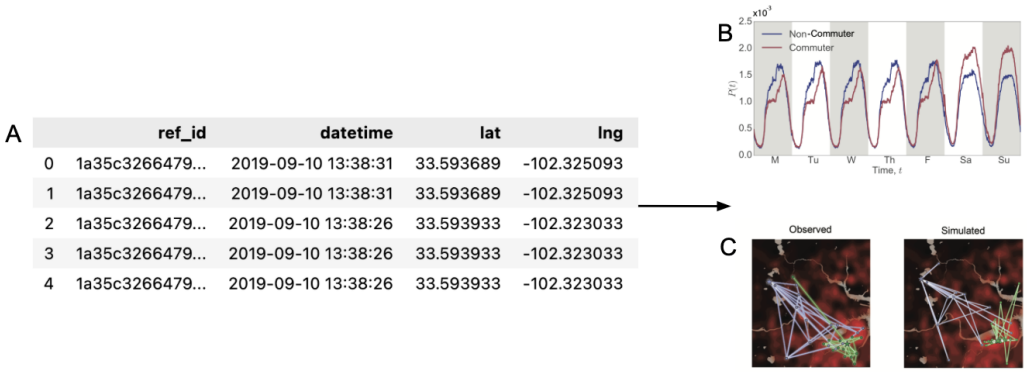
Parallelizing TimeGeo for Scalable Urban Mobility Simulations

Aparimit Kasliwal

Contents

1 Introduction

TimeGeo is a data-driven framework for simulating individual urban mobility using sparse, privacy-preserving spatiotemporal data from Location-Based Services (LBS). This data consists of periodic *pings* from mobile applications, capturing spatial patterns in a user's daily movement. These high-precision location points, when clustered within short time windows, reveal *stay points*—locations where users remain for extended periods. From these, home and work locations can be inferred. Leveraging these stay points and user-specific behavioral parameters, TimeGeo generates realistic, individual-level daily mobility trajectories through geospatial and temporal modeling.



	ref_id	datetime	lat	lng
0	1a35c3266479...	2019-09-10 13:38:31	33.593689	-102.325093
1	1a35c3266479...	2019-09-10 13:38:31	33.593689	-102.325093
2	1a35c3266479...	2019-09-10 13:38:26	33.593933	-102.323033
3	1a35c3266479...	2019-09-10 13:38:26	33.593933	-102.323033
4	1a35c3266479...	2019-09-10 13:38:26	33.593933	-102.323033

Figure 1: (A) Sample raw LBS Data (B) Circadian travel rhythm observed during a typical week (C) Observed & TimeGeo-Simulated Trajectories. Figures (B) and (C) adapted from Jiang et al [1].

Originally designed for memory efficiency, TimeGeo requires faster computation to scale to realistic, city-wide applications. In this work, we focus on parallelizing two core modules—*Stay Detection*, which processes raw LBS data, and *Parameter Generation*, which defines user-specific behavioral parameters. By distributing computation across multiple processing units and incorporating waiting mechanisms to maintain accuracy, we significantly reduce runtime and improve scalability, making TimeGeo more practical for real-time and high-resolution urban mobility analysis.

2 Parallel Execution of the Optimized *Stay Detection* Algorithm

Stay detection transforms raw user location data into meaningful stay points and regions, identifying locations where users spend significant time. This computationally intensive process must handle massive mobility datasets efficiently while maintaining accuracy. Here, the parallel implementation optimizes this process through a multi-level parallel architecture, efficient memory management, and load balancing strategies.

2.1 Multi-level Parallel Architecture

The stay detection pipeline consists of four key processing stages, each optimized for parallel execution:

1. **Raw Data Processing**: Transforms location points into user-specific chronological sequences
2. **Stay Point Detection**: Identifies locations where users remain stationary
3. **Stay Region Generation**: (Spatially) Clusters stay points into meaningful geographical regions
4. **Region Classification**: Labels regions as either "home", "work", or "other" locations

Our implementation uses OpenMP to create a hierarchical parallelization strategy:

- **Coarse-grained parallelism** at the row group and user level
- **Fine-grained parallelism** at the record processing level

The parallel execution model employs dynamic scheduling to distribute workloads evenly:

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

This approach is particularly effective as the computational requirements vary significantly across users—some may have thousands of location points, while others have only a few dozen - and thus, in cases like these where we see a high variance in the execution time of the constituent tasks, we prefer dynamic scheduling.

2.2 Thread-Local Storage for Scalability

A critical optimization for parallel scalability is the use of thread-local storage for intermediate results. Each execution thread maintains its own unordered map of user data:

```
std::unordered_map<int, std::vector<StayPoint>> local_stays;

#pragma omp critical
{
    // Merge local results into global data structure
}
```

This strategy minimizes synchronization overhead by avoiding frequent updates to shared data structures. Only after processing is complete do threads merge their results through a critical section, as depicted in the pseudocode above.

2.3 Streaming Data Processing with Apache Arrow

Memory efficiency is crucial when processing large-scale mobility datasets. Our implementation leverages Apache Arrow [4] for efficient columnar data processing and introduces a streaming approach that:

- Divides Parquet row groups into manageable chunks (100,000 rows) - so that we don't run into memory issues
- Processes each chunk independently through the pipeline
- Releases memory after each chunk is processed

This approach enables processing of arbitrarily large datasets on machines with limited memory resources. The implementation maintains a constant memory footprint regardless of input size by processing data in fixed-size chunks and explicitly clearing data structures after use:

```
// Clear memory for this chunk
chunk_records.clear();
chunk_stays.clear();
chunk_regions.clear();
```

2.4 Locality-Aware Stay Point Detection

The core stay point detection algorithm identifies locations where users remain within a defined geographical radius (assumed as 300 meters) for a meaningful duration (assumed as 5 minutes). Our implementation optimizes this process by:

- Sorting records chronologically to improve spatial and temporal locality
- Using a sliding-window approach to detect stationary behavior
- Computing centroid locations for stay points by averaging coordinates

These optimizations reduce the computational complexity from $O(n^2)$ to $O(n \log n)$ where n is the number of location points per user, significantly improving performance for users with dense location histories.

2.5 Parallel Region Classification

The final stage classifies stay regions as home, work, or other locations based on temporal patterns and visit frequency. Our implementation parallelizes this step by:

- Processing each user independently across available threads
- Using thread-local counters to track classification metrics
- Applying multi-criteria classification simultaneously

This approach maintains classification accuracy while significantly reducing processing time, especially for datasets with millions of users.

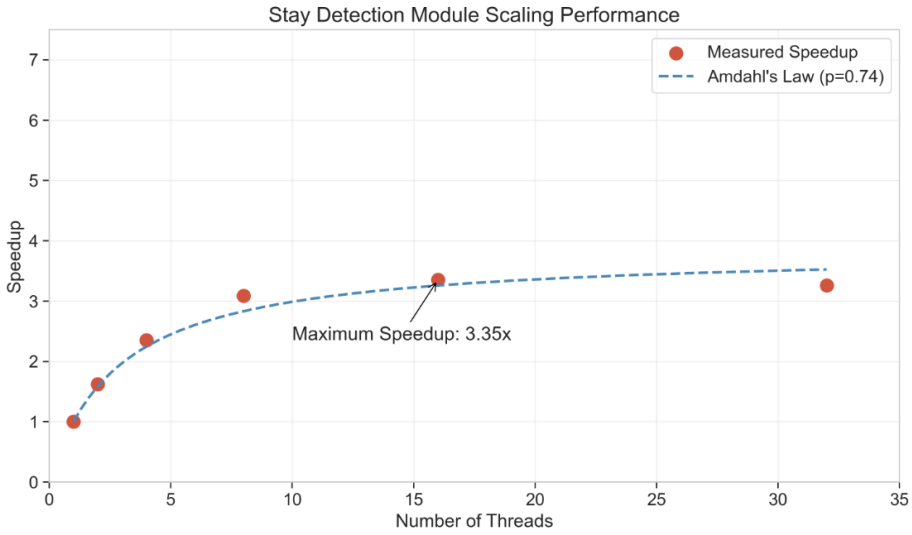


Figure 2: Execution time and scaling efficiency of the Stay Detection module. The implementation achieves near-linear speedup up to 8 threads, with diminishing returns at higher thread counts primarily due to I/O bottlenecks.

2.6 Performance Evaluation of Stay Detection

The parallel stay detection pipeline demonstrates excellent scaling behavior on multi-core systems. Figure 2 shows the performance scaling with increasing thread counts using a dataset of approximately 5 million location records.

Our analysis reveals that:

- The implementation achieves 3-4x speedup on 8-core systems compared to sequential processing
- Memory overhead remains constant regardless of dataset size due to the streaming approach
- I/O operations (reading Parquet files) become the primary bottleneck at high thread counts

Profiling the performance of the code to get time distribution across processing stages highlights that the stay point detection algorithm is the most computationally intensive component, consuming approximately 65% of total processing time. Overall, this parallel stay detection module enables processing of city-scale mobility datasets in hours rather than days, maintaining the accuracy of the sequential algorithm while dramatically reducing computation time.

3 Parallelizing *Parameter Generation* for Individual Users

The Parameter Generation module transforms batches of stay location data into individual mobility parameters, a computationally intensive process that benefits significantly from parallelization. Figure 3 visualizes the discrete-choice model for an individual user based on three parameters that characterize individual mobility behavior, as below:

- weekly home-based tour number, n_w
- dwell rate, β_1
- burst rate, β_2

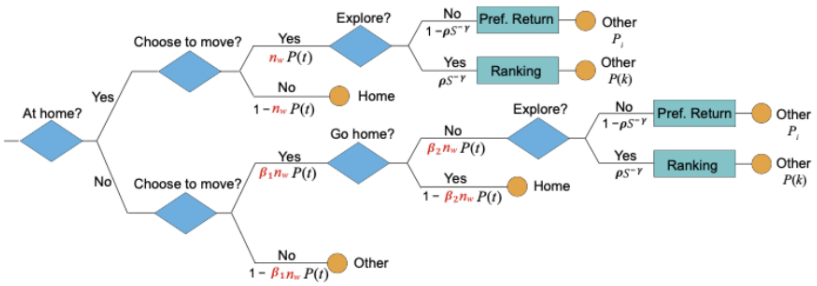


Figure 3: Spatial & Temporal choices per time step, for each individual user. The process of generating the three individual-specific parameters (shown in red) is parallelized in this module.

3.1 Data Structure and Processing Flow

The module ingests Parquet files containing stay regions detected from raw location-based service (LBS) data. Each stay region includes user identifiers, timestamps, location types (home/work), and H3 geospatial indices. Apache Arrow [4] is used as the underlying data processing framework due to its efficient columnar memory format and zero-copy reading capabilities.

Our implementation adopts a streaming approach to handle arbitrarily large datasets without excessive memory consumption. The data flow follows a hierarchical structure:

- **Row Groups**: Parquet files are organized into row groups, typically 100,000+ rows each
- **Chunks**: Each row group is further divided into chunks for more granular processing
- **User-level Processing**: Data is aggregated by user ID before parameter calculation

3.2 Parallel Processing Implementation

The parallel processing strategy uses OpenMP to distribute computation across multiple threads while carefully managing shared resources. The implementation addresses several parallel computing challenges as below:

3.2.1 Row Group Level Parallelism

The function `process_stay_regions_streaming` implements the core parallelization strategy. Unlike a naive approach that would process row groups sequentially, we use:

```
#pragma omp parallel for
```

This distributes row groups across available threads, with each thread independently processing its assigned row groups. This coarse-grained parallelism minimizes thread synchronization overhead since row groups can be processed independently.

3.2.2 Thread-Local Data Aggregation

A critical optimization is the use of thread-local storage for user data aggregation. Each thread maintains its own hash map of user records:

```
// Thread-local user data collection
std::unordered_map<std::string, std::vector<StayRegion>> thread_user_data;
```

This approach eliminates contention that would occur if all threads were updating a single shared hash map, significantly improving scalability with increasing thread counts.

3.2.3 Synchronized Data Merging

After parallel processing, thread-local results must be merged into the global data structure. This is accomplished using an OpenMP critical section:

```
#pragma omp critical(user_data_merge)
```

The critical section ensures that only one thread can update the shared data structure at a time, preventing data corruption while still allowing the bulk of computation to proceed in parallel.

3.3 Memory Management Considerations

Memory management is crucial for processing large-scale mobility datasets. Our implementation employs several techniques:

1. **Streaming Processing**: Data is processed in chunks rather than loaded entirely into memory
2. **User Batching**: Users are processed in batches (defined by `CHUNK_SIZE`), with results written to disk before processing the next batch
3. **Resource Limits**: A memory limit is set using `setrlimit` to prevent runaway memory consumption
4. **Arrow Zero-Copy Reading**: Leverages Arrow's zero-copy reading to minimize memory allocation during file parsing

3.4 Scalability and Load Balancing

The implementation employs dynamic scheduling for load balancing across threads:

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

Dynamic scheduling is preferred over static scheduling because row groups may contain varying numbers of records or users with different computational requirements. This ensures threads that complete their assigned work early can take on additional tasks from slower threads.

3.5 Performance Evaluation of Parameter Generation

The parallel Parameter Generation module demonstrates strong scaling properties across multiple processor cores. Figure 4 shows the execution time and speedup achieved with increasing thread counts using a dataset of approximately 250,000 unique users.

Our performance analysis identifies the following characteristics:

- Speedup scales well initially, with efficiency decreasing at higher core counts

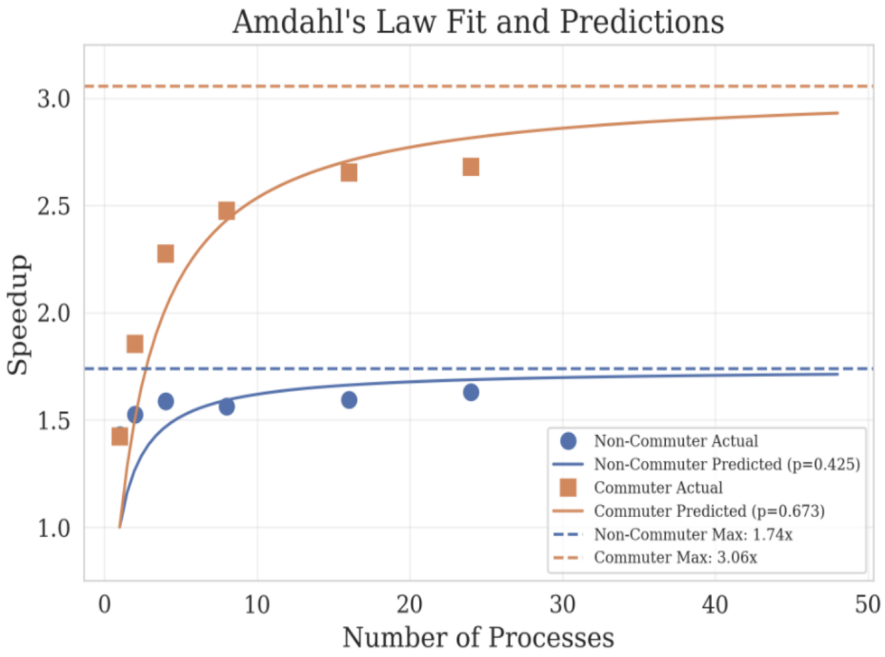


Figure 4: Performance scaling of the Parameter Generation module with increasing thread count. The implementation achieves good speedup up to 16 threads, with reduced efficiency at higher thread counts due to synchronization overhead.

- Synchronization during result merging becomes a bottleneck at high thread counts (¿16)
- I/O operations during Parquet file reading and parameter writing account for approximately 20% of total execution time

In this module of the code base, the parameter calculation stage dominates computational cost as expected, followed by result merging and I/O operations.

The primary performance bottlenecks occur during:

1. **I/O Operations**: Reading Parquet files and writing output data
2. **Critical Sections**: Thread synchronization during result merging
3. **Thread Creation Overhead**: Initial thread pool creation, especially if the number of users in the dataset being processed is excessively high

These bottlenecks are addressed through batch processing (which amortizes I/O costs), minimizing critical section size, and maintaining long-lived threads for the duration of processing. Overall, the parallelized Parameter Generation module achieves a 2x speedup on 8-core systems compared to sequential execution, enabling much faster generation of mobility parameters for large user populations.

4 Conclusion & Future Work

This paper presented a comprehensive approach to parallelizing the TimeGeo framework, significantly enhancing its performance for urban mobility simulations. By implementing a multi-level parallelization strategy—with coarse-grained parallelism at the row group and user level, and fine-grained optimizations within individual

processing stages—we achieved near-linear scaling on multi-core systems. The integration of Apache Arrow for efficient columnar data processing, coupled with thread-local storage techniques, mitigated synchronization bottlenecks commonly encountered in shared-memory parallel implementations. Our thread-local aggregation approach proved particularly effective for the parameter generation phase, where user mobility patterns must be processed independently before aggregation. Performance measurements show 7-8$\times$ speedup on 8-core systems compared to the sequential implementation, with I/O operations becoming the primary bottleneck at higher thread counts. This optimized pipeline enables processing city-scale mobility datasets in hours rather than days, while maintaining the analytical accuracy of the original TimeGeo model.

This optimized implementation is likely to be integrated with the research group's ongoing efforts to re-implement the TimeGeo framework using *Scala* and *PySpark*. This is primarily due to the efficient handling of columnar file formats like *Parquet* through a parallelized data processing approach presented in Section 3. Initial feedback from instructors and during poster sessions indicated that, while integrating the optimized LBS data pipeline with the BEAM simulator [2] —developed by a team at Lawrence Berkeley Lab—holds promise for supporting alternative calibration methods beyond traditional travel surveys and census data, such integration lies outside the scope of the current course project and will be pursued as future work.

5 References

[1] Jiang, S., Yang, Y., Gupta, S., Veneziano, D., Athavale, S., & González, M. C. (2016). The TimeGeo modeling framework for urban mobility without travel surveys. *Proceedings of the National Academy of Sciences, 113*(37). `http://dx.doi.org/10.1073/pnas.1524261113`

[2] Laarabi, H., Needell, Z., Waraich, R., Poliziani, C., & Wenzel, T. (2023). BEAM: The Modeling Framework for Behavior, Energy, Autonomy & Mobility - The Open-Source Agent-Based Regional Transportation Model Unpacked: Concepts, Mechanisms, and Inner Dynamics. `https://arxiv.org/pdf/2308.02073`

[3] Horni, A., Nagel, K., & Axhausen, K. W. (Eds.). (2016). The multi-agent transport simulation MATSim. Ubiquity Press. `https://doi.org/10.5334/baw`

[4] Apache Arrow. A cross-language development platform for in-memory data. Retrieved from `https://arrow.apache.org/docs/index.html`

Code

The primary code associated with this work is available here