# In-Memory Distance Threshold Similarity Searches on Moving Object Trajectories

Michael Gowanlock
Department of Information and Computer Sciences and
NASA Astrobiology Institute
University of Hawai‘i, Honolulu, HI, U.S.A.
Email: gowanloc@hawaii.edu

Henri Casanova
Department of Information and Computer Sciences
University of Hawai‘i, Honolulu, HI, U.S.A.
Email: henric@hawaii.edu

***Abstract*—The need to query spatiotemporal databases that store trajectories of moving objects arises in a broad range of application domains. In this work, we focus on in-memory distance threshold searches which return all moving objects that are found within a given distance $d$ of a fixed or moving object over a time interval. We propose algorithms to solve such searches efficiently, using an R-tree index to store trajectory data and two methods for filtering out trajectory segments so as to reduce segment processing time. We evaluate our algorithms on both real-world and synthetic in-memory trajectory datasets. Choosing an efficient trajectory splitting strategy to reduce index resolution increases the efficiency of distance threshold searches. Moreover, we demonstrate that distance threshold searches can be performed in parallel using a multithreaded implementation and we observe that high parallel efficiency (72.2%-85.7%) can be obtained. Interestingly, the traditional notion of considering good trajectory splits by minimizing the volume of hyperrectangular minimum bounding boxes (MBBs) so as to reduce index overlap is not well-suited to improve the performance of in-memory distance threshold searches.**

***Keywords*-query optimization; query parallelization; spatiotemporal databases; trajectory searches.**

## I. Introduction

Moving object databases (MODs) have gained attention as applications in several domains analyze trajectories of moving objects. Some examples include the movement patterns of animals in ecology studies, vehicles in traffic monitoring applications, stellar bodies in astrophysical simulations and the movement of objects in many other domains. Contributing to the motivation for MOD research is the proliferation of mobile devices that provide location information such as GPS tracking, which can provide short-term information to users, or whose data can be stored for subsequent processing. Regardless of how trajectory data is generated, trajectory similarity searches are used to gain insight into application domains, which attempt to find similarities between the properties of trajectories. Such similarities could be due to trajectories that cluster together over a time interval, or to trajectories with similar spatial properties, such as being short, or long, over a time interval. We focus on MODs that store historical trajectories [1], [2], [3], [4], [5], and that must support searches over subsets, or perhaps the full set, of the trajectory histories. In particular, we focus on two types of trajectory similarity searches, which we term *distance threshold searches*:

1) Find all trajectories within a distance $d$ of a given static point over a time interval $[t_0, t_1]$.
2) Find all trajectories within a distance $d$ of a given trajectory over a time interval $[t_0, t_1]$.

An example query of the first type would be to find all animals within a distance $d$ of a water source within a day. An example query of the second type would be to find all police vehicles on patrol within a distance $d$ of a moving stolen vehicle during an afternoon. We investigate efficient distance threshold searches on MODs, making the following contributions:

- We propose algorithms to solve the two types of in-memory distance threshold searches above.
- We make the case for using an in-memory R-tree index for storing trajectory line segments.
- Given a set of candidate line segments returned from the R-tree, we propose methods to filter out line segments that are not part of the query result set.
- We propose decreasing index resolution to exploit the trade-off between the volume occupied by the trajectories, the amount of index overlap, the number of entries in the index, and the number of candidate trajectory segments to process by exploring three trajectory splitting strategies.
- We demonstrate that, for in-memory searches, lower-bounding the index resolution is more important than minimizing the volume of MBBs, and thus index overlap.
- We parallelize the distance threshold search in a shared-memory environment using OpenMP and show that high parallel efficiency can be achieved.
- We evaluate our proposed algorithms using both real-world and synthetic datasets for both 3-D and 4-D trajectory data (i.e., the temporal dimension plus either 2 or 3 spatial dimensions).

This paper is organized as follows. Section II discusses related work. Section III defines the distance threshold search problem. Section IV discusses the indexing method. Section V details our algorithms. Section VI presents results from an initial performance evaluation of the two distance threshold searches outlined above. Section VII motivates, proposes, and evaluates methods for filtering the candidate line segments. Section VIII presents and evaluates methods for splitting trajectories to reduce index resolution for efficient query processing. Finally, Section IX concludes the paper with a brief summary of findings and perspectives on future work.

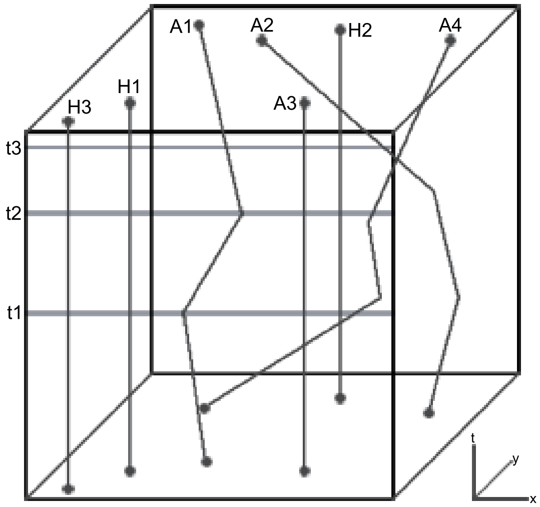


Figure 1. An illustration of trajectories and associated searches. Hospitals (H), ambulances (A), and time intervals between $t_0$ and $t_4$ are shown. Note that the vertical dimension refers to time in this example.

## II. Related Work

A trajectory is a set of points traversed by an object over time in Euclidean space. In MODs, trajectories are stored as sets of spatiotemporal line segments. The majority of the literature on indexing spatiotemporal data utilizes R-tree data structures [6]. An R-tree indexes spatial and spatiotemporal data using MBBs. Each trajectory segment is contained in one MBB. Leaf nodes in the R-tree store pointers to MBBs and the segments they contain (coordinates, trajectory id). A non-leaf node stores the dimensions of the MBB that contains all the MBBs stored (at the leaf nodes) in the non-leaf node's sub-tree. Searches traverse the tree to find all (leaf) MBBs that overlap with a query MBB. Variations of the R-tree and systems have been proposed for efficient trajectory processing, such as TB-trees [7], STR-trees [7], 3DR-trees [8], SETI [9], SECONDO [10] and TrajStore [11].

In what follows we first review work on $k$ Nearest Neighbors ($k$NN) searches. Although other types of searches have been studied (e.g., flocks [12], convoys [5], swarms [13]), $k$NN searches are the most related to distance threshold searches. We then review the (scarce) related work on distance threshold searches. Finally, we review work on the parallelization of searches in the MOD literature.

### A. Nearest Neighbor Searches in Spatiotemporal Databases

Our work is related to, but as explained in later sections, also has major differences with the spatiotemporal $k$NN literature. We illustrate the typical $k$NN searches in Figure 1.

$Q1$ Find the nearest hospital to hospital $H_1$ during the time interval [$t_0$,$t_4$], which results in hospital $H_3$.

$Q2$ Find the nearest hospital to ambulance $A_1$ during the time interval [$t_0$,$t_1$], which results in hospital $H_1$.

$Q3$ Find the nearest ambulance to ambulance $A_2$ during the time interval [$t_1$,$t_4$], which results in ambulance $A_4$.

$Q4$ Find the nearest ambulance to ambulance $A_4$ *at any instant* in the time interval [$t_0$,$t_4$]; this results in multiple ambulances, since the query is continuous: ambulance $A_3$ in the interval [$t_0$,$t_1$), ambulance $A_2$ in the interval [$t_1$,$t_3$) and ambulance $A_3$ in the interval [$t_3$,$t_4$].

The example searches above are representative of the four main types of $k$NN searches that have been studied in the literature. The first type of search finds the nearest stationary data object to a static query object. An example is $Q1$ above. In [14], the authors propose a method that relies on the R-tree to perform NN searches, and then generalize their approach to handle $k$NN searches. The static objects are contained within a MBB. To process a search, MBBs are selected and then accessed from the R-tree to determine if a candidate NN is contained therein. To find the nearest neighbor two possible distance metrics are proposed: *MINDIST* and *MINMAXDIST*. These metrics are used for ordering and pruning the R-tree search.

The next type of search is moving query and static data, or the Continuous Nearest Neighbor (CNN) query [15], [16]. An example is $Q2$ above. In this search, the ambulance (or moving query) is continuously changing position relative to the hospitals; therefore, depending on the route, traffic, and other factors, the shortest distance to a hospital may change. The method in [15] employs a sampling technique on moving query objects, where query points are interpolated in between two sampled positions. The accuracy of this method is dependent on the sampling rate, which has the effect of making the method computationally expensive and can potentially return the wrong result. The CNN method developed by [16] avoids the computationally expensive drawbacks of [15]. Both works use an R-tree index.

The third type of search is moving query and moving data ($Q3$ above). There has been a considerable amount of work on this type of search (see for instance [17], [18], [19], [20], [21] for works published on this topic since 2005).

The last type of search is continuous moving query and continuous moving data (trajectories), which is the most related to this work. In $Q4$ above, multiple data points are returned as objects change over the time interval, in contrast to $Q3$ which is not continuous. These types of historical continuous searches have been investigated in [22], [23], [24], [10]. In comparison to the other NN variants, these searches propose new challenges: i) they are historical, meaning large segments can be processed; ii) they are continuous so that the candidate set changes over the time interval of the query. Opportunities arise for ordering and pruning the search efficiently, leading to several proposed new indexing techniques such as the TB-tree [7], the STR-tree [7], the 3DR-tree [8] and SETI [9].

The $k$NN search on historical continuous moving object trajectories is related to this work, but as detailed in Section IV, there are key differences that distinguish distance threshold searches from $k$NN searches. In particular, with distance threshold searches there is no possible pruning of the search space (while pruning is a main focus in the $k$NN literature).

### B. Distance Threshold Searches

A distance threshold search can be seen as a $k$NN query with an unknown value of $k$. As a result, previous work on $k$NN searches (with known $k$) cannot be applied directly to distance threshold searches (with unknown $k$). To the best of our knowledge, other than our previous work [1], which we extend in this paper, the work in [4] is the only other work on distance threshold searches. In [4] the authors propose sequential, out-of-core solutions to the distance threshold search. They compare the performance of four implementations, one that is based on an R-tree index and three that utilize different plane-sweep approaches, and find that an adaptive plane-sweep approach performs the best in several experimental scenarios.

A difference between our work and that in [4], is that they only return whether trajectories are within a query distance $d$ of query trajectory, and not the time interval within which this occurs. For example, over the time period [0,1], one trajectory may be within the query distance in the interval [0.1,0.2], and another trajectory within the interval between [0.4,0.9]. In contrast, our approach does provide these time intervals as this information can be useful in many application domains. One such example will be given in Section III-A. A more striking difference between [4] and this work is that we focus on in-memory databases, with a focus on efficient trajectory indexing strategies.

### C. Parallelization of Searches in Spatial and Spatiotemporal Databases

The majority of the work in the literature on spatiotemporal databases has been in the context of out-of-core implementations where part of the database resides in memory and part of it on disk. Not much attention has been given to the parallelization of spatiotemporal similarity searches. To the best of our knowledge, the parallelization of distance threshold searches on moving object trajectories has not been investigated. In what follows we review relevant previous work on the parallelization of other searches.

The work in [25] provides a parallelization approach for finding patterns in a set of trajectories. The main contribution is an efficient way to decompose the computation and assign the trajectories to processors, so as to minimize computation and decrease communication costs. In [26], the authors propose a parallel solution for mining trajectories to find frequent movement patterns, or T-patterns [27]. They utilize the MapReduce framework in combination with a multi-resolution hierarchical grid to find patterns within the trajectory data. The importance of having multiple resolutions is that the level of resolution determines the types of patterns that may be found with the pattern identification algorithm. In [28], the authors propose two algorithms to search for the $k$ most similar trajectories to a query trajectory where the trajectory database is distributed across a number of nodes. Their approach attempts to perform the similarity search, such that all of the relevant trajectory segments most similar to the query trajectory do not need to be sent across the network for evaluation, thereby reducing communication overhead. Finally, the work in [29] examines various indexing techniques for spatial and spatiotemporal data for use in the context of multi-core CPUs and many-core GPUs. Interestingly, the authors suggest that the traditional indexing techniques used for sequential executions may not be well-suited to emerging computer architectures.

## III. Problem Statement

### A. Motivating Example

One motivation application for this work is in the area of astrobiology/astronomy [30]. The field of astrobiology studies the origin, evolution, distribution, and future of life in the universe. The study of the habitability of the Earth suggests that life can exist in a multitude of environments. The past decade of exoplanet searches implies that the Milky Way, and hence the universe, hosts many rocky, low mass planets that may sustain complex life. The Galactic Habitable Zone is thought to be the region(s) of the Galaxy that may favor the development of complex life. With regards to long-term habitability, some of these regions may be inhospitable due to transient radiation events, such as supernovae explosions or close encounters with flyby stars that can gravitationally perturb planetary systems. Studying habitability thus entails solving the following two types of *distance threshold searches* on the trajectories of (possibly billions of) stars orbiting the Milky Way: (i) Find all stars within a distance $d$ of a supernova explosion, i.e., a non-moving point over a time interval; and (ii) Find the stars, and corresponding time periods, that host a habitable planet and are within a distance $d$ of all other stellar trajectories. These are exactly the distance threshold searches that we study in this work. Solving these searches for all stars with habitable planets will result in finding the duration of time flyby stars are within a distance threshold, for example 0.1 parsecs, of a habitable planetary system that is orbiting the Milky Way. As mentioned in Section II-B, it is useful to know the time intervals in which trajectories are within the query distance $d$. In this application, there is a difference between a flyby star that is only within the query distance for a short time interval, in comparison to a longer time interval, where the latter will mean a greater gravitational effect between the flyby star and planetary system. Additionally, the point distance search will find all stars hosting habitable planets sufficiently nearby a supernova such that the planets lose a significant fraction of the ozone in their atmospheres.

### B. Problem Definition

Let $D$ be a database of trajectories, where each trajectory $T_i$ consists of $n_i$ 4D (3 spatial + 1 temporal) line segments. Each line segment is defined by the following attributes: $x_{start}$, $y_{start}$, $z_{start}$, $t_{start}$, $x_{end}$, $y_{end}$, $z_{end}$, $t_{end}$, trajectory id, and segment id. These coordinates for each segment define the segment's MBB (note that the temporal dimension is treated in the same manner as the spatial dimensions). Linear interpolation is used to answer searches that lie between $t_{start}$ and $t_{end}$ of a given line segment.

We consider historical continuous *searches* for trajectories within a distance $d$ of a *query* $Q$, where $Q$ is a moving object's trajectory, $Q_t$, or a stationary point, $Q_p$. More specifically:

- DistTrajSearch_Q$_p$($D$,$Q_p$,$Q_{start}$,$Q_{end}$, $d$) searches $D$ to find all trajectories that are within a distance $d$ of a given query static point $Q_p$ over the query time period [$Q_{start}$,$Q_{end}$]. The query is continuous, such that the trajectories found may be within the distance threshold

$d$ for a subinterval of the query time [$Q_{start}$,$Q_{end}$]. For example, for a query $Q_1$ with a query time interval of [0,1], the search may return $T_1$ between [0.1,0.3] and $T_2$ between [0.2,0.6].

- DistTrajSearch_Q$_t$($D$,$Q_t$,$Q_{start}$,$Q_{end}$, $d$) is similar but searches for trajectories that are within a distance $d$ of a query trajectory $Q_t$.

DistTrajSearch_Q$_p$ is a simpler case of DistTrajSearch_Q$_t$. We focus on developing an efficient approach for DistTrajSearch_Q$_t$, which can be reused as is for DistTrajSearch_Q$_p$. We present experimental results for both types of searches.

In all that follows, we consider *in-memory databases*, meaning that the database fits and is loaded in RAM once and for all. Distance threshold searches are relevant for scientific applications that are typically executed on high-performance computing platforms such as clusters. It is thus possible to partition the database and distribute it over a (possibly large) number of compute nodes so that the application does not require disk accesses. It is straightforward to parallelize distance threshold searches (replicate the query across all nodes, search the MOD independently at each node, and aggregate the obtained results). We leave the topic of distributed-memory searches for future work. Instead we focus on efficient in-memory processing at a single compute node, which is challenging and yet necessary for achieving efficient distributed-memory executions. Furthermore, as explained in Section IV, no criterion can be used to avoid index tree node accesses in distance threshold searches. Therefore, there are no possible I/O optimizations when (part of) the database resides on disk, which is another reason why we focus on the in-memory scenario.

## IV. Trajectory Indexing

Given a distance threshold search for some query trajectory over some temporal extent, one considers all relevant query MBBs (those for the query trajectory segments). These query MBBs are *augmented* in all spatial dimensions by the threshold distance $d$. One then searches for the set of trajectory segment MBBs that overlap with the query MBBs, since these segments may be in the result set. Efficient indexing of the trajectory segment MBBs can thus lower query response time.

The most common approach is to store trajectory segments as MBBs in an index tree [22], [23], [24], [10]. Several index trees have been proposed (TB-tree [7], STR-tree [7], 3DR-tree [8]). Their main objective is to reduce the number of tree nodes visited during index traversals, using various pruning techniques (e.g., the *MINDIST* and *MINMAXDIST* metrics in [14]). While this is sensible for $k$NN searches, instead for distance threshold searches *there is no criterion for reducing the number of tree nodes that must be traversed*. This is because any MBB in the index that overlaps the query MBB may contain a line segment within the distance threshold, and thus must be returned as part of the candidate set.

Let us consider for instance the popular TB-tree, in which a leaf node stores only contiguous line segments that belong to the same trajectory and leaf nodes that store segments from the same trajectory are chained in a linked list. As a result, the TB-tree has high “temporal discrimination” (this terminology was introduced in [7]). Figure 2 shows a trajectory stored inside four leaf nodes within a TB-tree (each leaf node is shown as a bounding box). The curved and continuous appearance of the trajectory is because multiple line segments are stored together in each leaf node. By contrast, the R-tree simply stores in each leaf node trajectory segments that are spatially and temporally near each other, regardless of the individual trajectories. Figure 3 depicts an example with 4 segments belonging to 3 different trajectories that could be stored in a leaf node of an R-tree. For a distance threshold search, the number of TB-tree leaf nodes processed to perform the search could be arbitrarily high (since segment MBBs from many different trajectories can overlap the query MBB). Therefore, the TB-tree reduces the important R-tree property of overlap reduction; with an R-tree it may be sufficient to process only a few leaf nodes since each leaf node stores spatially close segments from multiple trajectories. For distance threshold searches, high spatial discrimination is likely to be more efficient than high temporal discrimination. Also, results in [7] show that the TB-tree performs better than the R-tree (for $k$NN searches) especially when the number of indexed entries is low; however, we are interested in large MODs (see Section III-A). We conclude that an R-tree index should be used for efficient distance threshold search processing.

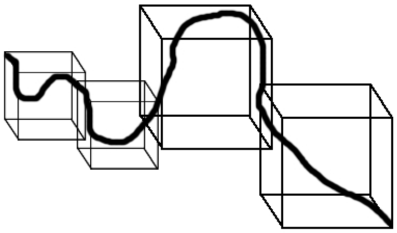


Figure 2. An example trajectory stored in different leaf nodes in a TB-tree.

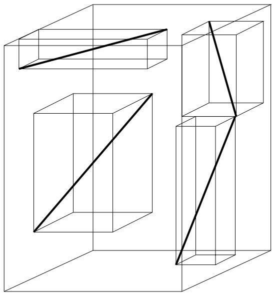


Figure 3. Four line segments belonging to three different trajectories within one leaf node of an R-tree.

## V. Search Algorithm

We propose an algorithm, TrajDistSearch (Figure 4), to search for trajectories that are within a threshold distance of a query trajectory (defined as a set of connected trajectory segments over some temporal extent). All entry MBBs that overlap the query MBB are returned by the R-tree index and are then processed to determine the result set. More specifically, the algorithm takes as input an R-tree index, $T$, a query trajectory, $Q$, and a threshold distance, $d$. It returns a set of time intervals annotated by trajectory ids, corresponding to the interval of time during which a particular trajectory is within distance $d$ of the query trajectory. After initializing the result set to the empty set (line 2), the algorithm loops over

```
1: procedure TRAJDISTSEARCH (R-tree T, Query Q, double d)
2:     resultSet $\leftarrow \emptyset$
3:     for all querySegmentMBB in Q.MBBSet do
4:         candidateSet $\leftarrow$ T.Search(querySegmentMBB, d)
5:         for all candidateMBB in candidateSet do
6:             (entrySegment, querySegment) $\leftarrow$ interpolate(
                       candidateMBB,querySegmentMBB)
7:             timeInterval $\leftarrow$ calcTimeInterval(
                       entrySegment,querySegment,d)
8:             if timeInterval $\neq \emptyset$ then
9:                 resultSet $\leftarrow$ resultsSet $\cup$ timeInterval
10:            end if
11:        end for
12:    end for
13:    return resultSet
14: end procedure
```

Figure 4. Pseudo-code for the TRAJDISTSEARCH algorithm (Section V).

all (augmented) MBBs that correspond to the segments of the query trajectory (line 3). For each such query MBB, the R-tree index is searched to obtain a set of candidate entry MBBs that overlap the query MBB (line 4). The algorithm then loops over all the candidates (line 5) and does the following. First, given the candidate entry MBB and the query MBB, it computes an entry trajectory segment and a query trajectory segment that span the same time interval (line 6). The algorithm then computes the interval of time during which these two trajectory segments are within a distance $d$ of each other (line 7). This calculation involves computing the coefficients of and solving a degree two polynomial [10]. If this interval is non-empty, then it is annotated with the trajectory id and added to the result set (line 9). The overall result set is returned once all query MBBs have been processed (line 13). Note that for a static point search Q.MBBSet (line 3) would consist of a single (degenerate) MBB with a $d$ extent in all spatial dimensions and some temporal extent, thus obviating the need for the outer loop. We call this simpler algorithm POINTDISTSEARCH.

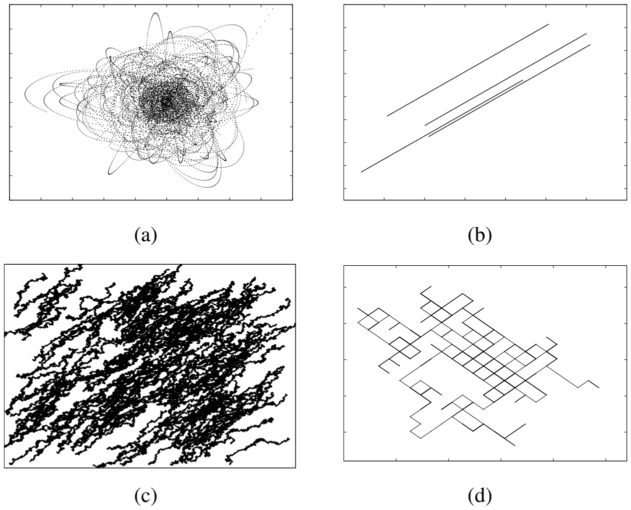
## VI. INITIAL EXPERIMENTAL EVALUATION

### A. Datasets

Our first dataset, *Trucks* [31], is used in other MOD works [23], [24], [10]. It contains 276 trajectories corresponding to 50 trucks that travel in the Athens metropolitan area for 33 days. This is a 3-dimensional dataset (2 spatial + 1 temporal). Our second dataset is a class of 4-dimensional datasets (3 spatial + 1 temporal), *Galaxy*. These datasets contain the trajectories of stars moving in the Milky Way's gravitational field (see Section III-A). The largest *Galaxy* dataset consists of 1,000,000 trajectory segments corresponding to 2,500 trajectories of 400 timesteps each. Distances are expressed in kiloparsecs (kpc). Our third dataset is a class of 4-dimensional synthetic datasets, *Random*, with trajectories generated via random walks. An adjustable parameter, $\alpha$, is used to control whether the trajectory is a straight line ($\alpha = 0$) or a Brownian motion trajectory ($\alpha = 1$). We vary $\alpha$ in 0.1 increments to produce 11 datasets for datasets containing between $\sim$1,000,000 and $\sim$5,000,000 segments. Trajectories with $\alpha = 0$ spans the largest spatial extent and trajectories with $\alpha = 1$ are the most localized. All trajectories have the same temporal extent but different start times. Other synthetic datasets exist, such as GSTD [32]. We do not use GSTD because it does not allow for 3-dimensional spatial trajectories.

TABLE I. CHARACTERISTICS OF DATASETS

| Dataset | Trajec. | Entries |
|---|---|---|
| Trucks | 276 | 112152 |
| *Galaxy*-200k | 500 | 200000 |
| *Galaxy*-400k | 1000 | 400000 |
| *Galaxy*-600k | 1500 | 600000 |
| *Galaxy*-800k | 2000 | 800000 |
| *Galaxy*-1M | 2500 | 1000000 |
| *Random*-1M ($\alpha \in \{0, 0.1, \ldots, 1\}$) | 2500 | 997500 |
| *Random*-2M ($\alpha = 1$) | 5000 | 1995000 |
| *Random*-3M ($\alpha = 1$) | 7500 | 2992500 |
| *Random*-4M ($\alpha = 1$) | 10000 | 3990000 |
| *Random*-5M ($\alpha = 1$) | 12500 | 4987500 |


(a)

(b)

(c)

(d)

Figure 5. (a) *Galaxy* dataset: a sample of 30 trajectories, (b) 4 trajectories in the *Random* dataset with $\alpha = 0$, (c) 200 trajectories in the *Random* dataset with $\alpha = 0.8$, (d) a sample trajectory in the *Random* dataset with $\alpha = 1$.

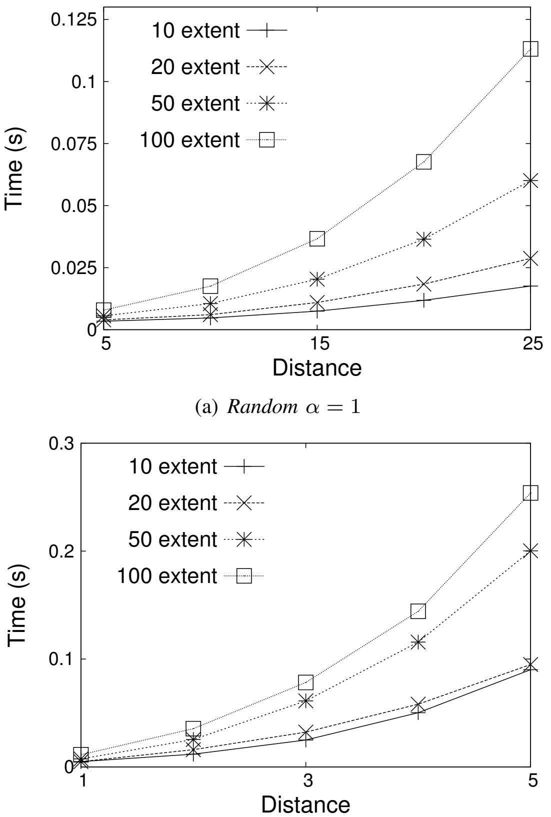
Figure 5 shows a 2-D illustration of the *Galaxy* and *Random* datasets. An illustration of *Trucks* can be found in previous works [23], [24]. Table I summarizes the main characteristics of each dataset. Our *Galaxy* and *Random* datasets are publicly available [33].

### B. Experimental Methodology

We have implemented algorithm TRAJDISTSEARCH in C++, reusing an existing R-tree implementation based on that initially developed by A. Guttman [6], and the code is publicly available [34]. We execute the sequential implementation on one core of a dedicated Intel Xeon X5660 processor, at 2.8 GHz, with 12 MB L3 cache and sufficient memory to store the entire index. In the multithreaded implementation, we show results up to 6 threads, which corresponds to the 6 cores on the CPU on the platform. We measure query response time averaged over 3 trials. The variation among the trials is negligible so that error bars in our results are not visible. We ignore the overhead of loading the R-tree from disk into memory, which can be done once before all query processing.

### C. Static Point Search Performance

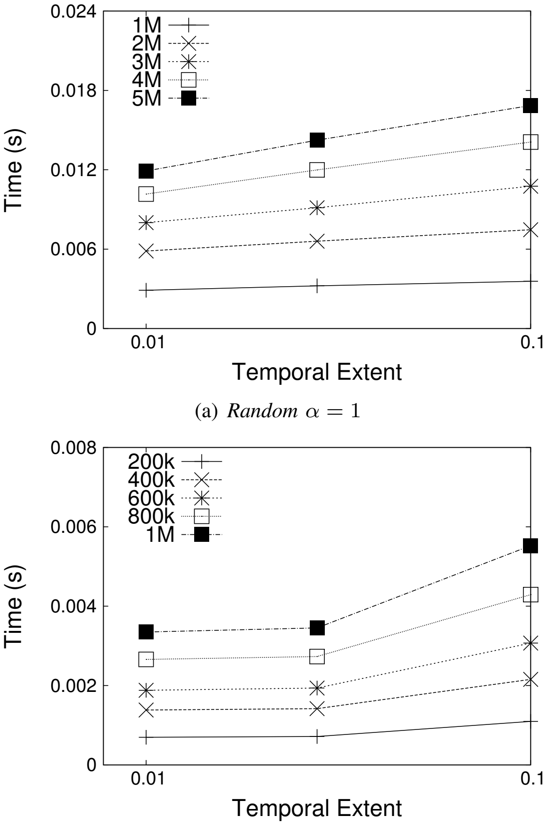
In this section, we assess the performance of POINTDISTSEARCH with the following searches:



(a) *Random* $\alpha = 1$

(b) *Galaxy*-1M

Figure 6. Query response time vs. threshold distance for 10%, 20%, 50% and 100% of the temporal extents of the trajectories in the datasets. (a) P1 using the *Random*-1M $\alpha = 1$ dataset; (b) the *Galaxy*-1M dataset with P2 (b).



(a) *Random* $\alpha = 1$

(b) *Galaxy* datasets

Figure 7. Query response time vs. various temporal extents of the trajectories in the datasets. (a) P3 using the *Random*-1M $\alpha = 1$ datasets; (b) P4 using the *Galaxy* datasets.

- P1: From the *Random*-1M $\alpha = 1$ dataset, 500 random points are selected with 10%, 20%, 50% and 100% of the temporal extent of the trajectories in the dataset, for various query distances.
- P2: Same as P1 but for the *Galaxy*-1M dataset.
- P3: From the *Random*-1M, 2M, 3M, 4M, 5M $\alpha = 1$ datasets, 500 random points are selected with 1%, 5%, and 10% of the temporal extent of the trajectories in the dataset, with query distance $d = 5$.
- P4: Same as S3 but for the *Galaxy*-200k, 400k, 600k, 800k, 1M datasets, where query distance $d = 1$.

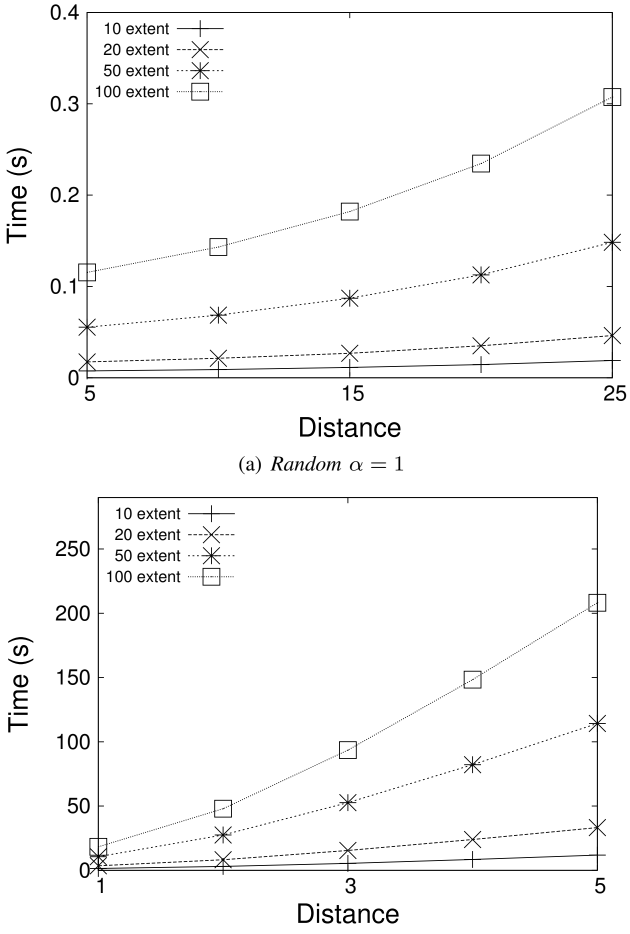
Figures 6 (a) and 6 (b) plot response time vs. query distance for P1 and P2 above. The response time increases superlinearly with the query distance and with the temporal extent. Figures 7 (a) and 7 (b) plot response time vs. temporal extent for P3 and P4 above, showing linear or superlinear growth in response time as the temporal extent increases. More specifically, Figure 7 (b) shows superlinear growth. This is because the trajectories in *Galaxy* are less constrained than in *Random*. We suspect that spatial under and overdensities of the trajectories in *Galaxy* may lead to searches that have qualitatively different behavior for different temporal extents. Next, we turn to our initial evaluation of trajectory searches.

### *D. Trajectory Search Performance*

We measure the query response time of TRAJDISTSEARCH for the following sets of trajectory searches:

- S1: *Random*-1M dataset, $\alpha = 1$, 100 randomly selected query trajectories, processed for 10%, 20%, 50% and 100% of their temporal extents, with various query distances.
- S2: Same as S1 but for the *Galaxy*-1M dataset.
- S3: *Random*-1M, 2M, 3M, 4M and 5M datasets, $\alpha = 1$, 100 randomly selected query trajectories, processed for 100% of their temporal extent, with various query distances.
- S4: *Galaxy*-200k, 400k, 600k, 800k, 1M datasets, 100 randomly selected trajectories, processed for 1%, 5% and 10% of their temporal extents, with a fixed query distance $d = 1$.

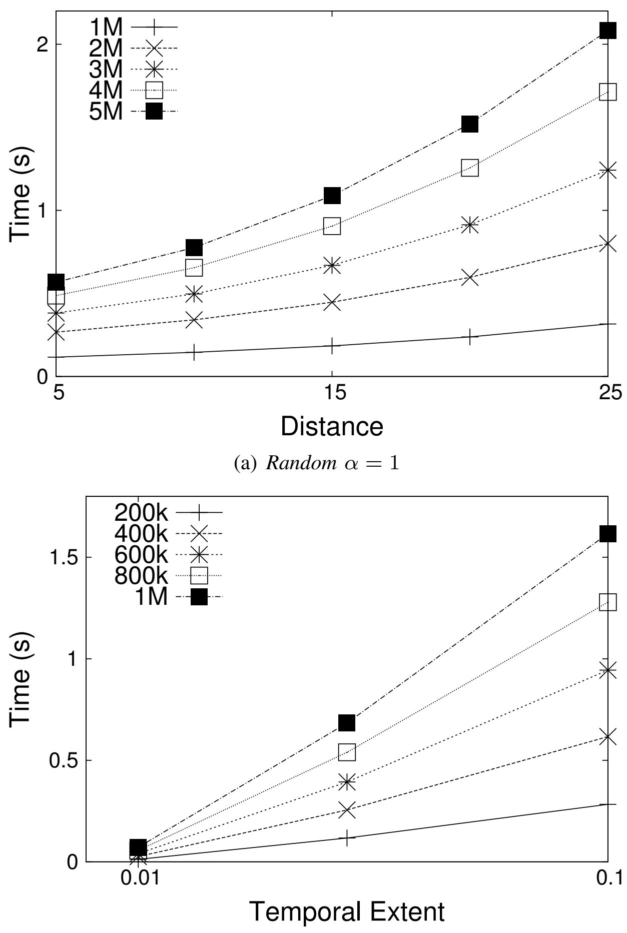
Figures 8 (a) and 8 (b) plot response time vs. query distance for S1 and S2 above. The response time increases slightly superlinearly with the query distance and with the temporal extents. In other words, the R-tree search performance degrades gracefully as the search is more extensive. Figures 9 (a) and (b) show response time vs. query distance and temporal extent respectively, for S3 and S4 above. The response time increases slightly superlinearly as the query distance increases for S3, and roughly linearly as the temporal



(a) *Random* $\alpha = 1$

(b) *Galaxy*-1M

Figure 8. Query response time vs. threshold distance for 10%, 20%, 50% and 100% of the temporal extents of trajectories. (a) S1 using the *Random*-1M $\alpha = 1$ dataset; (b) S2 using the *Galaxy*-1M dataset.



(a) *Random* $\alpha = 1$

(b) *Galaxy* datasets

Figure 9. (a) Response time vs. threshold distances for various numbers of segments in the index using search S3. (b) Response time vs. temporal extent for various numbers of segments in the index using search S4.

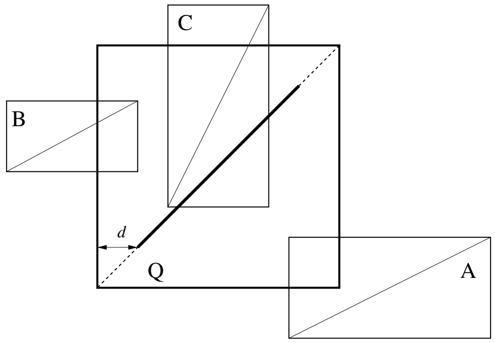


Figure 10. Three example entry MBBs and their overlap with a query MBB.

extent increases for S4. Both of these figures show results for various dataset sizes. An important observation is that the response time degrades gracefully as the datasets increase in size. More interestingly, note that for a fixed temporal extent and a fixed query distance, a larger dataset means a higher trajectory density, and thus a higher degree of overlap in the R-tree index. In spite of this increasing overlap, the R-tree still delivers good performance. These trends are expected, as we see the performance of the algorithm degrade with increasing query distance, temporal extent, or dataset size. In the next sections we address optimizations to reduce response time further.

## VII. TRAJECTORY SEGMENT FILTERING

The results in the previous section show that POINTDISTSEARCH and TRAJDISTSEARCH maintain roughly consistent performance behavior over a range of search configurations (temporal extents, threshold distances, index sizes). In this and the next section, we explore approaches to reduce response time, using TRAJDISTSEARCH as our target.

At each iteration our algorithm computes the moving distance between two line segments (line 7 in Figure 4). One can bypass this computation by "filtering out" those line segments for which it is straightforward (i.e., computationally cheap) to determine that they cannot possibly lie within distance $d$ of the query. This filtering is applied to the segments once they have been returned by the index, and is thus independent of the indexing method.

Figure 10 shows an example with a query MBB, $Q$, and three overlapping MBBs, $A$, $B$, and $C$, that have been returned from the index search. The query distance $d$ is indicated in the (augmented) query box so that the query trajectory segment is shorter than the box's diagonal. MBB $A$ contains a segment that is outside $Q$ and should thus be filtered out. The line segment in $B$ crosses the query box boundary but is never within distance $d$ of the query segment and should be filtered out. $C$ contains a line segment that is within a distance $d$ of the query segment, and should thus not be filtered out. For this segment a moving distance computation must be performed (Figure 4, line 7) to determine whether there is an interval of time in which the two trajectories are indeed within a distance $d$ of each other. The fact that candidate segments are returned that should in fact be ignored is inherent to the use of MBBs: a segment occupies an infinitesimal portion of its MBB's space. This issue is germane to MODs that store trajectories using MBBs.

In practice, depending on the dataset and the search, the number of line segments that should be filtered out can be large. Figure 11 shows the number of candidate segments returned by the index search and the number of segments that are within the query distance vs. $\alpha$, for the *Random*-1M dataset, with 100 randomly selected query trajectories processed for 100% of their temporal extent. The fraction of candidate segments that are within the query distance is below 16.5% at $\alpha = 1$. In this particular example, an ideal filtering method would filter out more than 80% of the line segments.

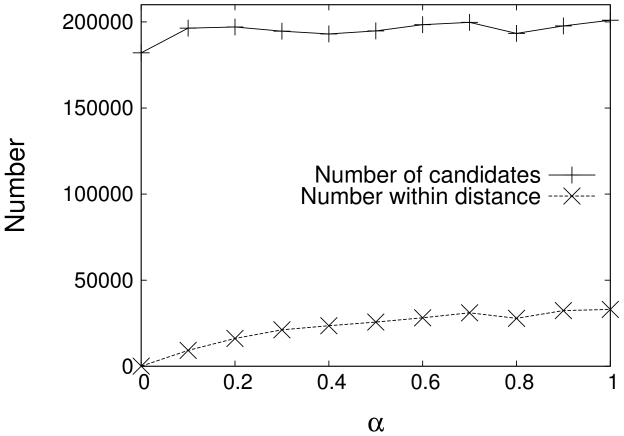


Figure 11. The number of moving distance calculations and the number that are actually within a distance of 15 vs. $\alpha$ in the *Random*-1M datasets.

### A. Two Segment Filtering Methods

After the query and entry line segments are interpolated so that they have the same temporal extent (Figure 4, line 6), various criteria may remove the candidate segment from consideration. We consider two filtering methods beyond the simple no filtering approach:

- **Method 1 –** No filtering.
- **Method 2 –** After the interpolation, check whether the candidate segment still lies within the query MBB. This check only requires floating point comparisons between spatial coordinates of the segment endpoints and the query MBB corners, and would occur between lines 6 and 7 in Figure 4. Method 2 would filter out A in Figure 10.
- **Method 3 –** Considering only 2 spatial dimensions, say $x$ and $y$, for a given query segment MBB compute the slope and the $y$-intercept of the line that contains the query segment. This computation requires only a few floating point operations and would occur in between lines 3 and 4 in Figure 4, i.e., in the outer loop. Then, before line 7, check if the endpoints of the candidate segment both lie more than a distance $d$ above or below the query trajectory line. In this case, the candidate segment can be filtered out. This check requires only a few floating point operations involving segment endpoint coordinates and the computed slope and $y$-intercept of the query line. Method 3 would filter out both A and B in Figure 10.

Other computational geometry methods could be used for filtering, but these methods must be sufficiently fast (i.e., low floating point operation counts) if any benefit over Method 1 is to be achieved. For instance, one may consider a method that computes the shortest distance between an entry line segment and the query line segment regardless of time, and discard the candidate segment if this shortest distance is larger than threshold distance $d$. However, the number of (floating point) operations to perform such filtering is on the same order as that needed to perform the full-fledge moving distance calculation.

### B. Filtering Performance

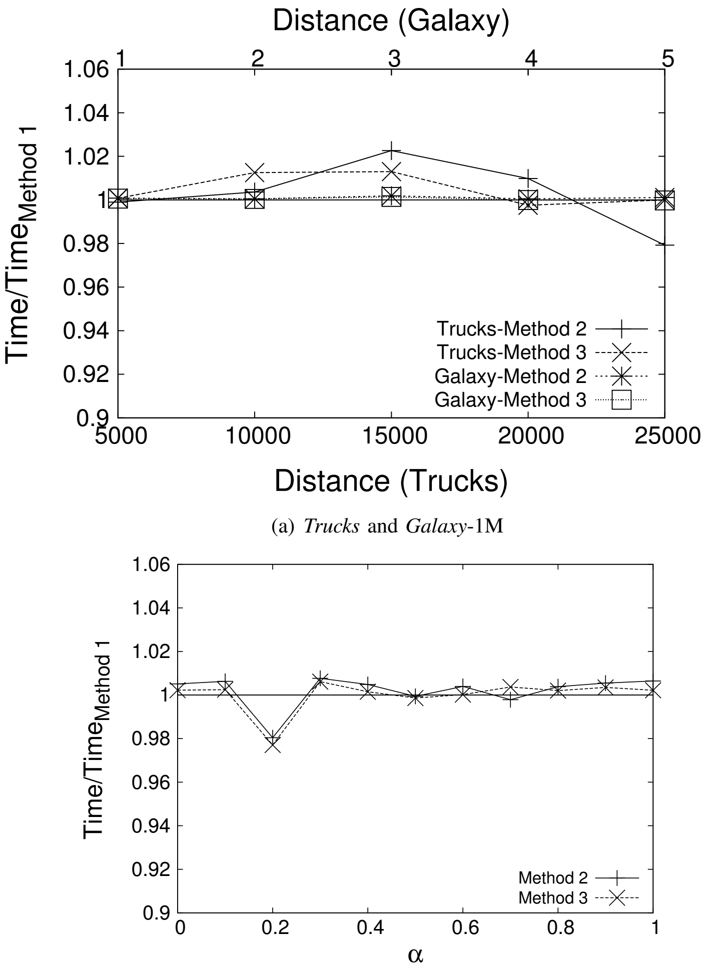
We have implemented the filtering methods in the previous section in TRAJDISTSEARCH and in this section we measure response times ignoring the R-tree search, i.e., focusing only on the filtering and the moving distance computation. We use the following distance threshold searches:

- S5: From the *Trucks* dataset, 10 trajectories are processed for 100% of their temporal extent.
- S6: From the *Galaxy*-1M dataset, 100 trajectories are processed for 100% of their temporal extent.
- S7: From the *Random*-1M datasets, 100 trajectories are processed for 100% of their temporal extent, with a fixed query distance $d = 15$.

Figure 12 (a) plots the relative improvement (i.e., ratio of response times) of using Method 2 and Method 3 over using Method 1 vs. the threshold distance for S5 and S6 above for the Galaxy and Trucks datasets. Data points below the $y = 1$ line indicate that filtering is beneficial. We see that filtering is almost never beneficial and can in fact marginally increase response time. Similar results are shown for the *Random*-1M datasets in Figure 12 (b).

It turns out that our methods filter only a small fraction of the line segments. For instance, for search S7 Method 2, resp. Method 3, filters out between 2.5% and 12%, resp. between 3.2% and 15.9%, of the line segments. Therefore, for most candidate segments the time spent doing filtering is pure overhead. Furthermore, filtering requires only a few floating point operations but also several if-then-else statements. The resulting branch instructions slow down executions (due to pipeline stalls) when compared to straight line code. We conclude that, at least for the datasets and searches we have used, our filtering methods are not effective.

One may envision developing better filtering methods to achieve (part of the) filtering potential seen in Figure 11. We profiled the execution of TRAJDISTSEARCH for searches S5,



(a) *Trucks* and *Galaxy*-1M

(b) *Random*-1M

Figure 12. Performance improvement ratio of filtering methods (a) for real datasets with S5 and S6, vs. query distance, (b) for *Random*-1M datasets with S7.

S6, and S7, with no filtering, and accounting both for the R-tree search and the distance computation. We found that the time spent searching the R-tree accounts for at least 97% of the overall response time. As a result, filtering can only lead to marginal performance improvements for the datasets and searches in our experiments. For other datasets and searches, however, the fraction of time spent computing distances could be larger. Nevertheless, given the results in this section, in all that follows we do not perform any filtering.

## VIII. Index Resolution

In this section, we propose methods to represent the trajectory segments in a different configuration within the index. According to the cost model in [35], index performance depends on the number of nodes in the index, but also on the volume and surface area of the MBBs. Query performance can be improved by finding a suitable number of nodes in the index combined with a good partitioning strategy of trajectory segments within MBBs. One extreme is to store an entire trajectory in a single MBB as defined by the spatial and temporal properties of the trajectory; however, this leads to a lot of “wasted MBB space.” The other extreme is to store each trajectory line segment in its own MBB, as done so far in this paper and in previous work on $k$NN searches [22], [23], [24], [10]. In this scenario, the volume occupied by the trajectory in the index is minimized, with the trade-off that the number of entries in the index will be maximized.

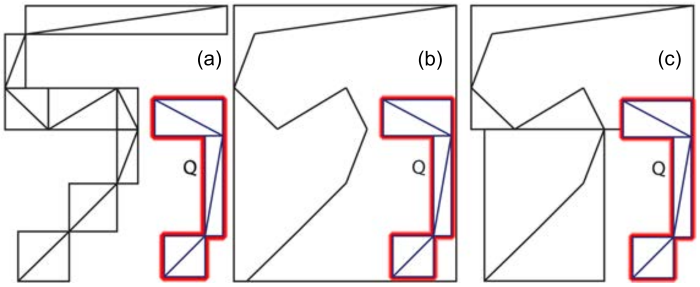


Figure 13. Illustration of the relationship between wasted space, volume occupied by indexed trajectories, and the number of returned candidate segments to process. An 8-segment trajectory is indexed in three different ways, and searched against a 3-segment query trajectory (denoted $Q$ in the figure), where the query distance is shown in red. (a) Each trajectory segment is stored in its own MBB. (b) The trajectory is stored in a single MBB. (c) The trajectory is stored in two MBBs.

In Figure 13 (a) we depict an entry trajectory that is stored with each segment in its own MBB, in Figure 13 (b), a trajectory that is stored in a single MBB, and in Figure 13 (c) a trajectory that is stored in two MBBs. A 3-segment query trajectory that is not within the query distance of the entry trajectory is shown, where the query distance is indicated by the red outline. Assigning a single line segment to a single MBB (Figure 13 (a)) minimizes wasted space but maximizes the number of nodes in the index that need to be searched. Storing an entire trajectory in its own MBB minimizes the number of index entries to be searched but leads to more index overlap and more candidate segments. For example, consider the query in Figure 13 (b). From the figure, it can be seen that each of the three query segments overlap the MBB, resulting in 3×8=24 candidate trajectory segments that need to be processed. However, in Figure 13 (a), the query trajectory does not overlap any of the entry MBBs, and therefore no candidate trajectory segments are returned; however, the index contains 8 elements instead of 1, as in Figure 13 (b). Figure 13 (c) shows the case in which the entry trajectory is stored in only 2 MBBs. In this case only 1 query segment overlaps an entry MBB, resulting in 1×5=5 candidate segments to process.

As shown above, assigning a fraction of a trajectory to a single MBB, as a series of line segments, increases the volume a trajectory occupies in the index, and the degree of index overlap. This is because the resulting MBB is larger in comparison to minimizing the volume of the MBBs by encompassing each individual trajectory line segment by its own MBB. As a result, an index search can return a portion of a trajectory that does not overlap the query, leading to increased overhead when processing the candidate set of line segments returned by the index. However, the number of entries in the index is reduced, thereby reducing tree traversal time. To explore the trade-off between the number of nodes in the index, the amount of wasted volume required by a trajectory, the index overlap, and the overhead of processing candidate trajectory segments, in this section, we evaluate three strategies for splitting individual trajectories into a series of consecutive MBBs. Such splitting can be easily implemented as an array of references to trajectory segments (leading to one extra indirection when compared to assigning a single segment per MBB). We evaluate performance experimentally by splitting the trajectories, and then creating their associated indexes, where the configuration with the lowest query response time is highlighted.

### *A. Static Temporal Splitting*

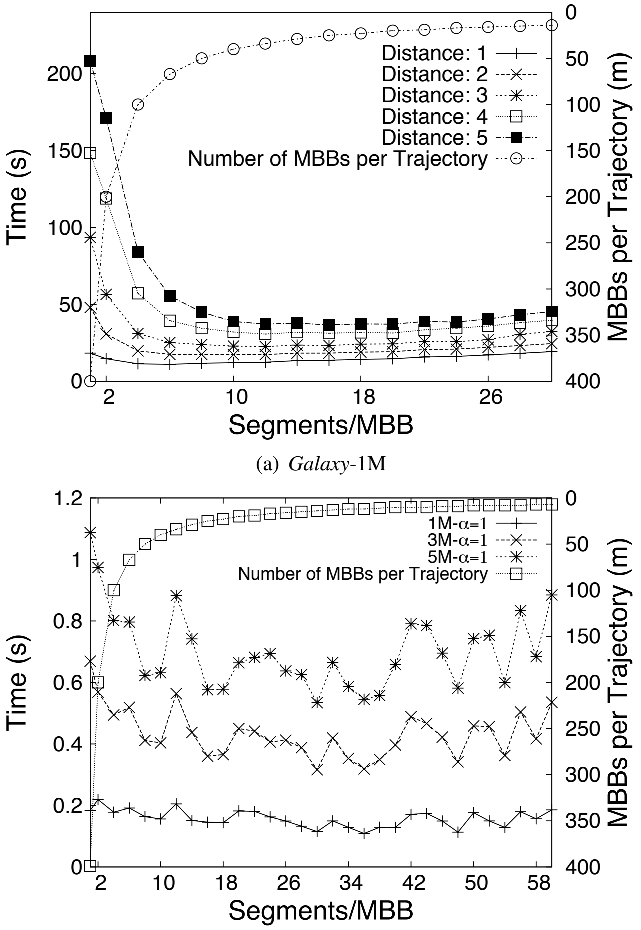
Assuming it is desirable to ensure that trajectory segments are stored contiguously, we propose a simple trajectory splitting method. Given a trajectory of $n$ line segments, we split the trajectory by assigning $r$ contiguous line segments per MBB, where $r$ is a constant. Therefore, the number of MBBs, $m$, to represent the trajectory is $m = \lceil \frac{n}{r} \rceil$. By storing segments contiguously, this strategy leads to high temporal locality of reference, which may be important for cache reuse in our in-memory database, in addition to the benefits of the high spatial discrimination of the R-tree (see Section IV).

Figure 14 plots response time vs. $r$ for the S6 (*Galaxy* dataset) and S7 (*Random* dataset) searches defined in Section VII-B. For S6, 5 different query distances are used, while for S7 the query distance is fixed as 15 but results are shown for various dataset sizes for $\alpha = 1$. The right y-axis shows the number of MBBs used per trajectory. The data points at $r = 1$ correspond to the original implementation (rather than the implementation with $r = 1$, which would include one unnecessary indirection).

The best value for $r$ depends on the dataset and the search. For instance, in the *Galaxy*-1M dataset (S6) using 12 segments per MBB (or $m = 34$) leads to the best performance. We note that picking a $r$ value in a large neighborhood around this best value would lead to only marginally higher query response times. In general, using a small value of $r$ can lead to high response times, especially for $r = 1$ (or $m = 400$). For instance, for S6 with a query distance of 5, the response time with $r = 1$ is above 208 s while it is just above 37 s with $r = 12$. With $r = 1$ the index is large and thus time-consuming to search. A very large $r$ value does not lead to the lowest response time since in this case many of the segments returned from the R-tree search are not query matches. Finally, results in Figure 14 (a) show that the advantage of assigning multiple trajectory segments per MBB increases as the query distance increases. For instance, for a distance of 2 using $r = 12$ decreases the response time by a factor 2.76 when compared to using $r = 1$, while this factor is 5.6 for a distance of 5. Note that the difference in response times between Figure 14 (a) and (b) are largely due to more query hits in *Galaxy* in comparison to *Random* for the query distances selected.

### *B. Static Spatial Splitting*

Another strategy consists in ordering the line segments belonging to a trajectory spatially, i.e., by sorting the line segments of a trajectory by the $x$, $y$, and $z$ values of the segment's origin lexicographically. We then assign $r$ segments per trajectory into each MBB, as in the previous method. With such spatial grouping, the line segments are no longer guaranteed to be temporally contiguous in their MBBs, but reduced index overlap may be achieved. Figure 15 plots response time vs. $r$ for the S7 (*Random* dataset) searches. We see that there is no advantage to assigning multiple trajectory segments to an MBB over assigning a single line segment to a MBB ($r = 1$ in the plot). When comparing with results in Figure 14 (b) we find that spatial splitting leads to query response times higher by several factors than that of temporal splitting.



(a) *Galaxy*-1M

(b) *Random* datasets

Figure 14. Static Temporal Splitting: Response time vs. $r$ for (a) S6 for the *Galaxy*-1M dataset for various query distances; and (b) S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15. The number of MBBs per trajectory, $m$, is shown on the right vertical axis.

### *C. Splitting to Reduce Trajectory Volume*

The encouraging results in Section VIII-A suggest that using an appropriate trajectory splitting strategy can lead to performance gains primarily by exploiting the trade-off between the number of entries in the index and the amount of wasted space that leads to higher index overlap. More sophisticated methods can be used. In particular, we implement the heuristic algorithm *MergeSplit* in [36], which is shown to produce a splitting close to optimal in terms of wasted space. *MergeSplit* takes as input a trajectory, $T$, as a series of $l$ line segments, and a constant number of MBBs, $m$. As output, the algorithm creates a set of $m$ MBBs that encapsulate the $l$ segments of $T$. The pseudocode of *MergeSplit* is as follows:

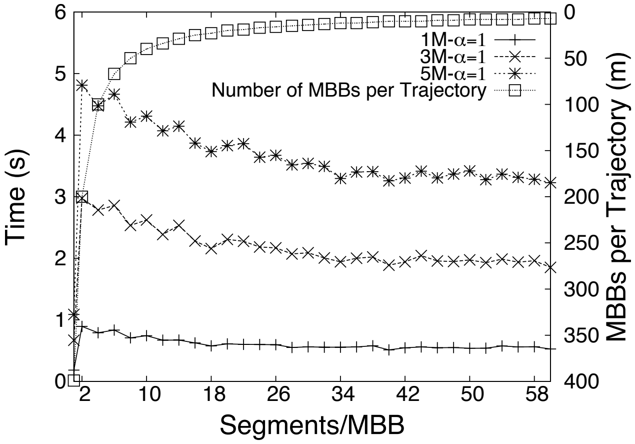


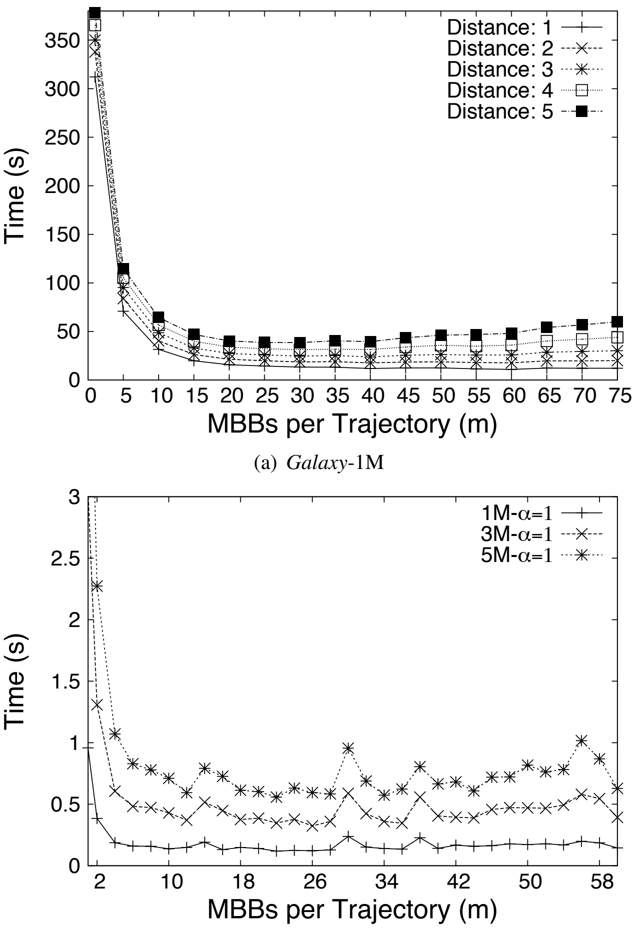
Figure 15. Static Spatial Splitting: Response time vs. $r$ using S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15. The number of MBBs per trajectory, $m$, for each data point is shown on the rightmost vertical axis.

1) For $0 \leq i < l$ calculate the volume of the merger of the MBBs that define $l_i$ and $l_{i+1}$ and store the resulting series of MBBs and their volumes.
2) To obtain $m$ MBBs, merge consecutive MBBs that produce the smallest volume increase at each iteration and repeat $(l-1)-(m-1)$ times. After the first iteration, there will be $l-2$ initial MBBs describing line segments, and one MBB that is the merger of two line segment MBBs.

Figure 16 shows response time vs. $m$ for S6 (*Galaxy* dataset) and S7 (*Random* datasets). Compared to static temporal splitting, which has a constant number of segments, $r$ per MBB, *MergeSplit* has a variable number of segments per MBB. From the figure, we observe that for the *Galaxy*-1M dataset (S6), $m = 30$ leads to the best performance. Comparing *MergeSplit* to the static temporal splitting (Figures 14 and 16 (a)), the best performance for the S6 (*Galaxy* dataset) is achieved by the static temporal splitting. For S7, the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets, *MergeSplit* is only marginally better than the static temporal splitting (Figures 14 and 16 (b)). This is surprising, given that the total hypervolume of the entries in the index for a given $m$ across both splitting strategies is higher for the simple static temporal splitting, as it makes no attempt to minimize volume. Therefore, the trade-off between the number of entries and overlap in the index cannot fully explain the performance of these trajectory splitting strategies for distance threshold searches.

### D. Discussion

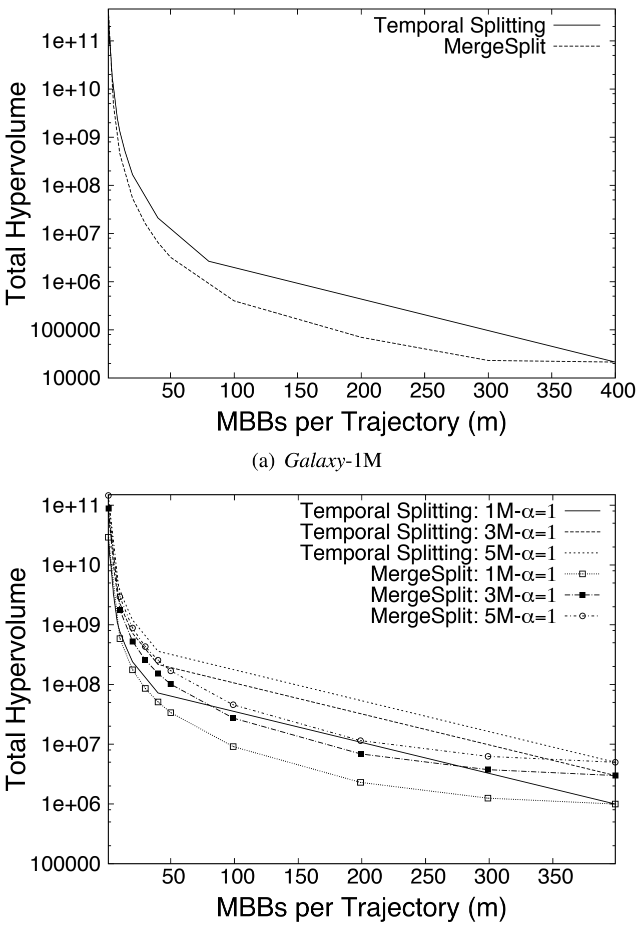
A good trade-off between the number of entries in the index and the amount of index overlap can be achieved by selecting an appropriate trajectory splitting strategy. However, comparing the results of the simple temporal splitting strategy (Section VIII-A) and *MergeSplit* (Section VIII-C), we find that volume minimization did not significantly improve performance for S7, and led to worse performance for S6. In Figure 17, we plot the total hypervolume vs. $m$ for the *Galaxy*-1M (S6) and the *Random*-1M, 3M, and 5M $\alpha = 1$ (S7) datasets. $m = 1$ refers to placing an entire trajectory in a single MBB, and the maximum value of $m$ refers to placing each individual line segment of a trajectory in its own MBB. For the static temporal splitting strategy, $m = 34$ leads to the best performance for the *Galaxy*-1M dataset (S6), whereas this value is $m = 30$ for *MergeSplit*. The total hypervolume of the MBBs in units of kpc$^3$Gyr for the static temporal grouping strategy at $m = 34$ is $3.6 \times 10^7$, whereas for *MergeSplit* at $m = 30$, it is $1.62 \times 10^7$, i.e., 55% less volume. Due to the greater volume occupied by the MBBs, index overlap is much higher for the static temporal splitting strategy. Figure 18 (a) plots the number of overlapping line segments vs. $m$ for S6 with $d = 5$. From the figure, we observe that independently of $m$, *MergeSplit* returns a greater number of candidate line segments to process than the simple temporal splitting strategy. *MergeSplit* attempts to minimize volume; however, if an MBB contains a significant fraction of the line segments of a given trajectory, then all of these segments are returned as candidates. The simple temporal grouping strategy has an upper bound ($r$) on the number of segments returned per overlapping MBB and thus can return fewer candidate segments for a query, despite occupying more volume in the index. For in-memory distance threshold searches, there is a trade-off between a trajectory splitting strategy that has an upper bound on the number of line segments per MBB, and index overlap, characterized



(a) *Galaxy*-1M

(b) *Random* datasets

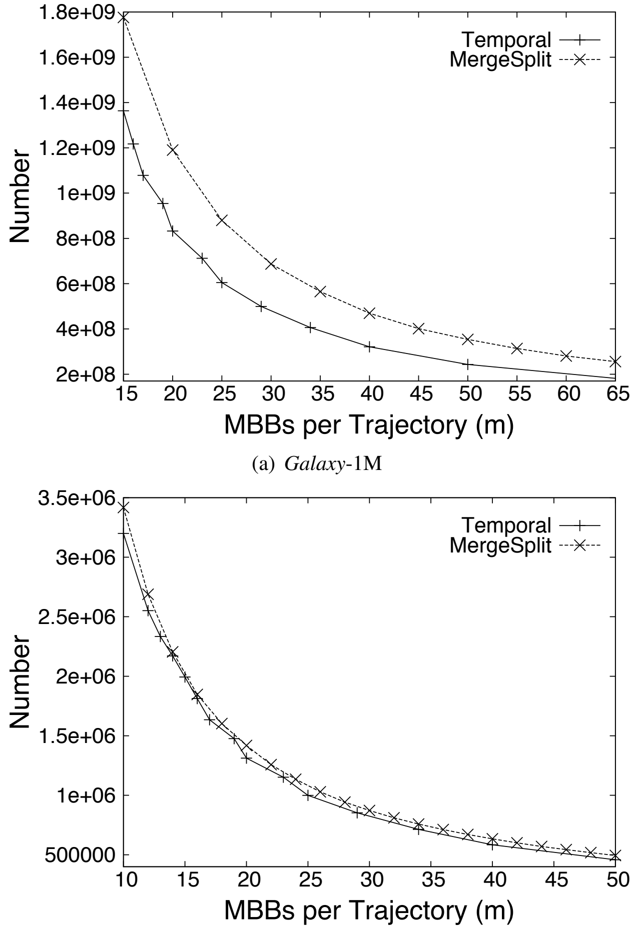
Figure 16. Greedy Trajectory Splitting: Response time vs. $m$ for (a) S6 for the *Galaxy*-1M dataset for various query distances; and (b) S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15.



(a) *Galaxy*-1M

(b) *Random*-1M

Figure 17. Total hypervolume vs. $m$ for the static temporal splitting strategy and *MergeSplit*. (a)for the *Galaxy*-1M dataset (S6); and (b) for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets (S7).



(a) *Galaxy*-1M

(b) *Random*-1M

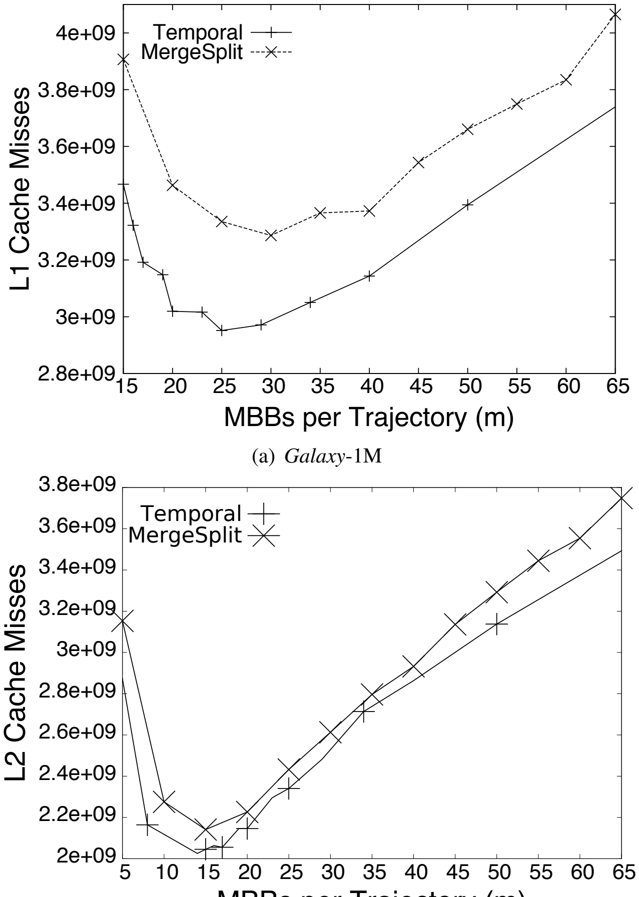
Figure 18. Total number of overlapping segments vs. $m$ for the static temporal splitting strategy and *MergeSplit*. (a) S6 for the *Galaxy*-1M dataset with $d = 5$; and (b) S7 for the *Random* $\alpha = 1$ dataset with $d = 15$.

by the volume occupied by the MBBs in the index. This is in sharp contrast to other works that focus on efficient indexing of spatiotemporal objects in traditional out-of-core implementations where the index resides partially in-memory and on disk, and therefore volume reduction to minimize index overlap is necessary to minimize disk accesses (e.g., [36]).

A single metric cannot capture the trade-offs between the number of entries in the index, volume reduction, index overlap, and the number of candidate line segments returned (germane to distance threshold searches). However, for *Galaxy*-1M (S6), a value of $m = 34$ and $m = 30$ lead to the best query response time for the temporal splitting strategy and *MergeSplit*, respectively (Figures 14 (a) and 16 (a)). Figure 19 (a) shows the number of L1 cache misses vs. $m$ for S6 with $d = 5$. The number of cache misses was measured using PAPI [37]. The best values of $m$ in terms of query response time for both of the trajectory splitting strategies ($m = 34$ and $m = 30$) roughly correspond to a value of $m$ that minimizes L1 cache misses. Thus, L1 cache misses appear to be a good indicator of relative query performance under different indexing methods. Figure 19 (b) shows the number L2 cache misses vs. $m$ for S6 with $d = 5$. We note that when comparing Figure 19 (a) and (b), there are more L1 cache misses for a given value of $m$ because the L1 cache is smaller than L2 cache. We see that unlike L1 cache misses, $m$ values that minimize L2 cache misses do not lead to the best response times for either splitting strategy. Therefore, L1 cache misses are a better predictor of query performance when comparing indexing methods. Future work for in-memory distance threshold searches should focus on improved cache reuse through temporal locality of reference (which is in part obtained by storing segments contiguously within a single MBB).

### *E. Performance Considerations for In-memory vs. Out-of-Core Implementations*

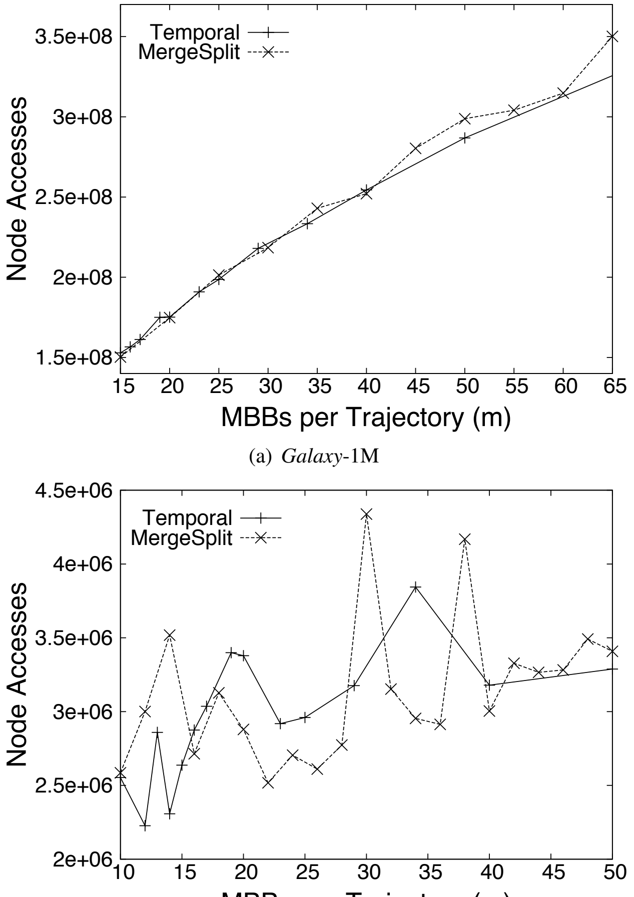
The focus of this work is on in-memory distance threshold searches; however, most of the literature on MODs assume out-of-core implementations, where the number of node accesses are used as a metric to estimate I/O activity. Figure 20 shows the number of node accesses vs. $m$ for both of the static temporal splitting strategy and *MergeSplit*. We find that for the *Galaxy*-1M dataset (S6) with $d = 5$, there is a comparable number of node accesses for both trajectory splitting methods. However, for S7 (*Random*-1M), on average, trajectory splitting with *MergeSplit* requires fewer node accesses and may



(a) *Galaxy*-1M

(b) *Random*-1M

Figure 19. L1 (a) and L2 (b) cache misses vs. $m$ for the static temporal splitting strategy and *MergeSplit* for the *Galaxy*-1M dataset (S6) with $d = 5$.



(a) *Galaxy*-1M

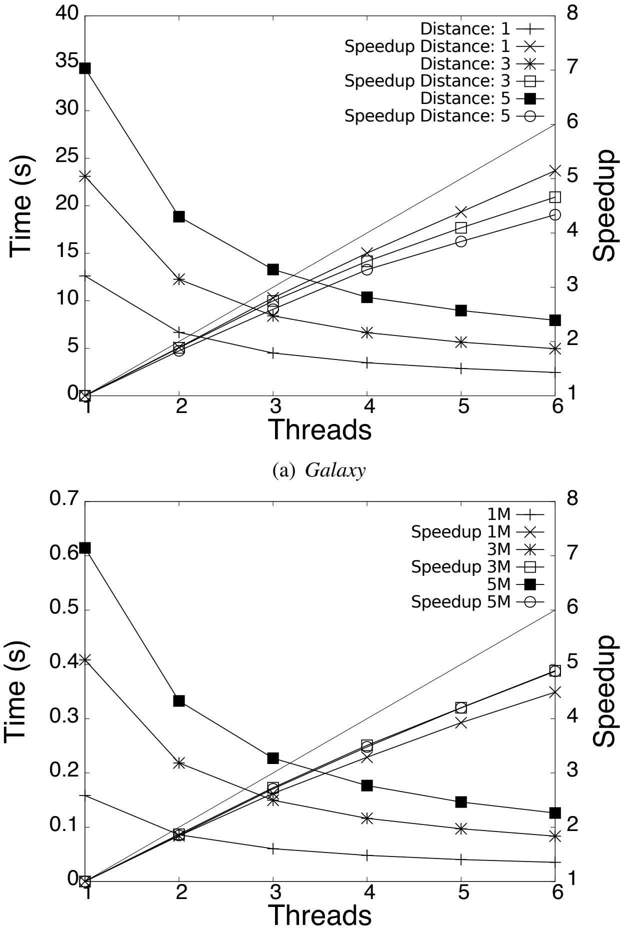
(b) *Random*-1M

Figure 20. Node Accesses vs. $m$ for the static temporal splitting strategy and *MergeSplit*. (a) S6 for the *Galaxy*-1M dataset with $d = 5$; and (b) S7 for the *Random* $\alpha = 1$ dataset with $d = 15$.

perform significantly better than the simple temporal splitting strategy in an out-of-core implementation. For example, in Figure 20 (b) some values of $m$ have a significantly higher number of node accesses, such as values around 14, 30, 38, due to the idiosyncrasies of the data, and resulting index overlap. However, as we demonstrated in Section VIII-D, distance threshold searches in the context of in-memory databases also benefit from reducing the number of candidate line segments returned, and this is not entirely volume contingent. Therefore, methods that consider volume reduction, such as the *MergeSplit* algorithm of [36], or other works that consider volume reduction in the context of query sizes, such as [38], may not be entirely applicable to distance threshold searches.

### F. Multi-core Execution with OpenMP

In Section VIII-D, we noted that indexing multiple line segments in a single MBB leads to performance improvements and that the temporal splitting strategy performed better than the spatial splitting strategy and *MergeSplit*. Regardless of the trajectory splitting strategy utilized, TRAJDISTSEARCH can be parallelized, e.g., using OpenMP, in a shared-memory environment. The iterations of the loop on line 3 of TRAJDISTSEARCH in Figure 4 are independent, each iteration can thus be assigned to a different thread. Figure 21 shows the response time on the 6-core platform described in Section VI-B vs. the number of threads for S6 and S7 with $r = 12$ and $r = 10$, respectively. These values of $r$ yield the best performance gain in the sequential implementation for S6 and S7 (Figure 14). Parallelizing the outer loop leads to high parallel efficiency between 72.2%-85.7%, with parallel speedup between 4.33 and 5.14 with 6 threads for query distances ranging from $d = 1$ to $d = 5$ for the *Galaxy* dataset with S6. For the *Random*-1M, 3M and 5M $\alpha = 1$ datasets, with 6 threads, we observe a speedup between 4.49 to 4.88, for a parallel efficiency between 74.8% and 81.3%. We note from Figure 21 (a) that the speedup decreases as $d$ increases. This suggests that as the number of candidate segments increases (with increasing $d$), there is likely to be increased memory contention, as more candidate segments between the threads are competing for space in the CPU cache. Additionally, with an increased $d$, there will be more nodes to visit in the R-tree; however, the threads can traverse the tree in parallel. It is not clear which mechanism predominantly causes the slowdown with increasing query distance. However, from previous sections we saw that the number of candidate trajectory segments can be large, and it is likely that processing candidate trajectory segments is the main bottleneck in parallelizing distance threshold searches.



(a) *Galaxy*

(b) *Random* datasets

Figure 21. Response time vs. number of threads (a) S6 for the *Galaxy* dataset for various query distances and $r = 12$; and (b) S7 for the *Random*-1M, 3M, and 5M datasets, with a query distance of 15 and $r = 10$.

Distance threshold searches, and perhaps other spatiotemporal searches on trajectories can be parallelized in a straightforward manner in a shared-memory environment because the searches can be performed independently of each other. The focus in the spatiotemporal database community has been on out-of-core, sequential implementations; however, with new architectures, and large main memories, there are a number of attractive alternatives to the current solutions.

## IX. Conclusion

In-memory distance threshold searches for trajectory and point searches on moving object trajectories are significantly different from the well-studied $k$NN searches [22], [23], [24], [10]. We made a case for using an R-tree index to store trajectory segments, and found it to perform robustly for two real world datasets and a synthetic dataset. We focused on 4-D datasets (3 spatial + 1 temporal) while other works only consider 3-D datasets [22], [23], [24], [10].

We demonstrated that for distance threshold searches, many segments returned by the index search must be excluded from the result set, because there is no limit to the number of candidate trajectory segments that can be returned. We have proposed computationally inexpensive solutions to filter out candidate segments, but found that they have poor selectivity. A more promising direction for reducing query response time is to reduce the time spent traversing the tree index.

We demonstrated that efficiently splitting trajectories is beneficial because the penalty for the increased index overlap is offset by the reduction in number of index entries. We find that for in-memory distance threshold searches, the number of line segments returned per overlapping MBB has an impact on performance, where attempts to reduce the volume of the MBBs that store a trajectory may be at cross-purposes with returning a limited number of candidate segments per overlapping MBB. Therefore, at least for in-memory implementations, trajectory splitting methods that focus on volume reduction are not necessarily preferable to a simple and bounded grouping of line segments in MBBs for distance threshold searches.

We show that the distance threshold search can be performed in parallel using threads in a shared-memory environment using OpenMP. The results show that the tree traversals and processing of candidate segments can be performed in parallel with high parallel efficiency. The results are encouraging for future prospects in parallel query optimization, and suggests that a promising future work direction is to investigate both shared- and distributed-memory implementations.

A future direction is to explore trajectory splitting methods that achieve volume reduction while bounding the number of MBBs used per trajectory. Another direction is to investigate non-MBB-based data structures to index line segments, such as that in [39]. Analytical models of query performance in these settings may be heavily dependent on modeling cache reuse.

One may wonder whether the idea of assigning multiple segments to an MBB is generally applicable, and in particular for $k$NN searches on trajectories [22], [23], [24], [10]. The $k$NN literature focuses on pruning strategies and associated metrics that require a high resolution index, thus implying storing a single trajectory segment in an MBB. Furthermore, $k$NN algorithms maintain a list of nearest neighbors over a time interval, which would lead to greater overhead if multiple segments were stored per MBB. Therefore, the approach of grouping line segments together in a single MBB may be ineffective for $k$NN searches. An interesting problem is to reconcile the differences between both types of searches in terms of index resolution.

## Acknowledgments

This material is based upon work supported by the National Aeronautics and Space Administration through the NASA Astrobiology Institute under Cooperative Agreement No. NNA08DA77A issued through the Office of Space Science, and by NSF Award CNS-0855245.

## References

[1] M. Gowanlock and H. Casanova, "In-memory distance threshold queries on moving object trajectories," in Proc. of the Sixth Intl. Conf. on Advances in Databases, Knowledge, and Data Applications, 2014, pp. 41–50.

[2] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "A data model and data structures for moving objects databases," in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 2000, pp. 319–330.

[3] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, “A foundation for representing and querying moving objects,” ACM Trans. Database Syst., vol. 25, no. 1, 2000, pp. 1–42.

[4] S. Arumugam and C. Jermaine, “Closest-point-of-approach join for moving object histories,” in Proc. of the 22nd Intl. Conf. on Data Engineering, 2006, pp. 86–95.

[5] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, “Discovery of convoys in trajectory databases,” Proc. VLDB Endow., vol. 1, no. 1, Aug. 2008, pp. 1068–1080.

[6] A. Guttman, “R-trees: a dynamic index structure for spatial searching,” in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1984, pp. 47–57.

[7] D. Pfoser, C. S. Jensen, and Y. Theodoridis, “Novel approaches in query proc. for moving object trajectories,” in Proc. of the 26th Intl. Conf. on Very Large Data Bases, 2000, pp. 395–406.

[8] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, “Spatio-temporal indexing for large multimedia applications,” in Proc. of the Intl. Conf. on Multimedia Computing and Systems, 1996, pp. 441–448.

[9] V. P. Chakka, A. Everspaugh, and J. M. Patel, “Indexing large trajectory data sets with SETI,” in Proc. of Conference on Innovative Data Systems Research, 2003, pp. 164–175.

[10] R. H. Güting, T. Behr, and J. Xu, “Efficient k-nearest neighbor search on moving object trajectories,” The VLDB Journal, vol. 19, no. 5, 2010, pp. 687–714.

[11] P. Cudre-Mauroux, E. Wu, and S. Madden, “TrajStore: an adaptive storage system for very large trajectory data sets,” in Proc. of the 26th Intl. Conf. on Data Engineering, 2010, pp. 109–120.

[12] M. R. Vieira, P. Bakalov, and V. J. Tsotras, “On-line discovery of flock patterns in spatio-temporal data,” in Proc. of the 17th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems, 2009, pp. 286–295.

[13] Z. Li, M. Ji, J.-G. Lee, L.-A. Tang, Y. Yu, J. Han, and R. Kays, “Movemine: Mining moving object databases,” in Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data, 2010, pp. 1203–1206.

[14] N. Roussopoulos, S. Kelley, and F. Vincent, “Nearest neighbor queries,” in Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1995, pp. 71–79.

[15] Z. Song and N. Roussopoulos, “K-nearest neighbor search for moving query point,” in Proc. of the 7th Intl. Symp. on Advances in Spatial and Temporal Databases, 2001, pp. 79–96.

[16] Y. Tao, D. Papadias, and Q. Shen, “Continuous nearest neighbor search,” in Proc. of the 28th Intl. Conf. on Very Large Data Bases, 2002, pp. 287–298.

[17] R. Benetis, S. Jensen, G. Karciauskas, and S. Saltenis, “Nearest and reverse nearest neighbor queries for moving objects,” The VLDB Journal, vol. 15, no. 3, 2006, pp. 229–249.

[18] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, “Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring,” in Proc. of ACM SIGMOD Intl. Conf. on Danagement of data, 2005, pp. 634–645.

[19] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao, “A threshold-based algorithm for continuous monitoring of k nearest neighbors,” IEEE Trans. on Knowl. and Data Eng., vol. 17, no. 11, 2005, pp. 1451–1464.

[20] X. Xiong, M. F. Mokbel, and W. G. Aref, “SEA-CNN: Scalable proc. of continuous k-nearest neighbor queries in spatio-temporal databases,” in Proc. of the 21st Intl. Conf. on Data Engineering, 2005, pp. 643–654.

[21] X. Yu, K. Q. Pu, and N. Koudas, “Monitoring k-nearest neighbor queries over moving objects,” in Proc. of the 21st Intl. Conf. on Data Engineering, 2005, pp. 631–642.

[22] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, “Nearest neighbor search on moving object trajectories,” in Proc. of the 9th Intl. Conf. on Advances in Spatial and Temporal Databases, 2005, pp. 328–345.

[23] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, “Algorithms for nearest neighbor search on moving object trajectories,” Geoinformatica, vol. 11, no. 2, 2007, pp. 159–193.

[24] Y.-J. Gao, C. Li, G.-C. Chen, L. Chen, X.-T. Jiang, and C. Chen, “Efficient k-nearest-neighbor search algorithms for historical moving object trajectories,” J. Comput. Sci. Technol., vol. 22, no. 2, 2007, pp. 232–244.

[25] S. Qiao, C. Tang, S. Dai, M. Zhu, J. Peng, H. Li, and Y. Ku, “Partspan: Parallel sequence mining of trajectory patterns,” in Fifth Intl. Conf. on Fuzzy Systems and Knowledge Discovery, vol. 5, Oct 2008, pp. 363–367.

[26] R. Jinno, K. Seki, and K. Uehara, “Parallel distributed trajectory pattern mining using mapreduce,” in 2012 IEEE 4th Intl. Conf. on Cloud Computing Technology and Science (CloudCom), Dec 2012, pp. 269–273.

[27] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, “Trajectory pattern mining,” in Proc. of the 13th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, 2007, pp. 330–339.

[28] D. Zeinalipour-Yazti, S. Lin, and D. Gunopulos, “Distributed spatio-temporal similarity search,” in Proc. of the 15th ACM Intl. Conf. on Information and Knowledge Management. ACM, 2006, pp. 14–23.

[29] J. Zhang, S. You, and L. Gruenwald, “Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs.” Information Systems, vol. 44, 2014, pp. 134 – 154.

[30] M. G. Gowanlock, D. R. Patton, and S. M. McConnell, “A model of habitability within the Milky Way galaxy,” Astrobiology, vol. 11, 2011, pp. 855–873.

[31] http://www.chorochronos.org/, accessed 5-February-2014.

[32] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento, “On the generation of spatiotemporal datasets,” in Proc. of the 6th Intl. Symp. on Advances in Spatial Databases, 1999, pp. 147–164.

[33] http://navet.ics.hawaii.edu/%7Emike/datasets/DBKDA2014/datasets.zip, accessed 12-February-2014.

[34] http://www.superliminal.com/sources/sources.htm, accessed 5-February-2014.

[35] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer, “Towards an analysis of range query performance in spatial data structures,” in Proc. of the 12th Symp. on Principles of Database Sys., 1993, pp. 214–221.

[36] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, “Efficient indexing of spatiotemporal objects,” in Proc. of the 8th Intl. Conf. on Extending Database Technology: Advances in Database Technology, 2002, pp. 251–268.

[37] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: A portable interface to hardware performance counters,” in Proc. of the Department of Defense HPCMP Users Group Conf., 1999, pp. 7–10.

[38] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento, “A trajectory splitting model for efficient spatio-temporal indexing,” in Proc. of the 31st Intl. Conf. on Very Large Data Bases, 2005, pp. 934–945.

[39] E. Bertino, B. Catania, and B. Shidlovsky, “Towards optimal indexing for segment databases,” in Proc. of the 6th Intl. Conf. on Advances in Database Technology, 1998, pp. 39–53.