vehicles per accident.
- Generates key distributions such as accident severity, causes, and traffic density.
- Identifies peak accident hours and high-risk zones based on nearby accident counts.

Analysis function (serial approach)

```cpp
// Analysis Function
void analyze(const vector<Accident> &accidents)
{
  long long total_inj = 0, total_fat = 0, total_veh = 0, total_nearby = 0;
  unordered_map<string, int> sevCount, causeCount, trafficCount, weatherCount;
  vector<int> hourly(24, 0);

  for (auto &a : accidents)
  {
    total_inj += a.injuries;
    total_fat += a.fatalities;
    total_veh += a.vehicles;
    total_nearby += a.nearby_accidents;
    sevCount[a.severity]++;
    causeCount[a.accident_cause]++;
    trafficCount[a.traffic_density]++;
    weatherCount[a.weather]++;
    int h = extractHour(a.time);
    if (h >= 0 && h < 24)
      hourly[h]++;
  }
```

Challenges In SerialApproach

- The program processes data one record at a time, which makes it very slow for large datasets.
- As the dataset grows, the execution time increases significantly, reducing efficiency.
- It uses only a single CPU core, leaving the remaining system resources idle.
- Difficult to handle real-time or large-scale accident data analysis due to limited speed.
- Any delay in one step affects the entire process since tasks run one after another.

Parallel Algorithm Using OpenMP

Data Loading and Thread division

- Read CSV file
- Parse Records
- Store them in a vector structure

Local Computation

#pragma omp parallel

- Each thread computes local totals (injuries, fatalities, vehicles, hourly counts) independently using private variables to avoid data conflicts.

Result Merging

#pragma omp atomic / reduction(+ : var)

- Safely combine results from all threads using atomic operations and reduction clauses.
- Ensures accurate, thread-safe global statistics.

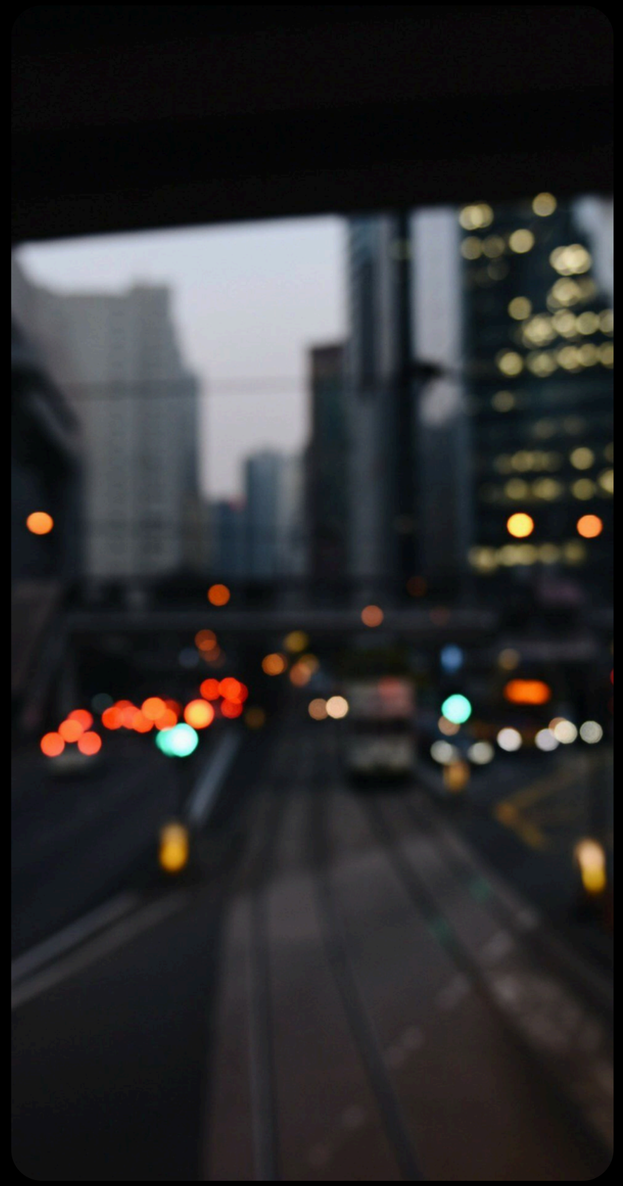
Analysis function (Parallel approach)

```
110  void analyze(const vector<Accident> &accidents)
111  {
112    int n = accidents.size();
113    if (n == 0)
114      return;
115
116    // Global accumulators
117    long long total_inj = 0, total_fat = 0, total_veh = 0, total_nearby = 0;
118    vector<int> hourly(24, 0);
119
120  // Parallel numerical + hourly aggregation
121  #pragma omp parallel
122    {
123      long long local_inj = 0, local_fat = 0, local_veh = 0, local_nearby = 0;
124      vector<int> local_hourly(24, 0);
125
126  #pragma omp for schedule(static)
127      for (int i = 0; i < n; i++)
128      {
129        local_inj += accidents[i].injuries;
130        local_fat += accidents[i].fatalities;
131        local_veh += accidents[i].vehicles;
132        local_nearby += accidents[i].nearby_accidents;
133        int h = accidents[i].hour;
134        if (h >= 0 && h < 24)
135          local_hourly[h]++;
136      }
```

```
138    // Merge local results using atomic updates
139  #pragma omp atomic
140        total_inj += local_inj;
141  #pragma omp atomic
142        total_fat += local_fat;
143  #pragma omp atomic
144        total_veh += local_veh;
145  #pragma omp atomic
146        total_nearby += local_nearby;
147
148        for (int h = 0; h < 24; ++h)
149        {
150          if (local_hourly[h] != 0)
151          {
152  #pragma omp atomic
153            hourly[h] += local_hourly[h];
154          }
155        }
156      }
```

Performance Comparison

Dataset	Serial Time	Parallel Time	SpeedUp
Original Dataset	0.622	0.13	4.7846x
Test Case 1	0.050	0.05	1x
Test Case 2	0.086	0.05	1.72x
Test Case 3	0.080	0.05	1.6x



Key Achievements

1 **Speedup**

Parallel version nearly 5x faster than serial

2 **Efficiency**

Strong parallel efficiency on multi-core systems

Technical Success

- Efficient OpenMP parallelization
- Thread-safe categorical counting
- Atomic operations for shared data
- Linear time complexity O(N)

Practical Impact

- Rapid analysis of 1M+ records
- Identifies high-risk zones
- Reveals temporal patterns
- Supports safety interventions

Time Complexity Analysis

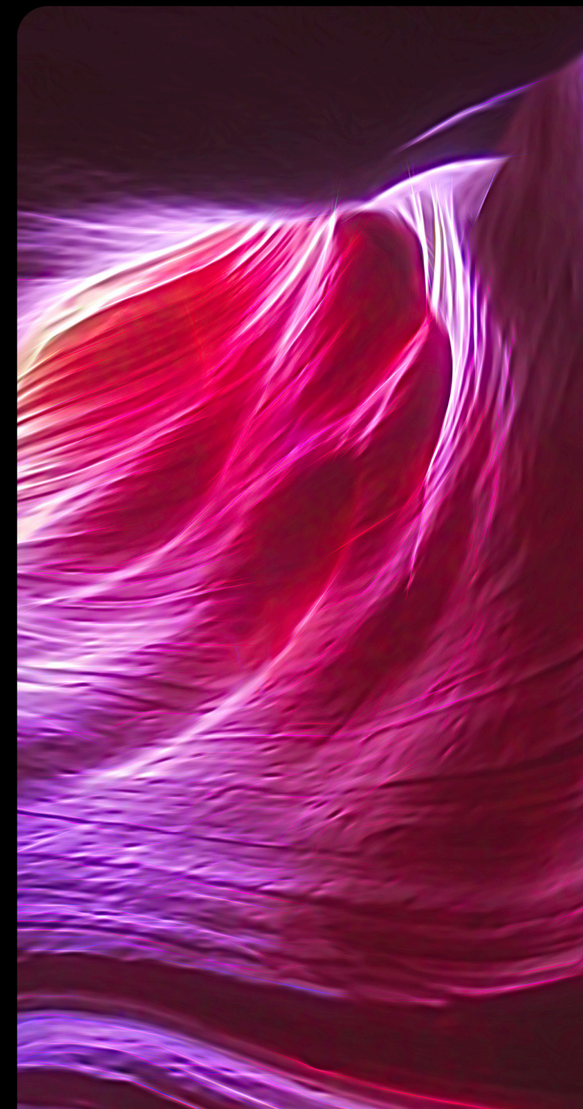
- The serial version processes each record one by one, giving a time complexity of O(N).
- The parallel version divides the dataset among multiple threads, reducing it to O(N/p) where p is the number of threads.
- Parallel execution achieves faster processing and better scalability for large datasets.
- Experimental results show around 4–5× speedup over the serial approach.

References

1. Freeway accident analysis using second order statistics by LSU Scholarly Repository

2. Analysis of traffic accident characteristics and recovery strategy of urban road network. (Article)

3. Analysis of traffic accidents based on Spark and casual inference



Conclusion

The parallel version using OpenMP made the analysis almost five times faster than the serial one.

It handled large datasets efficiently by sharing the work among multiple threads.

Overall, the project proved that parallel processing improves performance and is well-suited for real-world traffic analysis.