IN-MEMORY DISTANCE THRESHOLD SEARCHES ON MOVING OBJECT TRAJECTORIES

A DISSERTATION SUBMITTED TO THE OFFICE OF GRADUATE EDUCATION OF THE UNIVERSITY OF HAWAI'I AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

MAY 2015

By
Michael Gowanlock

Dissertation Committee:

Henri Casanova, Chairperson
Kim Binsted
Rich Gazan
Lipyeow Lim
Joshua Barnes

We certify that we have read this dissertation and that, in our opinion, it is satisfactory in scope and quality as a dissertation for the degree of Doctor of Philosophy in Computer Science.

DISSERTATION COMMITTEE

_____
Chairperson

_____

_____

_____

_____

Acknowledgments

This dissertation is the result of the encouragement of several individuals, and I cannot adequately express my gratitude to them. I would like to begin by thanking Kim Binsted. I met Kim at the Astrobiology Science Conference in 2010 in League City, Texas. I arranged to meet with Kim, as I knew that she was also a computer scientist and had research interests in astrobiology. She encouraged me to apply to the Computational Astrobiology Summer School in the summer of 2010, and the graduate program at the University of Hawai‘i (UH). Additionally, she suggested that I meet with Henri Casanova to discuss research directions in high performance computing. I was accepted to the summer school, and had the pleasure of meeting Henri. I applied to the graduate program at UH, was accepted, and began to work with Henri as my advisor in January 2011. If I had not met Kim at AbSciCon, I certainly would not have ended up in Hawaii, and my life would have taken a significantly different path. I would like to thank Kim for her encouragement, and the incredible opportunity that she made possible.

I would like to thank Rich Gazan for his support in multiple capacities. Rich advocated that I receive a graduate assistantship from the University of Hawaii NASA Astrobiology Institute (UHNAI). Without this financial support, I would have not been able to attend UH. As Rich's graduate assistant, he allowed me the freedom to pursue any research avenue that I thought was interesting under the AIRFrame project. I very much enjoyed working on our projects relating to interdisciplinary research. Beyond academic support, I would like to express my gratitude to Rich for all of his kindness and support over the years. Thank you for all of the rides to the airport, the rounds of beer, helping me move, and for also letting me borrow your spare mattress when I could not afford one (and please express my gratitude to Leah as well!). Rich embodies human virtues to a degree unparalleled by most, and I have benefited from his disposition countless times.

When looking for a university to attend for my doctoral studies, one challenge was finding an advisor that would be open to the possibility of allowing me to pursue a project related to

my masters work on the habitability of the Milky Way galaxy. Working with Henri has been a testament to his open-minded approach to research. I would like to thank Henri for accepting me as his student, despite the uncertain research path ahead. During the first year and a half of my studies, I spent most of my time reading astronomy papers in search of a potential connection to computer science. After having several meetings where I would discuss topics such as the distribution of dark matter in our Galaxy, Henri made the suggestion that I make an astronomy cut-off date of the middle of October 2012; otherwise, I would not make progress towards the completion of my degree. This proved to be excellent advice, as I immersed myself in the computer science literature, and defended my dissertation proposal only five months later in March 2013. Without Henri's guidance, I would probably still be showing up to our weekly meetings to discuss astronomy. Henri encouraged me to define my own research avenue, and provided me with much needed support. As a result of Henri's mentoring philosophy, I believe that, if possible, students should choose their own projects. Henri has been a wonderful advisor and I really do not have the words to express how much I have enjoyed working with him over the years. Henri has been very influential in shaping me as an academic and I cannot thank him enough for his support. I consider myself lucky to have been Henri's student, and I will truly miss our meetings.

I would like to thank David Schanzenbach, who has assisted me over the years with research projects and provided me with technical assistance on numerous occasions. David wrote scripts for research projects that were beyond my capabilities, thus leading to better research outputs, and reduced the amount of monotonous labor I had to endure during data collection. David was the progenitor of the GPU research thrust presented in this dissertation. Without his suggestion and assistance, I may not have considered the GPU in this work. David has been accommodating beyond articulation. He even provided technical assistance for the dissertation defense. I would like to thank David for all of his generosity over the years. Everyone in the lab has benefited from David's helpful character. When he eventually leaves the lab, I am certain that everything will break shortly thereafter.

I would like to thank Lipyeow Lim for providing feedback on manuscripts before we sent them for review. I am indebted to Josh Barnes for his insightful comments on the astronomy component of this work, and for the galaxy merger dataset. Additionally, I would like to acknowledge my friends in Hawaii who made the last four years an enjoyable experience. Finally, I would

like to acknowledge the financial support of the NASA Astrobiology Institute, the Canadian Space Agency, IEEE, and the University of Hawai‘i Department of Information and Computer Sciences.

Abstract

Processing moving object trajectories arises in many applications. To gain insight into target domains, historical continuous trajectory similarity searches find those trajectories that have similar attributes in common. In this work, we focus on a trajectory similarity search, the distance threshold search, which finds all trajectories within a given distance of a query trajectory over a time interval. We investigate novel approaches for the efficient processing of these searches over in-memory databases on the CPU and on the GPU.

On the CPU, we use an in-memory R-tree index to store trajectory data and evaluate its performance on a range of trajectory datasets. The R-tree is a well-studied out-of-core indexing scheme. However, in the context of in-memory searches, we find that the traditional notion of considering good trajectory splits by minimizing the volume of MBBs so as to reduce index overlap is not well-suited to improve the performance of in-memory distance threshold searches. Another finding is that computing good trajectory splits to achieve a trade-off between the time to search the index-tree and the time to process the candidate set of trajectory segments may not be beneficial when considering large datasets.

The GPU is an attractive technology for distance threshold searches because of the inherent data parallelism involved in calculating moving distances between pairs of polylines; however, challenges arise from the SIMD programming model and limited GPU memory. We study the processing of distance threshold searches using GPU implementations that avoid the use of index-trees. We focus on two scenarios for which we propose GPU-friendly indexing schemes for trajectory data. In the first scenario the database fits in GPU memory but the entire query set does not, thereby requiring back-and-forth communication between the host and the GPU. We find that our indexing scheme yields significant speedup over a multithreaded CPU implementation. We gain insight into our GPU algorithms via a performance model that proves accurate across a range of experimental scenarios. In the second scenario, we study the case where both the query set

and the database fit in GPU memory. With this relaxed memory constraint, the design space for trajectory indexing schemes is larger. We investigate indexing schemes with temporal, spatial, and spatiotemporal selectivity. Through performance analyses, we determine the salient characteristics of the trajectories that are conducive to the proposed indexes. In particular, we find that there are two complementary niches for GPU and CPU distance threshold search algorithms. Namely, using the GPU is preferable when the trajectory dataset is large and/or when the threshold distance is large. The GPU is thus the better choice for large-scale scientific trajectory dataset processing, as for instance in the motivating application for this work in the domain of Astrobiology.

Table of Contents

List of Tables

List of Figures

List of Acronyms

CNN Continuous Nearest Neighbor

CUDA Compute Unified Device Architecture

CPU Central Processing Unit

FSG Flatly Structured Grids

GHZ Galactic Habitable Zone

GPGPU General Purpose Computing on Graphics Processing Units

GPS Global Positioning System

GPU Graphics Processing Unit

IMF Initial Mass Function

k**NN** k Nearest Neighbor

MBB Minimum Bounding Box

MOD Moving Object Database

NFW Navarro, Frenk, and White

NN Nearest Neighbor

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PAPI Performance Application Programming Interface

RAM Random Access Memory

SIMD Single Instruction, Multiple Data

Chapter 1

Introduction

Applications in many domains require studying the movement of objects in space over time. Examples of such applications include those that collect data from , traffic monitoring, wireless communication networks, migration patterns of a given species, and tracking the motions of stars in astrophysical simulations. The generation of trajectories is application specific. For example, the trajectories of vehicles are obtained through monitoring traffic, trajectories of users are generated by tracking mobile devices in wireless communication networks, trajectories of animals are generated by monitoring the movements of species through remote sensing, and trajectories of stellar bodies arise from the laws of physics. Regardless of the manner in which trajectory data is generated, these applications process spatiotemporal trajectory datasets to gain insight into their target domains. In terms of processing trajectories, large-scale scientific simulations present great computational challenges in terms of computation and memory requirements as a result of the number of objects and trajectories that need to be processed.

Note that in the above applications, the focus is on *historical continuous* trajectories [15], where trajectory data is recorded and stored for processing. This is in contrast to works that focus on the near future motions of objects [4] (e.g., for the purposes of prediction), which do not record a trace of the positions of objects over a long time duration.

A trajectory is defined by a set of positions that describe the motion of a moving object over a time interval (i.e., x, y, z positions, and the corresponding time that the object was at each position). The continuous nature of trajectories requires that each point traversed by the trajectory be approximated by a polyline, where points are connected via line segments. In a given application domain, the goal will be to find trajectories with certain attributes in common, such as

similar spatial properties (short, long, curvature, etc.), proximity, clustering over time, and other spatiotemporal properties. Therefore, studying the motions of trajectories often necessitates the efficient implementation of *similarity searches*. For example, an often quoted example of trajectory data mining in the literature [14] is tracking the movement of endangered animals to determine when they are close to a static point, such as a food source, or nearby other animals over a time interval. Queries of this type are formulated as *k* Nearest Neighbor (*k*NN) searches on moving object trajectories over a particular time period. The continuous nature of the spatial and temporal dimensions of the data present additional challenges that set it apart from other solutions to the canonical *k*NN problem.

In this work, we focus on *distance threshold searches*, which find all trajectories, or stationary points within a distance d of a query trajectory over a time interval. An example search in the context of a zoology application is as follows: find all prey within a distance of 500 m of all predators. The distance threshold search can be viewed as kNN searches with an unknown value of k and thus unknown result set size. We presume that some of the lessons learned through the development of solutions for distance threshold searches can be applied more broadly to other spatiotemporal queries.

1.1 Motivation

We make contributions to a number of fields of computer science. However, one goal of this work is to simultaneously advance an application area. Unfortunately, often the development of computational solutions are not utilized within the application area that posed the problem. This occurs when computer scientists do not work directly with practitioners in other fields, thus resulting in a limited impact on the application area. In contrast, we aim to have a direct impact on the field of astrobiology by developing efficient algorithms for processing stellar trajectories, by eventually carrying out research in this domain.

A specific astrobiological application provides the initial motivation for this work. The study of the habitability of the Earth suggests that life can exist in a multitude of environments. Furthermore, the past decade of exoplanet searches implies that the Milky Way, and hence the universe, may host very large numbers of rocky, low mass planets. Therefore, if environments on rocky exoplanets are similar to those found on Earth, then we anticipate that the universe provides many habitable environments for life to thrive. The Galactic Habitable Zone (GHZ) is described as

the region in the Galaxy that may favour the development of complex life. The stars in the Milky Way have varying spatial and temporal distributions. These distributions have a major impact on the properties and locations of stars that we expect to host habitable planets. Furthermore, with regards to long-term habitability, some of these regions may be inhospitable to complex life due to transient radiation events, such as supernovae or gamma ray bursts. Combining knowledge and new data from the many constituent fields of astrobiology yield estimates, though in their infancy, of habitability on large scales.

Our previous work [19] investigated modeling the GHZ using a Monte-Carlo approach, by assigning stationary stars properties as derived from the major observational properties of the Milky Way. Using this information, we calculated the fraction of stars that we expect to have planets and study the frequency in which planets are sterilized by supernova events. Figure 1.1 shows one of the results in [19]. We trace the history of each habitable planet to determine the periods that they remain habitable, and plot those that are habitable at the present day (a birth date of 0 indicates that the star formed at the present day). For example, as shown on the figure, the Sun is located at a galactocentric radius of 8 kpc, and was born 4.5 Gyr ago. The region indicating "not enough time" suggests that young planets may not be conducive to the development of complex life as there may not have been enough time for biological evolution. The region indicating "insufficient metallicity" suggests that initial generations of stars will not have planets because there are not enough materials for planet formation. Habitable planets are most prevalent in the inner Galaxy, which is expanding outwards with time. We suggest that the inner disk of the Galaxy is expected to favor the emergence of complex life at $r > 2.5$ kpc. The majority of stars that host habitable planets in our model are expected to be older than the Sun, and the Sun is located in an unexceptional region.

In comparison to other works, we did not use a probabilistic model, and attempted modeling the morphology of the disk of the Galaxy more realistically by using 3-D instead of 2-D models and improved modeling of supernova sterilization events by populating stars individually. Therefore, moving forward with modeling the habitability of the Milky Way, we elect to continue modeling stars on an individual basis. Furthermore, the previous works on the GHZ have ignored the inner disk and bulge of the Galaxy. To model this region, it is crucial to model the orbits of stars, as they are largely on elliptical orbits in the galactic bulge. When considering stellar kinematics, an interesting question that has not been addressed arises, which is that of close encounters

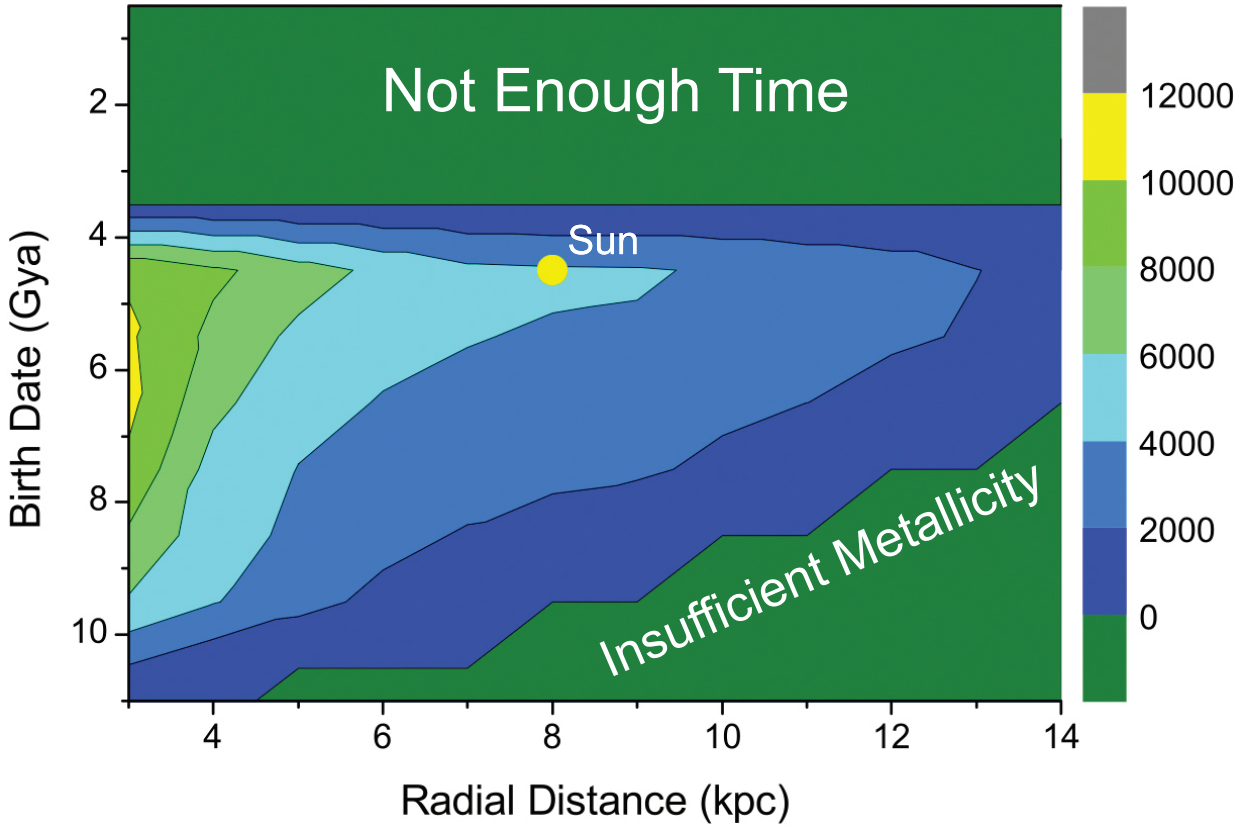


Figure 1.1. The number of habitable planets per pc is plotted as a function of radial distance (r) and birth date, from Figure 10, Model 4 of our previous work [19]. The Sun is located at 8 kpc from the Galactic centre.

between potentially habitable planets and dangerous flyby stars that can gravitationally perturb planetary systems. Studying habitability thus entails solving the following two types of historical continuous searches on the trajectories 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 distance threshold searches, as defined earlier. This work targets the efficient execution of distance threshold searches, and as such, will have a direct impact on the target astrobiology application.

1.2 Challenges

There are a number of challenges to mining moving object trajectories for the distance threshold search. Two key challenges are outlined as follows:

1. **Data Dependence** – Trajectories are defined by a series of points that trace an object's history. Given that objects move over time, the performance of the algorithms and proposed solutions are largely *data dependent*. For example, one trajectory may not be within a query distance d of any other trajectories over a time interval [t_0,t_1], and another trajectory may be within the query distance d of many trajectories over the same time interval. The performance goal in this work, as in other works focusing on querying spatiotemporal objects, is to minimize query response time. As the response time difference between queries may vary considerably, it is challenging to develop solutions that are efficient across a range of trajectory datasets and application scenarios.

2. **Locality** – Due to the above mentioned data dependence, trajectory processing algorithms have irregular memory access patterns. Thus, there may be few (if any) options available to achieve good spatial locality. Furthermore, there may not be a good method to process a set of queries to achieve memory accesses that benefit from exploiting the memory hierarchy through temporal locality of memory accesses. For example, over the course of processing a trajectory, the data required may change considerably and may rarely be reachable in faster components of the memory hierarchy, such as the CPU's cache.

   When considering multithreaded parallel implementations in shared memory environments, competition can arise between threads for space in CPU cache. If the threads require similar data elements from memory, then there will be low levels of unwanted cache eviction. However, if threads require divergent data items from main memory, which is likely due to the data dependence of the application, then higher levels of unwanted cache eviction will occur, thus decreasing query throughput.

   When considering implementations that utilize graphics processing units (GPUs), divergent memory access patterns between GPU threads will decrease the propensity for optimizing algorithms to benefit from coalesced memory accesses. In addition, the data dependence issue diminishes the capacity to schedule threads together that are likely to access similar

data elements. These two challenges are unavoidable, and suggest that trajectory processing algorithms may not be suitable for data-parallel architectures such as the GPU.

Given these challenges, a major target optimization of the distance threshold search is the development of efficient indexing schemes. However, query throughput also depends on the selectivity of the index in addition to considering data dependence and locality issues. Furthermore, proven indexes for the CPU architecture may not translate well to emerging architectures such as the GPU, thus necessitating alternate indexing schemes. We answer some of these underlying questions as part of this work.

1.3 Trajectory Processing Approaches

Numerous methods to index and process moving object trajectories efficiently have been developed by researchers in the field of spatial and spatiotemporal databases. Most works focus on *out-of-core* implementations where only part of the database resides in memory while the majority resides on disk. As a result, indexing techniques map nodes in index-trees to pages on disk, where the data layouts are optimized so as to reduce disk accesses. For processing trajectories, such indexes include the R-tree [21] and other variants of it, such as TB-trees [47], STR-trees [47], 3DR-trees [61], and SETI [12]. Additionally, systems have been designed to process and analyze trajectories, such as TrajStore [13] and SECONDO [20]. Index-trees have been used to index trajectories by defining them by their spatial, temporal, or spatiotemporal properties. For a given query trajectory, the index-tree is searched and those trajectory segments that overlap the query trajectory in one or more of its dimensions are returned as candidates. Next, the candidates are processed to find those that match the search criteria. In particular, the R-tree stores trajectories within minimum bounding boxes (MBBs), which treat the spatial and temporal dimensions identically. Thus, this index has been very successful based on the ability to search across all of the dimensions that describe an object. Since the motivation of the R-tree is to avoid disk accesses, the data layout is optimized for the memory and disk elements of the memory hierarchy.

Most previous work in the spatiotemporal database community focuses on out-of-core, sequential implementations. However, current architectures and memory capacities offer attractive alternatives, namely the ability to perform searches entirely in memory and in parallel. Many

modern machines have large memories, and data structures suited for indexing spatial and spatiotemporal objects can benefit from storing the entire index in-memory. In instances where the entire trajectory database cannot be stored in-memory, the current out-of-core solution is to swap pages of data to and from disk to process a given set of queries. Alternatively, by streaming large amounts of data between memory and disk, a better solution may be to partition the database and set of queries into batches temporally, and incrementally process the query set, by loading subsets of the database into memory at a time. This method of incrementally processing spatiotemporal data is only advantageous because commodity machines have large memory capacities. In instances where even larger memory capacities are required, then one can scale up to a distributed implementation over multiple nodes in a cluster where each node has CPUs and local memory. The database can be partitioned across a set of compute nodes. Then, it is often conceptually straightforward to parallelize (distance threshold) searches by replicating a query across all nodes, search the index-tree independently at each node, and aggregate the obtained results.

The scalability afforded by in-memory databases can take advantage of the multi-core CPU architectures available in off-the-shelf commodity machines. Threads running on their own cores can be assigned and process their own set of queries. Each thread is assigned a set of queries and searches the index-tree for possible candidate trajectories that may be within the query distance. The thread then processes the candidate set to find those trajectories within the query distance, d. Since each thread is processing its own set of queries, the query set is processed in parallel.

Beyond the use of multi-core CPU architectures, many-core GPUs have become mainstream, are programmable for general purpose computing, and should be well-suited to the large number of moving distance calculations required for processing distance threshold searches on trajectories in spatiotemporal databases. While there has been very little attention given to parallel solutions by the spatiotemporal database community that make use of multi-core CPUs, there has been even less attention given to exploiting the parallelism of GPUs, using the General Purpose Computing on Graphics Processing Units (GPGPU) paradigm. This is likely because the GPU is a new and quickly evolving architecture. Many open questions remain about future architectural designs and best practices for developing efficient GPGPU algorithms and implementations. Furthermore, the main algorithmic ideas and data layout designs that are well-suited to the CPU may not translate well to the GPU. For trajectory similarity searches, this means that a break from tradi-

tional indexing methods may be required to achieve good parallel efficiency when using the GPU. For example, memory management on the GPU is quite different than that of the CPU; therefore, scans of elements may be preferable to the complicated tree traversals previously proposed for out-of-core, sequential solutions. This work is a response to the emerging technologies available for trajectory processing by advancing alternative techniques to out-of-core, sequential solutions.

1.4 Contributions

In what follows, we outline the contributions made in this work:

- We motivate the use of an in-memory R-tree to index trajectory data for a CPU implementation and investigate efficient trajectory indexing strategies by exploring the trade-offs between the volume trajectories occupy in the index, the degree of index overlap, the number of entries in the index and the time to process the candidate set of trajectory segments.
- We demonstrate that high parallel efficiency can be obtained when using a multithreaded shared-memory implementation that relies on the sequential implementation above.
- Due to the data parallelism afforded by the GPU, we utilize the technology and processes queries in batches to address the memory limitations of GPUs. We advance a GPU-friendly indexing scheme conducive to these batches. Using the index, we develop a GPU kernel to perform the distance threshold search that minimizes branch instructions to achieve good parallel efficiency on the GPU. We compare our GPU implementation to the CPU implementation, and show that using the GPU can afford a significant speedup when using the GPU-friendly indexing technique that abandons index-trees.
- We gain insight into the above mentioned algorithms for the GPU by developing a performance model that that accurately estimates response time across a range of experimental scenarios. Thus, the model allows for selecting application parameters that lead to good response time.
- We expand on the GPGPU research thrust to address the scenario wherein the memory constraint is removed and compare the performance of three different indexing techniques for

the GPU that have temporal, spatial, and spatiotemporal selectivity. We show that the GPU yields a significant speed up over the CPU implementation in a range of experimental scenarios.

- Although not the focus of this work, we investigate the possibility of improving the efficiency of trajectory generation in an n-body astrophysical simulation by reordering computation to improve cache reuse.

Methods to process moving object trajectories have traditionally been advanced by the spatiotemporal database community. This work shows that some of the proposed optimizations do not scale to meet the demands of large-scale in-memory databases. To exploit emerging manycore architectures, such as the GPU, we find that traditional approaches should be abandoned. Instead, we advance novel methods to solve distance threshold trajectory similarity searches on these architectures, and are thus able to obtain significant performance gains in a wide range of experimental and application scenarios. We also demonstrate the remaining utility of existing methods through finding the two complementary niches for CPU and GPU distance threshold search algorithms.

The solutions developed in this work can be applied to applications outside of astrobiology. Moreover, there is a degree of overlap between different spatiotemporal searches; as such, the index-trees mentioned above have been applied to many different types of spatial and spatiotemporal searches. Similarly, we expect that the solutions developed in this work or the lessons learned in the process, will be applicable to other types of spatiotemporal queries, including other trajectory similarity searches.

1.5 Outline

In Chapter 2, we provide background information and describe related work. Chapter 3 describes the properties of the trajectory datasets that are used in this work. In Chapter 4 we demonstrate sequential and parallel CPU-based, in-memory solutions to the distance threshold search. Chapter 5 considers a GPU-friendly indexing scheme, which processes a query set in batches to decrease memory pressure on the GPU. Additionally, we advance a performance model of this execution. Chapter 6 evaluates different trajectory indexing techniques for the GPU with

spatial, temporal, and spatiotemporal selectivity. Finally, in Chapter 7 we summarize our findings and discuss future research directions.

Chapter 2

Background and Related Work

A key question in database research is the efficient retrieval of data. In the most general context, database management systems support arbitrary queries. However, in specific domains it is possible to achieve more efficient retrieval if there are structures and constraints on the data stored in the database and/or if particular types of queries are expected. Such a domain is that of spatial and spatiotemporal databases or moving object databases (MOD) that store the trajectories of moving objects. As discussed in Chapter 1, trajectory data, which arises in many scientific domains, presents both challenges and opportunities that are investigated in the spatiotemporal database community.

A trajectory is a set of points traversed by an object over time, where the trajectory is approximated by connecting the points via polylines (line segments). Trajectory databases store these points and/or polylines and aim to efficiently retrieve the data to answer queries on trajectories. Many classes of searches on trajectories are possible; however, there are two overarching classes that pertain to trajectories and other spatiotemporal objects: (i) *historical trajectory* searches which are concerned with the entire history of a trajectory [15, 22, 60, 64, 30, 42, 47, 58, 50, 44]. These searches are generally used to gain insight into an application area by studying a trace of the motions of objects, where large temporal intervals of the path followed by a moving object is of importance; and (ii) *near real-time searches* that are concerned with the near-future positions of moving objects, where a long historical trace is unnecessary as it becomes outdated [29, 66, 4, 48, 3]. In this work, we are concerned with the first type of search.

For searches on trajectories common searches are *trajectory similarity searches*, i.e., finding trajectories within a database that exhibit similarity in terms of spatial and/or temporal

proximity, or exhibit similarity in terms of spatial and/or temporal features so that trajectories can be classified as belonging to a certain group. Many kinds of similarity searches have been studied in various domains, such as convoy searches [24], flock searches [65], and swarm searches [35]. A predominant trajectory similarity search that is used in many application areas is the kNN (k Nearest Neighbors) search on trajectories [16, 14, 17, 20]. These different searches necessitate different similarity metrics. Therefore, a focus of this research area is efficiently indexing trajectories to suit the similarity metrics, as the indexing requirements for one search may not be applicable to another search. For example, some searches may be able to compute similarity criteria using an index with a coarse grained resolution [18], whereas other searches may require a high resolution index (which leads to a greater computational cost).

2.1 Spatiotemporal Databases

The fields of spatial and spatiotemporal databases have advanced a number of methods for solving queries regarding moving objects with varying spatial and temporal constraints. Performing a join with a moving object of interest on another set of moving objects is a typical type of query. The typical approach in spatiotemporal database works (and used for other types of indexed data) proceeds in two phases: (i) search an index to obtain a preliminary candidate result set; (ii) use refinement to produce the final result set. In general, an index is a data structure that improves response time by organizing the data in a way that allows for faster searches for data items. For example, consider a scenario where a database stores an unordered list of n events over the period of a year, where each tuple (date, event) contains the date and the event that occurs on that date. If a user wants to know what events occur on a particular day, then the database would need to scan all n events to find those that fall on the date. However, a simple indexing scheme could store the events in a data structure that partitions the tuples by month. Thus, the search only needs to scan a single bin corresponding to the month of the searched event. Assuming that the n events are uniformly distributed over the year, finding the items that fall on a particular day requires scanning $n/12$ tuples. This simple indexing scheme would thus achieve a (theoretical, upper limit) performance gain of a factor of 12.

Selectivity (also referred to as partition granularity) refers to the ability of an index to efficiently search for a set of objects that may meet the query criteria, where search performance is improved when relevant objects can easily be discerned from irrelevant ones. In our example

of an event database above, the index partitions the events by month. This may be reasonable for a small database of events. However, when considering larger databases, partitioning by month may not achieve enough selectivity. Therefore, in our example, it may be beneficial to re-index the database by partitioning events by day, thus achieving better selectivity.

When considering the query criteria, a search can be *pruned*, where part of the index is avoided as it is deemed to not contain any objects that should be part of the candidate set. For example, in a kNN point search, where $k = 2$, if there are two objects in the candidate set with a distance $d < 3$, it would be impossible for objects with $d > 3$ to be part of the candidate set; therefore, portions of the index could be avoided that place objects at a distance $d > 3$ of the query point object. Pruning a search depends upon the selectivity of the index and the characteristics of the query.

Depending on the constraints of a problem, finding the relevant candidate set of objects that may fit the query can be problematic. In general, the cost of an index-search is (at least partially) proportional to the dimensionality of the objects. With pure spatial objects, the dimensionality of an index search is the number of spatial dimensions (e.g., x, y and z dimensions in a 3-dimensional Cartesian coordinate space). However, when a temporal dimension is considered, where the objects move positions over time, the temporal dimension can be treated as either one of the other spatial dimensions, or as a different, non-spatial dimension. These cases are considered when developing efficient indexes for spatiotemporal searches on moving objects. For example, when considering episodic events, such as the near-future positions of an object, or a small time interval of the positions of an object, the temporal extent is short, and the query performance is mostly dependent on spatial selectivity. Conversely, if a long time duration is studied, such as in historical continuous searches on moving objects, then temporal selectivity may be more important than spatial selectivity.

As discussed above, the search phase focuses on *pruning*, i.e., avoiding parts of the index based on the selecting criteria of the query. To this end, several index-trees have been proposed for indexing trajectory data as inspired by the success of the popular R-tree [21], such as TB-trees [47], STR-trees [47], 3DR-trees [61], SETI [12], and implemented in systems such as TrajStore [13] and SECONDO [20]. More specifically, index-trees map nodes to pages stored on disk. Performance is largely a function of the number of index-tree nodes that are accessed, aiming to keep this number low so as to avoid avoiding costly data transfers between memory and disk. Thus, the

indexes have been developed within the milieu of traditional databases that assume a fraction of the index and data elements can be stored in memory, and the rest is stored on disk. To an extent, the optimizations proposed by the spatial and spatiotemporal database communities are tied to the underlying characteristics of this architecture.

By way of example, index-trees have been used for kNN searches on trajectories. R-trees, or other similar data structures, index spatial and spatiotemporal data using minimum bounding boxes (MBBs). One configuration is where each trajectory segment is contained in one MBB (where each segment is defined by two points that a trajectory has traversed). Leaf nodes in the R-tree store pointers to MBBs and the trajectory line segments that they contain (identified by the dimensions of the MBB and the trajectory ID). A non-leaf node stores the dimensions of the MBB that contains all of 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, thus producing a candidate set. Afterwards, the candidate set is filtered to produce a final result set. In what follows, we first review work on k Nearest Neighbors (kNN) searches, as they are related to distance threshold searches. We then review related work on distance threshold searches.

2.1.1 Nearest Neighbor Searches in Spatiotemporal Databases

Our work is related to, but as explained in later sections, also has major differences with the spatiotemporal kNN literature. We illustrate the typical kNN searches for four examples (see Figure 2.1):

$Q1$ 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].

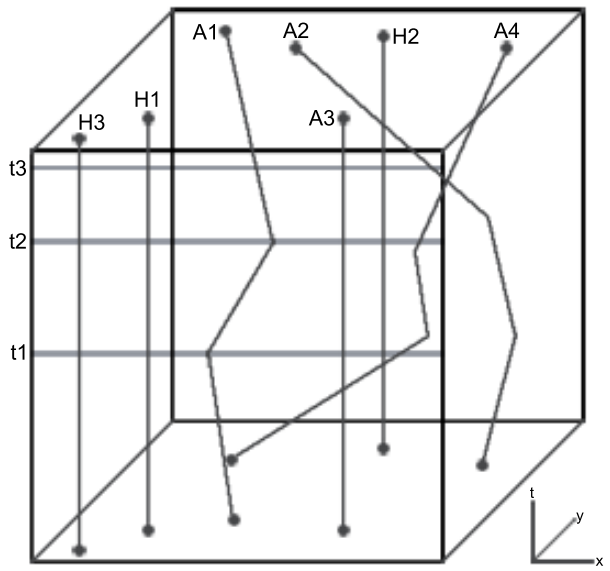


Figure 2.1. An illustration of trajectories and associated searches. Hospitals (H), ambulances (A), and time intervals between t_0 and t_4 are shown. Note that the vertical dimension refers to time in this example.

The example searches above are representative of the four main types of kNN 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 [55], the authors propose a method that relies on the R-tree to perform NN searches, and then generalize their approach to handle kNN 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 [57, 59]. 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 [57] 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 [59] avoids the computationally expensive drawbacks of [57]. 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 [7, 40, 39, 67, 69] 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 [16, 14, 17, 20]. 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 [47], the STR-tree [47], the 3DR-tree [61] and SETI [12].

Given the variety of query types shown above, many works have addressed the challenges of each scenario. The continuous moving query and continuous moving data scenario (Q_4) is of relevance to our work.

2.2 Distance Threshold Similarity Search

There are some queries which cannot be pruned as a function of the elements in the candidate set, such as queries with an unbounded number of elements in the result set. The distance threshold search, which is the focus of this work, that finds all trajectories within a query distance, d, of a query trajectory, is one such example of this type. For example, one search may return a large number of elements in the result set, and another search may not return any elements in the result set. In contrast, in a kNN search, assuming that there are more than k elements in the database, there will be a constant result set size of k elements. Thus, distance threshold similarity searches can be viewed as kNN searches with an unknown value of k and thus unknown result set size. As a result, several of the aforementioned index-trees, while efficient for kNN searches, are not efficient for distance threshold searches because, as k is unbounded, standard index pruning methods based on the elements in the candidate set cannot be used. In an out-of-core setting, there

would be no criteria to limit the search, and hence disk accesses, as the number of segments in the candidate set is unbounded.

While the kNN search on trajectories is used extensively in many applications, we depart from these searches and focus on distance threshold searches. Although distance threshold searches are relevant to several application domains, they have not received a lot of attention in the literature. The work in [5] solves a similar problem, i.e., finding trajectories in a database that are within a query distance d of a search trajectory, and focus on an out-of-core implementation. The authors propose four query processing strategies: one based on the R-tree, and three that use a plane-sweep approach. To the best of our knowledge that aforementioned work is the only other work that addresses distance threshold searches in the literature.

2.3 Parallelization of In-Memory Trajectory Searches

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 in-memory 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 [49] 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 [18]. 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 [70], 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 [72] 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.

Spatial and spatiotemporal data indexing methods have been advanced for use on the GPU [72, 71, 68, 36]. Given the SIMD nature of the GPU, proposed indexes for this architecture may be less sophisticated than index-trees used in the context of out-of-core databases. This is in part because branches in the instruction flow cause thread serialization and thus loss of parallel efficiency [23]. Several types of searches need to compute a large number of distance calculations, many of which can be performed in parallel, which makes many core architectures attractive targets. The kNN search is one such type of search, and as such has been studied in the context of the GPU [46, 28] and on hybrid CPU-GPU environments [33].

The trajectory similarity search studied in this work (Section 2.2) shares many of the same attributes as the kNN searches: they can be performed in parallel, and require many distance calculations, and thus the GPU is a good target architecture. However, in contrast, the distance calculation is more complicated than the distance calculations between point objects, as our work requires precise comparisons between individual polylines. Additionally, unlike the kNN search, our search cannot be pruned as described in Section 2.2. Lastly, there are additional challenges related to indexing trajectories in comparison to point objects.

Chapter 3

Trajectory Datasets

To elucidate the performance and behavior of the algorithms and implementations that are the focus of this work, we evaluate them using trajectory datasets with a wide range of properties (i.e. small, large, sparse, dense, and various temporal properties).

In this work, we utilize the following classes of trajectory datasets: i) datasets found in previous work; ii) synthetic random walk datasets; iii) a dataset motivated by stars moving in the gravitational field of the Milky Way; and, iv) a dataset of a galaxy merger. With the exception of (iii), in what follows, we describe high-level overviews of the datasets. More detailed descriptions of these datasets appear when they are first mentioned in their respective chapters.

3.1 Datasets in Previous Work

There are few datasets publicly available for testing our methods for two reasons: 1) they are often too small; and 2) many of the datasets available only consider 2 spatial dimensions. In contrast, our work is concerned with potentially large datasets with 3 spatial dimensions and 1 temporal dimension. Despite the drawbacks of existing datasets, we utilize a dataset of trucks moving in the Athens metropolitan area for 33 days [1], which has been used in other works [14, 17, 20]. This dataset contains 276 trajectories corresponding to 50 trucks.

3.2 Random Walk Datasets

We consider a series of 4-D (3 spatial dimensions + 1 temporal dimension) trajectory datasets that are generated from random walks. To elucidate the performance and behavior of the algorithms presented in this work, we vary the properties of these random walks. In one class of random walk datasets, the trajectories vary in terms of the straightness of the random walks. We consider a series of datasets that vary from the two possible extremes: on the one hand, a random walk that is characterized by a straight line and on the other, a random walk similar to that of Brownian motion. Furthermore, we also introduce random walk datasets with a range of start times. For example, all of the trajectories in a dataset may start moving at roughly the same time, or the entire dataset may exhibit periods of increased activity or inactivity. These temporal patterns are chosen as they reflect a range of temporal behavior that is characteristic of real-world datasets. We also consider random walk datasets that vary in terms of size, and density.

3.3 Galaxy Dataset

In this section, we outline the construction of a dataset used to create the trajectories of particles orbiting under the influence of the gravity of the Milky Way. To generate the trajectories of particles, we implement a gravitational field consistent with three major mass components of the Milky Way. We inject massless test particles into the gravitational field with an initial velocity and record the positions of the test particles throughout the simulation. The mass of the gravitational field remains static over time, and is constructed using discrete stationary mass points, hereafter referred to as pseudoparticles. These pseudoparticles are a representation of the mass of a given region. The three mass components considered are the bulge, disk, and dark matter halo. The bulge represents stars in the inner region of the Galaxy. The disk represents the majority of the stars in the Milky Way, and contains the Sun. The bulge and disk mass components represent the baryonic matter that we consider, where baryonic matter is composed of baryons (the matter we experience daily). The other mass component we consider is the dark matter halo (non-baryonic matter) that is expected to be the most abundant type of matter in the universe. The presence of dark matter has been inferred through interactions with baryonic matter. For example, dark matter is required

to explain the velocities of the stars in the Milky Way, particularly those towards the outskirts. We begin by outlining the construction of our mass models.

3.3.1 Baryonic Mass Components

We implement a bulge mass profile and a disk mass profile. The axisymmetric bulge model is implemented using a mass density profile. The disk mass profile is implemented using a stellar number density distribution and then the mass density distribution is derived by applying a stellar mass to each individual star using an initial mass function.

Mass Component: Axisymmetric Galactic Bulge

We create an axisymmetric profile of the Galactic bulge. We elect to implement the axisymmetric mass density profile of [37]. The axisymmetric mass model is an approximation of the non-axisymmetric model of [10]. The mass density profile is as follows:

$$\rho_b = \frac{\rho_{b,0}}{(1 + r'/r_0)^\alpha} exp[-(r'/r_{cut})^2], \tag{3.3.1}$$

where,

$$r' = \sqrt{R^2 + (z/q)^2}, \tag{3.3.2}$$

where r' is in cylindrical coordinates, $\alpha = 1.8$, $r_0 = 0.075$ kpc, $r_{cut} = 2.1$ kpc, with axial ratio $q = 0.5$, and a scale density of $\rho_{b,0} = 9.93 \times 10^{10}$ $M_\odot kpc^{-3}$. With these parameters, the total mass of the bulge in our model yields $\sim 8.9 \times 10^9$ $M_\odot$.

Mass Component: Galactic Disk

In this section, we outline the construction of the mass model of the disk of the Milky Way. The axisymmetric bulge above is described in terms of the mass density distribution. In contrast, for the disk of the Galaxy, we develop the mass density distribution by applying a mass from an initial mass function (IMF) to stars from a stellar number density distribution.

The IMF is thought to be nearly independent of environment [11]. Initial estimates of the IMF, such as that of [56] advocate a distribution that follows a power-law with $\alpha = 2.35$. [34] notes that there are uncertainties in the IMF, thus other IMFs have been proposed. [38], [31], and

[11] suggest that at subsolar masses, α should be lower than 2.35, flattening the IMF. Additionally, a study of the IMF of the bulge by [73] is consistent with an IMF that is nearly identical to that of the solar neighborhood, particularly that of [32] and [53]. We select the IMF of [31], which suggests a less steep IMF in the low mass star range than that of [56].

We assign stellar masses using the Monte Carlo technique such that the resulting distribution follows a power-law and matches an IMF which is described as a two part power-law function given by [31]. The value $\alpha = 1.3$ when $0.08 \leq \mathrm{M} < 0.5$, and $\alpha = 2.3$ when $\mathrm{M} \geq 0.5$, where M is the stellar mass (as a fraction of the Solar mass). The maximum and minimum stellar masses for main sequence stars are defined as 100 $\mathrm{M}_{\odot}$ and 0.08 $\mathrm{M}_{\odot}$ respectively [32], where $\mathrm{M}_{\odot}$ is the mass of the Sun.

We model the stellar number density distribution of the disk as consistent with our previous work in [19]. In particular, we implement a single stellar number density distribution of the disk, that of [27] as follows:

$$\rho_D(R, Z) = \rho_D(R, Z; L_1, H_1) + f\rho_D(R, Z; L_2, H_2), \tag{3.3.3}$$

where,

$$\rho_D(R, Z; L, H) = \rho_D(R_{\odot}, 0)e^{R_{\odot}/L} \times e^{\left(-\frac{R}{L} - \frac{Z+Z_{\odot}}{H}\right)}. \tag{3.3.4}$$

The quantity ρ_D is the number of stars per unit volume in pc^3. The coordinate Z is the vertical height above or below the midplane of the Galaxy, and R is the radial distance from the Galactic centre, and $R_{\odot} = 8$ kpc is the galactocentric distance to the Sun. We utilize the values $H1$ = 300 pc, $L1$ = 2600 pc, $H2$ = 900 pc, $L2$ = 3600 pc, and f = 0.12, corresponding to the thin disk scale height and length, the thick disk scale height and length and the thick-to-thin disk density normalization.

Furthermore, given that the Kroupa IMF is found to be observationally consistent with the IMF of the disk, as we find the local number density is $\sim$70% of that found by [52], we do not vary the IMF. We normalize the distribution of stars based on an estimate of the total disk mass in the Milky Way. The disk mass estimate of [9] yields $4.2 \times 10^{10}\mathrm{M}_{\odot}$ in disk stars ($0.3 \times 10^{10}\mathrm{M}_{\odot}$ has been subtracted, as it corresponds to gas mass). Using the Kroupa IMF, we are able to match our disk mass ($4.2 \times 10^{10}\mathrm{M}_{\odot}$) by normalizing $\rho_D(R_{\odot}, 0)$=0.084 stars pc^{-3}.

3.3.2 Mass Component: Dark Matter Mass Density profile

The baryonic matter in the Galaxy constitutes only part of the total matter in the Milky Way, and a small fraction of the total matter in the universe. To properly model the orbits of stars in the Galaxy, we implement a dark matter halo in cold dark matter cosmology.

While the mass accretion history of dark matter in the Galaxy will have an effect on the dynamics of our test particles over time, we elect to use a fixed dark matter profile. We believe that this method is reasonable, as the majority of the stars in the Galaxy are influenced to a greater extent by baryonic matter rather than dark matter, where the baryonic matter has a decreasing influence on the orbits of stars as a function of distance from the galactic center. Furthermore, the disk stars in the outskirts of the Galaxy, which are those stars most affected by dark matter, are relatively young, and form after the dark matter halo has accreted most of its present day mass. Therefore, our fixed dark matter profile should permit a broadly consistent present day velocity curve for our model of the Milky Way, despite the absence of evolution.

In [37], the author constrains the mass of the Milky Way using observational and kinematic data. We use the properties of one of their models to implement a dark matter mass distribution in our model. The dark matter halo is fit in [37] using the Navarro, Frenk, and White (NFW) profile [43]. The dark matter halo density is modeled as a sphere whose origin is the Galactic center.

The NFW dark matter density profile as a function of galactocentric radius is as follows:

$$\rho(r) = \frac{\rho_s}{(r/r_s)(1 + (r/r_s))^2}, \tag{3.3.5}$$

where r_s and ρ_s are the characteristic radius and density on the sphere, respectively. From [43],

$$r_s = \frac{r_{200}}{c}, \tag{3.3.6}$$

where c and r_{200} are the NFW concentration parameter and virial radius respectively. The virial radius (r_{200}) is the radius of a sphere where the average density of the sphere is greater than a factor of 200 of a critical density (ρ_{crit}). The virial mass (M_{200}) describes the total mass within the virial radius. The orbits of celestial objects are influenced by the mass contained within this radius. The concentration parameter c describes the concentration of the inner region of the dark

matter sphere. Galaxies that form in the early universe have a high concentration c, as they form in an environment with a higher mean background density, whereas later forming galaxies have a lower concentration.

In [37], the dark matter halo has $r_s = 20.2\ \mathrm{kpc}$, $M_{200} = 1.40 \times 10^{12}\ \mathrm{M}_\odot$, and a present day concentration, $c \approx 9.545$. Additionally, [37] finds $R_\odot = 8.29\ \mathrm{kpc}$. To utilize the halo density profile in Equation 3.3.5, we calculate r_{200} from Equation 3.3.6, which yields $r_{200} \approx 193\ \mathrm{kpc}$. To match the M_{200} and r_{200} of the halo described above, we assign a value of $\rho_s = 9.341 \times 10^6\ \mathrm{M}_\odot\mathrm{kpc}^{-3}$. Following [37], this results in a local dark matter density of $\rho_{h,\odot} = 0.011\ \mathrm{M}_\odot\mathrm{pc}^{-3}$ in our model. The dark matter density distribution is shown in Figure 3.1.

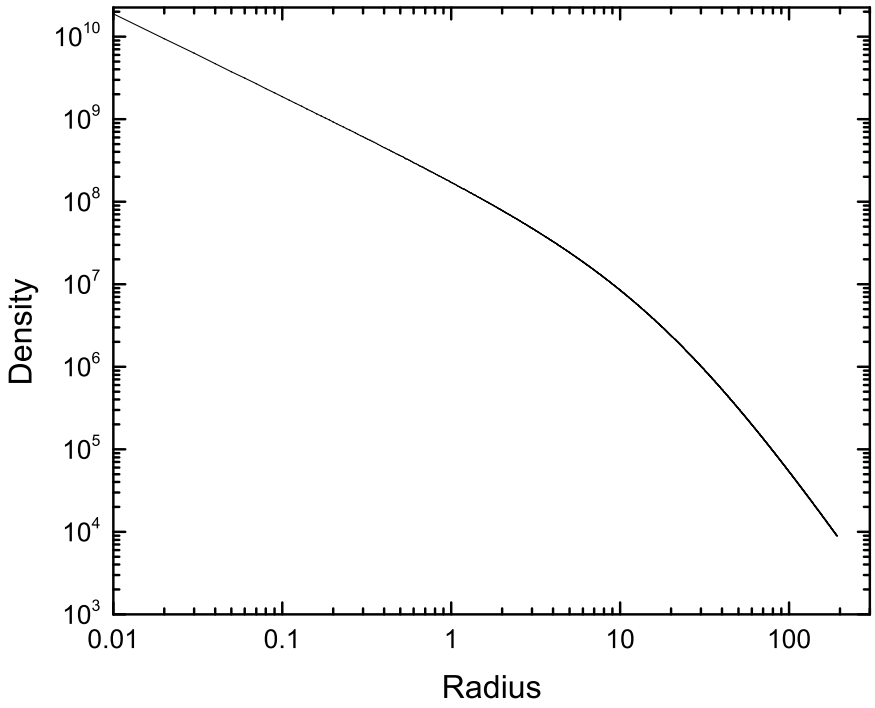


Figure 3.1. The dark matter density profile as a function of radius (kpc) on a sphere in our model in units of $\mathrm{M}_\odot$ kpc^{-3}.

We note, however, that the calculation of ρ_s is often utilized with the following equation [43]:

$$\delta_c = \frac{\rho_s}{\rho_{\mathrm{crit}}}, \tag{3.3.7}$$

where δ_c is the characteristic NFW overdensity, and is converted into the NFW concentration c with

$$\delta_c = \frac{200}{3} \times \frac{c^3}{[ln(1+c) - c/(1+c)]}. \tag{3.3.8}$$

The critical density ($\rho_{\rm crit}$) depends on the the mean background density at a particular time with the following equation [43]:

$$\rho_{crit} = \frac{3H^2}{8\pi G}, \tag{3.3.9}$$

where H is the Hubble constant. We did not utilize these formulations for our calculation of ρ_s as we are only interested in the present day dark matter profile which was normalized as described above.

The baryonic matter also contributes to the mass of the halo of the Milky Way. For example, [6] finds that the stellar halo is $\sim 3.7 \times 10^8 M_\odot$. This mass in negligible in comparison to the mass of the dark matter halo. Therefore, we do not create a mass model of the stellar halo as we expect it to have little impact on the dynamics of our model Milky Way galaxy.

3.3.3 Mass Model Visualization and Terminology

In this section, we illustrate the implementation of the mass models. Each mass component (bulge, disk and dark matter halo) extends to different volumes; therefore, three different arrays of structures are implemented to store the pseudoparticle data. For each mass component, we partition the space to assign a mass to each pseudoparticle. For the bulge and disk mass components, the stars are populated into their respective volumes, an initial mass function is applied, and a total mass is created for that volume. The dark matter mass density profile is implemented by segmenting the volume and applying the mass density distribution to assign a mass to each pseudoparticle. Figure 3.2 shows an example segmentation of the space in two dimensions. The red region indicates a cell. The red dots indicate the center of a cell by volume and are the location of the respective pseudoparticles. Finally, the red arrow is the length of a single radial cell extent. The vertical cell extent is not shown in Figure 3.2, but refers to the height of a vertical cell.

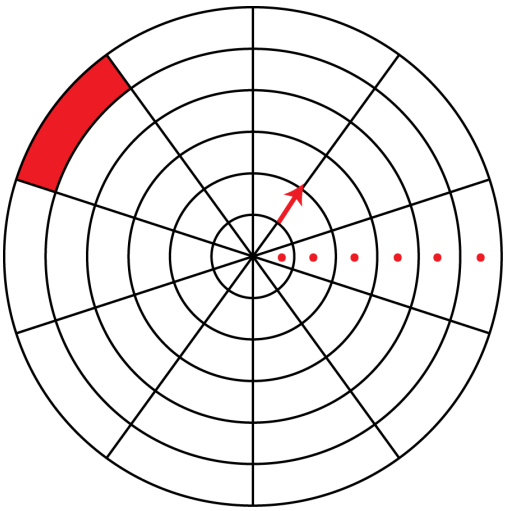


Figure 3.2. Visualization of the mass mass models in two dimensions. Red region demarcates a cell, red dots are the locations of pseudoparticles, and arrow length is the radial cell extent. The vertical cell extent, not shown here, is the vertical height of a cell.

3.3.4 Dynamics and N-body Simulation Parameters

In a direct force n-body simulation, the force on a particle i from a particle j is calculated as follows:

$$F_{ij} = \frac{(Gm_i m_j)(r_j - r_i)}{(|r_j - r_i|^2)(|r_j - r_i|)}, \tag{3.3.10}$$

where G is the gravitational constant, m_i and m_j are the masses of the particles respectively, and r_i and r_j are the positions of the particles. The left factor in the equation gives the magnitude of the force and the right factor gives the direction.

To suppress strong gravitational interactions between test particles and close pseudoparticles, the above equation is modified as follows:

$$F_{ij} = Gm_i m_j \frac{r_j - r_i}{(|r_j - r_i|^2 + \epsilon^2)^{3/2}}, \tag{3.3.11}$$

where ϵ is called the softening length. Note that in our prototype application, the forces calculated using the equation above are the total forces of each pseudoparticle on each individual test particle.

3.3.5 Calculation of the Softening Length

An adequate softening length (ϵ) depends upon the distribution of particles in the simulation. We outlined the construction of the mass distribution throughout the Milky Way, which is comprised of disk, bulge and dark matter mass components. We calculate the softening length using the method of [54], which calculates it in the densest region of Plummer and Hernquist spheres. Furthermore, the softening length is calculated to be the average distance between the nearest neighbor of each particle.

We calculate the nearest neighbor of each particle within a radius which is given by the mass model with the largest radial cell extent in the innermost region of our model of the Milky Way[1]. Furthermore, we limit the maximum height of this volume of pseudoparticles to be the height of the bulge, which approximates the maximum vertical height that a particle is found to orbit in our model. Limiting the height ensures that we are not calculating the nearest neighbors of dark matter pseudoparticles distributed outside the disk and bulge mass components. Pseudoparticles with a distance of 0 between them, or those pseudoparticles located at the same position are excluded from the nearest neighbor search. Given the nearest neighbor of each pseudoparticle, we calculate the average distance between these pseudoparticles, and assume that a test particle, on average, orbits at the midpoint between two pseudoparticles, thus reducing the mean distance by a factor of 2. Additionally, [54] find that the softening length should be between 1.5-2$\times$ smaller than the average distance between nearest neighbor particles. Therefore, we choose a softening length that is 1.75$\times$ smaller than this value. The softening length in our model is calculated as follows:

$$\epsilon = \frac{\text{Average distance between pseudoparticles}}{2 \times 1.75}. \tag{3.3.12}$$

3.3.6 Generation of the Galaxy Dataset

Trajectories are generated by injecting test particles into the gravitational field (a function of the pseudoparticles) with an initial velocity. The motions of these test particles are tracked and recorded. We create a series of trajectory datasets of different sizes from the motions of these test

[1]For example, if the radial extents of the dark matter halo, disk and bulge are 1 kpc, 0.5 kpc and 0.3 kpc respectively, then the nearest neighbor of each pseudoparticle within a radius of 1 kpc is calculated.

particles, denoted *Galaxy*, which we use throughout this work. Figure 3.3 illustrates the motions of 30 trajectories from one of the datasets.

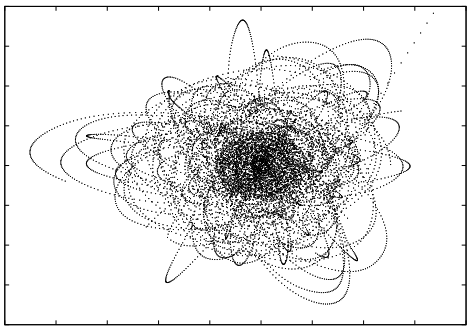
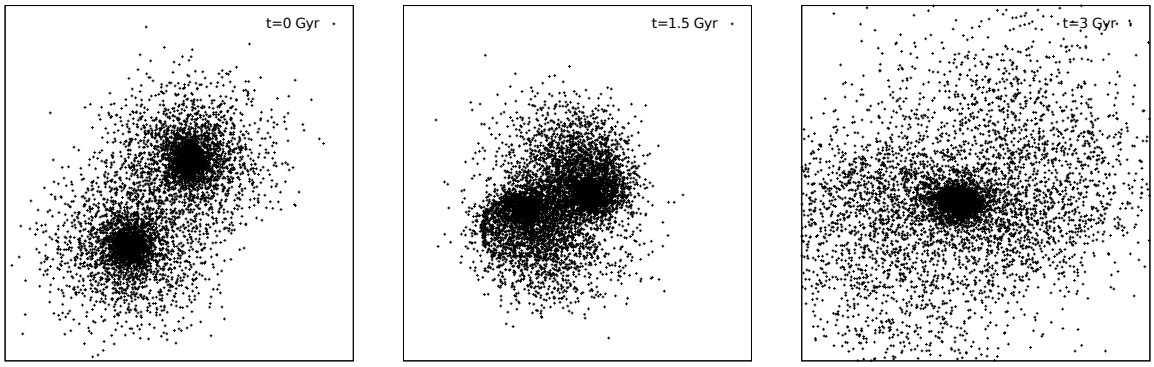


Figure 3.3. A sample of 30 trajectories from the Galaxy dataset.

3.4 Merger Dataset

Our last dataset is from the output of a simulation of a galaxy merger that contains disk and halo components[2]. We limit the dataset to contain the trajectories of the disk particles in the simulation. This dataset has been scaled from simulation units to kpc so as to represent a radial disk scale length consistent with a Milky Way-like galaxy. Figure 3.4 shows a sample of the particles in the Merger dataset at three time periods: (a) at the beginning of the simulation at 0 Gyr, (b) at 1.5 Gyr, and (c) at the end of the simulation at 3 Gyr.

[2]Dataset obtained from Josh Barnes.



(a) (b) (c)

Figure 3.4. Sample particle positions in the Merger dataset at times 0 Gyr (a), 1.5 Gyr (b) and 3 Gyr (c).

Chapter 4

CPU Indexing Scheme and Algorithms for Distance Threshold Searches

We propose two distance threshold search algorithms for the CPU and motivate the use of the R-tree [21] to index the database of trajectories instead of alternative index-trees. Furthermore, as motivated in Chapter 2, the index, and data elements that are pointed to by the index are all stored in main memory. One drawback of circumscribing objects in MBBs is that the object (trajectory) occupies an infinitesimal volume in comparison to the MBB. Therefore, we propose methods to filter out line segments that are not part of the final result set. We propose decreasing index resolution to exploit the trade-off between the volume occupied by the trajectories, the degree of index overlap, the number of entries in the index, and the number of candidate trajectories that need to be processed by exploring three trajectory splitting strategies. We find that, for our in-memory searches, lower-bounding the index resolution is more important than minimizing the volume of MBBs and thus index overlap. Finally, we investigate a multithreaded implementation that uses OpenMP and demonstrate that high parallel efficiency can be achieved.

4.1 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 interval [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 interval [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.

4.2 Trajectory Indexing

One trajectory indexing approach utilizes index-trees. These indexes store trajectory data in tree nodes. In this work, we define a trajectory as a set of connected trajectory segments over some temporal extent. Trajectory data (a series of trajectory segments) can then be stored in MBBs, where an MBB describes the spatial and/or temporal properties of the trajectory. After the database of trajectories is generated, a trajectory search can proceed. Given a search for some query trajectory over some temporal extent, one considers all relevant query MBBs. That is, a series of query trajectory segments are circumscribed by MBBs and then the MBBs are *augmented* in all spatial dimensions by the threshold distance d. One then searches for the set of trajectory segment MBBs in the database that overlap with the augmented query MBBs, since the overlapping MBBs may contain trajectories that are in the result set. Efficient indexing of the trajectory segment MBBs can thus lower query response time. After the index is searched, the candidates are refined to produce the final result set.

As discussed in above and in Section 2.1, index-trees [16, 14, 17, 20] have been utilized for spatiotemporal queries. Several index-trees have been proposed (TB-tree [47], STR-tree [47], 3DR-tree [61]). 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 [55]). While this is sensible for kNN 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.

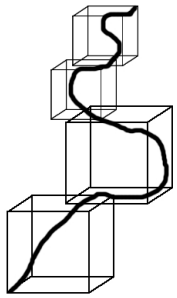


Figure 4.1. An example trajectory stored in different leaf nodes in a TB-tree.

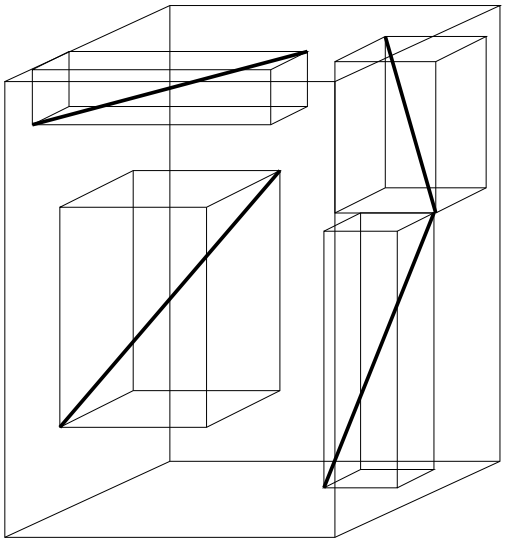


Figure 4.2. Four line segments belonging to three different trajectories within one leaf node of an R-tree.

Let us consider for instance the 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 selectivity, which means that the index aims to efficiently retrieve line segments based on the temporal properties of the query. Figure 4.1 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 4.2 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 [47] show that the TB-tree performs better than the R-tree (for kNN searches) especially when the number of indexed entries is low. In this work, we are interested in large MODs (see Section 1.1). We conclude that an R-tree index should be used for efficient distance threshold search processing.

4.3 Search Algorithm

We propose an algorithm, TRAJDISTSEARCH (Algorithm 1), to search for trajectories that are within a threshold distance of a query trajectory. 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 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 [20]. The moving distance between two line segments is derived in Appendix B. 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.

Algorithm 1 Pseudo-code for the TRAJDISTSEARCH algorithm (Section 4.3).

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

4.4 Initial Experimental Evaluation

4.4.1 Datasets

As described in Chapter 3, our first dataset, *Trucks* [1], is used in other MOD works [14, 17, 20]. It contains 276 trajectories corresponding to 50 trucks. This is a 3-dimensional dataset

Table 4.1. Characteristics of Datasets

Dataset	Trajectories	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

(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 Chapter 3). 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). As discussed in Chapter 3, our third dataset is a class of 4-dimensional synthetic datasets, *Random*, with trajectories generated via random walks. An adjustable parameter, α, is used to control whether the trajectory is a straight line ($\alpha = 0$) or a Brownian motion trajectory ($\alpha = 1$). We vary α 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 [62]. We do not use GSTD because it does not allow for 3-dimensional spatial trajectories.

Figure 4.3 shows a 2-D illustration of the *Galaxy* and *Random* datasets. An illustration of *Trucks* can be found in previous works [14, 17]. Table 4.1 summarizes the main characteristics of each dataset.

4.4.2 Experimental Methodology

We have implemented algorithm TRAJDISTSEARCH in C++, reusing an existing R-tree implementation based on that initially developed by A. Guttman [21], and the code is publicly available [2]. We execute the sequential implementation on one core of a dedicated Intel Xeon

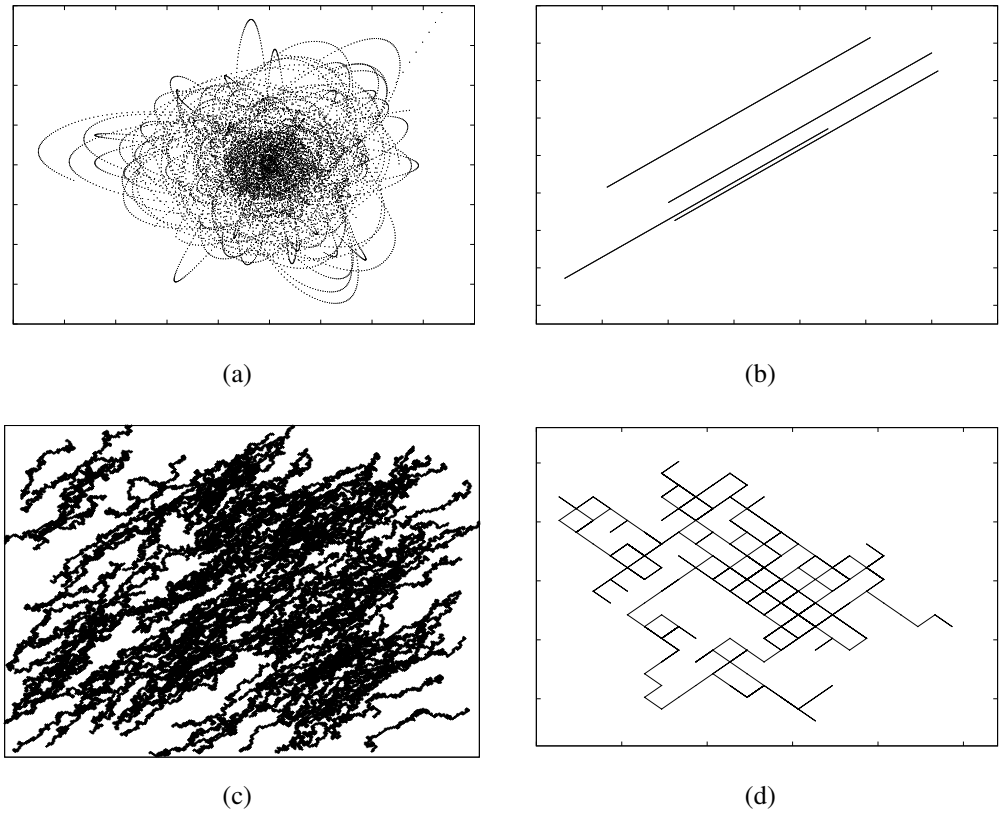


Figure 4.3. (a) *Galaxy* dataset: a sample of 30 trajectories, (b) 4 trajectories in the *Random* dataset with $\alpha = 0$, (c) 200 trajectories in the *Random* dataset with $\alpha = 0.8$, (d) a sample trajectory in the *Random* dataset with $\alpha = 1$.

X5660 processor, at 2.8 GHz, with 12 MB L3 cache and sufficient memory to store the entire index. 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. The implementations have been validated to ensure correctness. To guarantee that we do not obtain false positive or negative results, we compare the results of our implementation to an alternate implementation that utilizes a brute force approach.

4.4.3 Static Point Search Performance

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

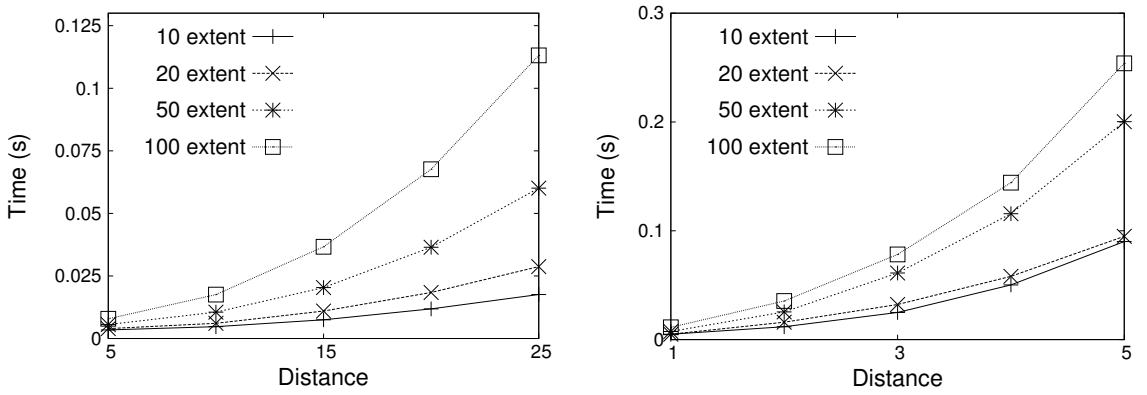
- P1: From the *Random*-1M $\alpha = 1$ dataset, 500 random points are selected with 10%, 20%, 50% and 100% of the temporal extent of the trajectories in the dataset, for various query distances.
- P2: Same as P1 but for the *Galaxy*-1M dataset.
- P3: From the *Random*-1M, 2M, 3M, 4M, 5M $\alpha = 1$ 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 4.4 (a) and 4.4 (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 4.5 (a) and 4.5 (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 4.5 (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.

4.4.4 Trajectory Search Performance

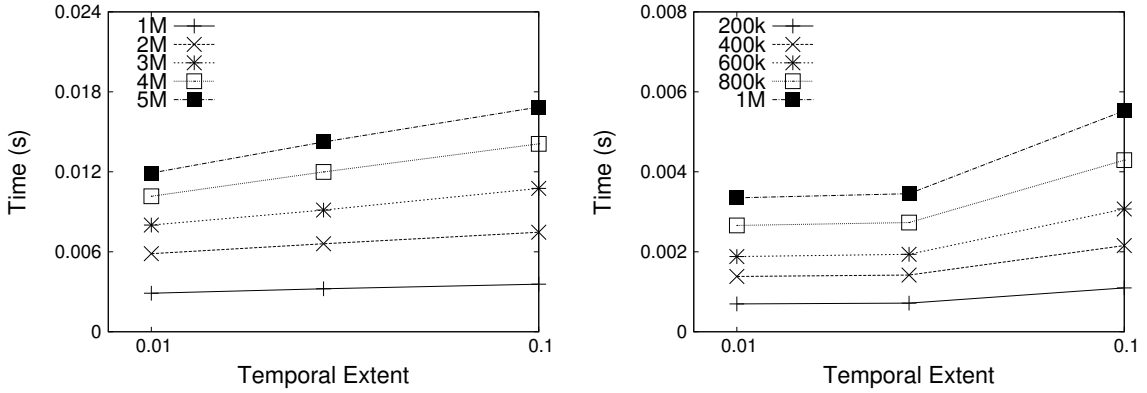
We measure the query response time of TRAJDISTSEARCH for the following sets of trajectory searches:

- 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.



(a) *Random* $\alpha = 1$ (b) *Galaxy*-1M

Figure 4.4. 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).

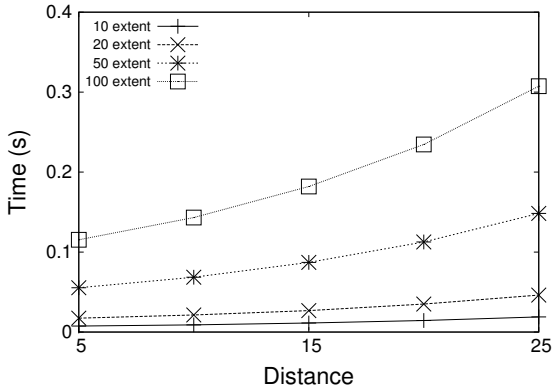


(a) *Random* $\alpha = 1$ (b) *Galaxy* datasets

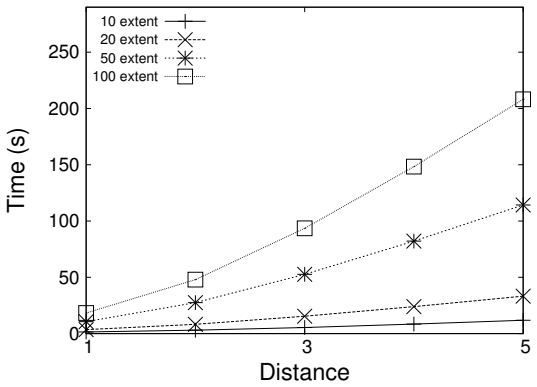
Figure 4.5. Query response time vs. various temporal extents of the trajectories in the datasets. (a) P3 using the *Random*-1M $\alpha = 1$ datasets; (b) P4 using the *Galaxy* datasets.

- 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 4.6 (a) and 4.6 (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 4.7 (a) and (b) show response time vs. query distance and temporal extent

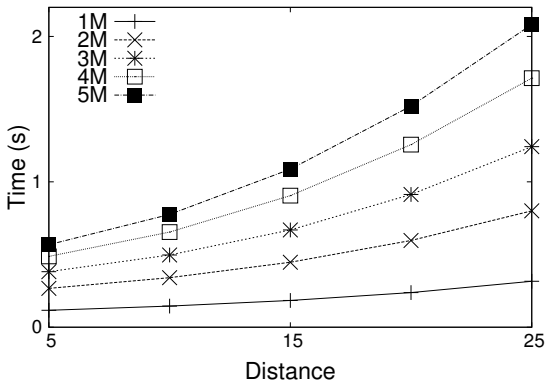


(a) *Random* $\alpha = 1$

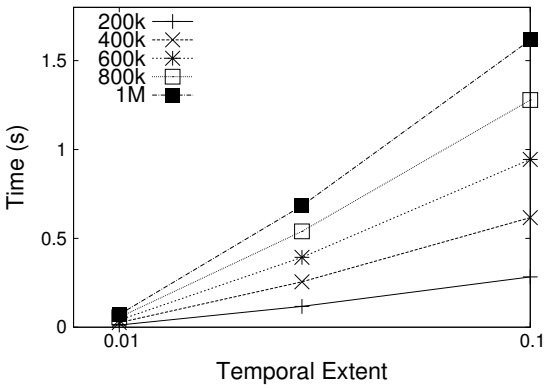


(b) *Galaxy*-1M

Figure 4.6. 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 4.7. (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.

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 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

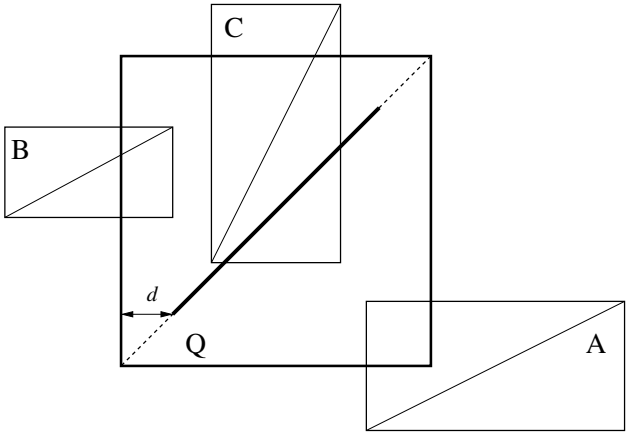


Figure 4.8. Three example entry MBBs and their overlap with a query MBB.

algorithm degrade with increasing query distance, temporal extent, or dataset size. In the next sections we address optimizations to reduce response time further.

4.5 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). 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 Algorithm 1). 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 4.8 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 1, 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 4.9 shows the number of candidate segments returned by the index search and the number of segments that are within the query distance vs. α, 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. Note that we observe an increase in the number of segments within the threshold distance as α increases because there is a greater chance of Brownian motion-like trajectories being within the query distance of each other in comparison to trajectories that are on the straighter end of the trajectory spectrum (approaching $\alpha = 0$).

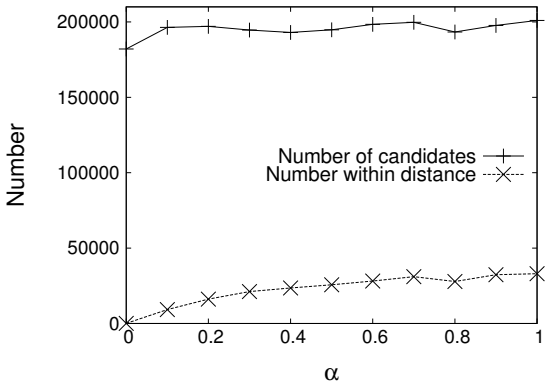


Figure 4.9. The number of moving distance calculations and the number that are actually within a distance of 15 vs. α in the *Random*-1M datasets.

4.5.1 Two Segment Filtering Methods

After the query and entry line segments are interpolated so that they have the same temporal extent (Algorithm 1, line 6), various criteria may remove the candidate segment from consideration. We consider two filtering methods beyond the simple no filtering approach:

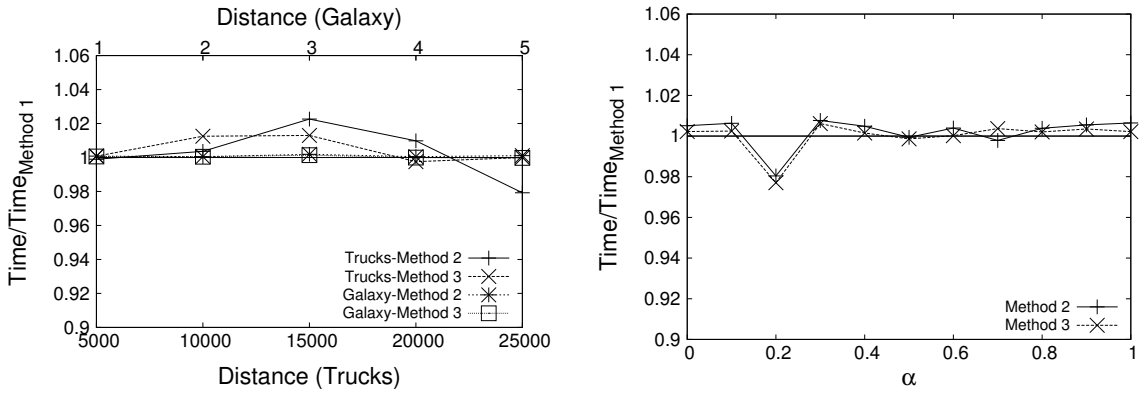
- **Method 1 –** No filtering.
- **Method 2 –** After the interpolation, check whether the candidate segment still lies within the query MBB. This check only requires floating point comparisons between spatial coordinates of the segment endpoints and the query MBB corners, and would occur between lines 6 and 7 in Algorithm 1. Method 2 would filter out A in Figure 4.8.
- **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 Algorithm 1, 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 4.8.

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.

4.5.2 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 $d = 15$.



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

Figure 4.10. Performance improvement ratio of filtering methods (a) for real datasets with S5 and S6, vs. query distance, (b) for *Random*-1M datasets with S7.

Figure 4.10 (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 4.10 (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 4.9. We profiled the execution of TRAJDISTSEARCH for searches S5, 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.

4.6 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 [45], 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 line segment of a trajectory in its own MBB, as done so far in this paper and in previous work on kNN searches [16, 14, 17, 20]. 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.

In Figure 4.11 (a) we depict an entry trajectory that is stored with each segment in its own MBB, in Figure 4.11 (b), a trajectory that is stored in a single MBB, and in Figure 4.11 (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 4.11 (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 4.11 (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 4.11 (a), the

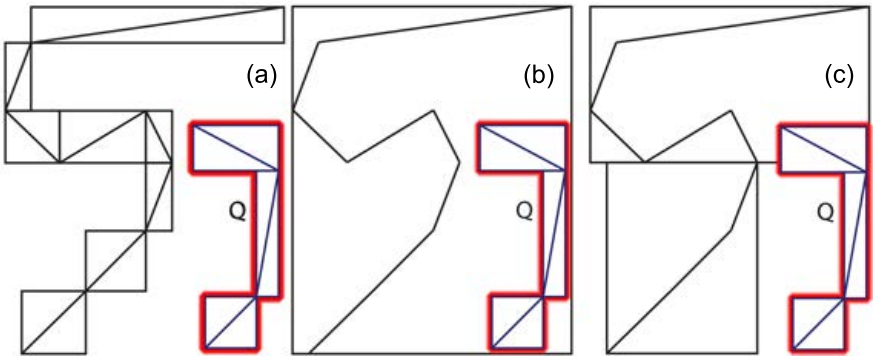
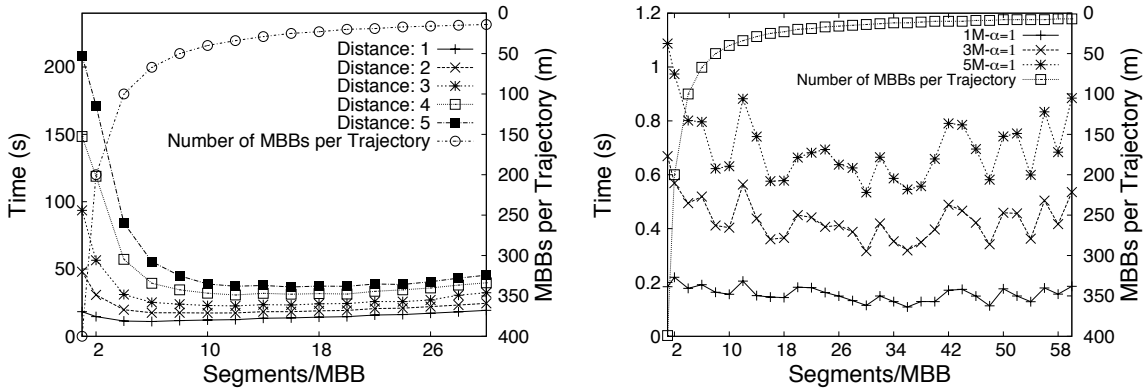


Figure 4.11. Illustration of the relationship between wasted space, volume occupied by indexed trajectories, and the number of returned candidate segments to process. An 8-segment trajectory is indexed in three different ways, and searched against a 3-segment query trajectory (denoted Q in the figure), 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.

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 4.11 (b). Figure 4.11 (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 $1 \times 5 = 5$ 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 each trajectory 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



(a) *Galaxy*-1M (b) *Random* datasets

Figure 4.12. 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.

indexes, where the configuration with the lowest query response time is highlighted. Analytical performance models of trajectory splitting methods are outside the scope of this work.

4.6.1 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 4.2).

Figure 4.12 plots response time vs. r for the S6 (*Galaxy* dataset) and S7 (*Random* dataset) searches defined in Section 4.5.2. 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 4.12 (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 4.12 (a) and (b) are largely due to significantly more query hits in *Galaxy* in comparison to *Random* for the query distances selected.

4.6.2 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 4.13 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 4.12 (b) we find that spatial splitting leads to query response times higher by several factors than that of temporal splitting.

4.6.3 Splitting to Reduce Trajectory Volume

The encouraging results in Section 4.6.1 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.

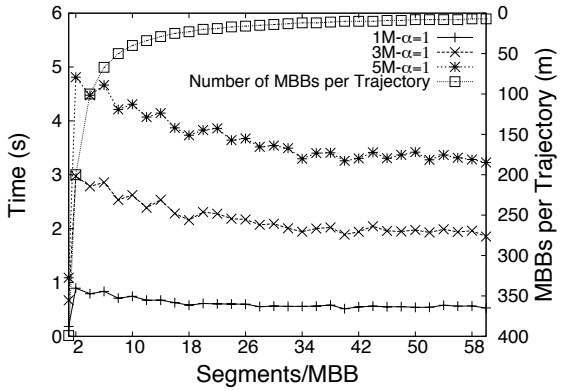


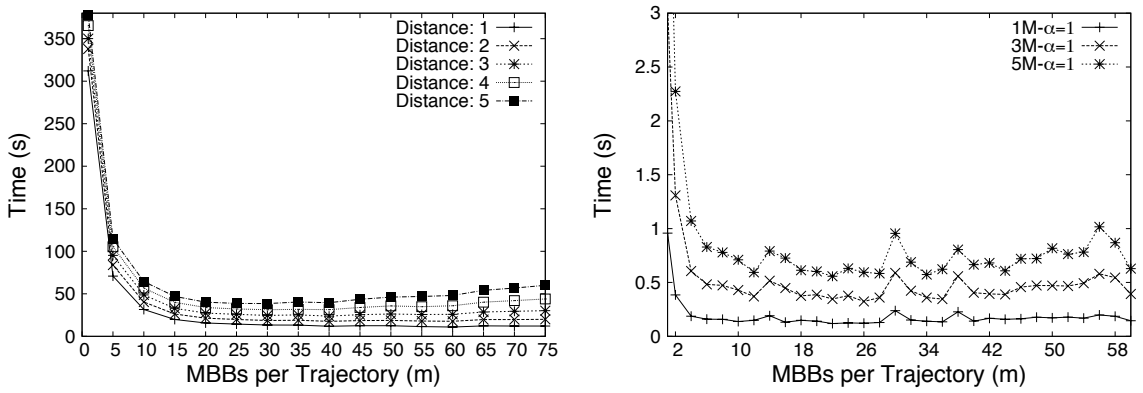
Figure 4.13. Static Spatial Splitting: Response time vs. r using S7 for the *Random*-1M, 3M, and 5M $\alpha = 1$ datasets and a query distance of 15. The number of MBBs per trajectory, m, for each data point is shown on the rightmost vertical axis.

More sophisticated methods can be used. In particular, we implement the heuristic algorithm *MergeSplit* in [22], 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:

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 4.14 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 4.12 and 4.14 (a)), the best performance for 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 4.12 and 4.14 (b)).



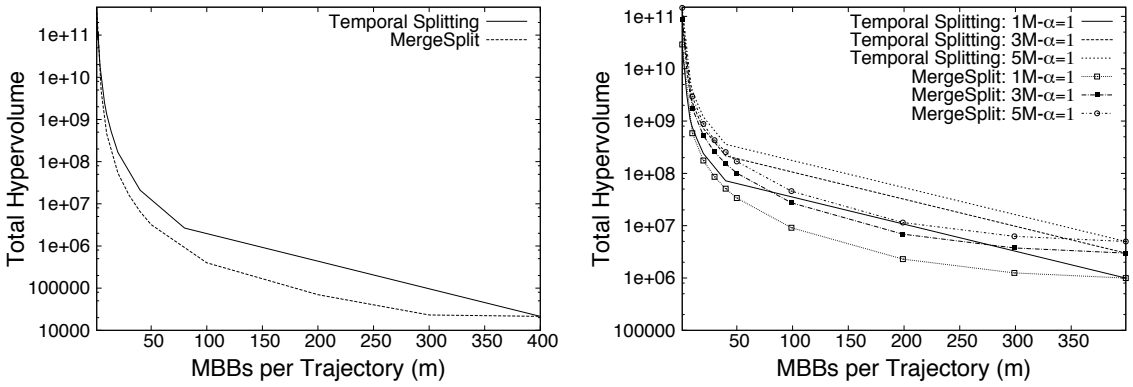
(a) *Galaxy*-1M (b) *Random* datasets

Figure 4.14. 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.

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. We discuss these trade-offs in the following section.

4.6.4 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 4.6.1) and *MergeSplit* (Section 4.6.3), we find that volume minimization did not significantly improve performance for S7, and led to worse performance for S6. In Figure 4.15, 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 3.6×10^7, whereas for *MergeSplit* at $m = 30$, it is 1.62×10^7, i.e., 55%

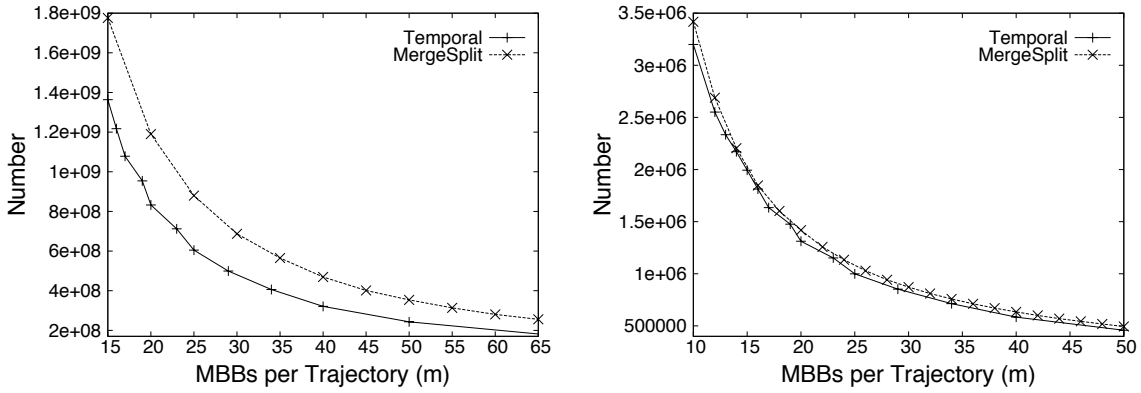


(a) *Galaxy*-1M (b) *Random*-1M

Figure 4.15. 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).

less volume. Due to the greater volume occupied by the MBBs, index overlap is much higher for the static temporal splitting strategy. Figure 4.16 (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 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., [22]).

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



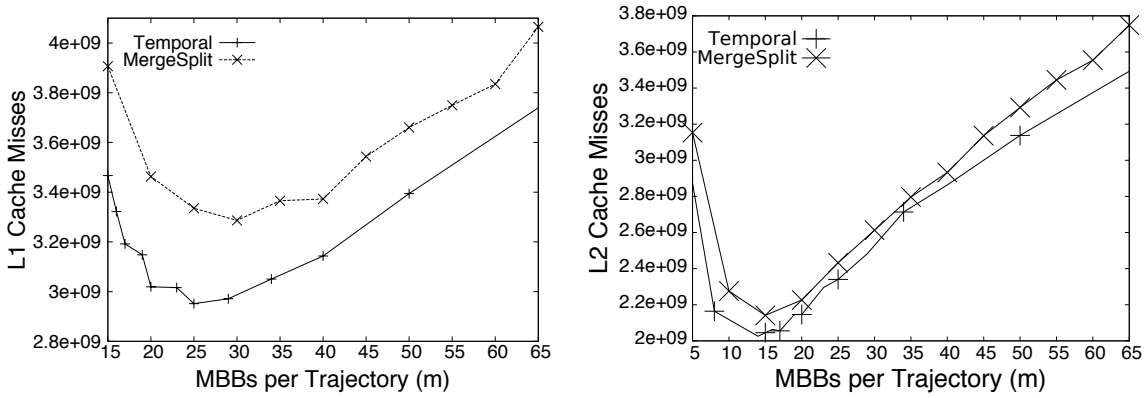
(a) *Galaxy*-1M (b) *Random*-1M

Figure 4.16. 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 $d = 15$.

(Figures 4.12 (a) and 4.14 (a)). Figure 4.17 (a) shows the number of L1 cache misses vs. m for S6 with $d = 5$. The number of cache misses was measured using PAPI [41]. 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 4.17 (b) shows the number L2 cache misses vs. m for S6 with $d = 5$. We note that when comparing Figure 4.17 (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).

4.6.5 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



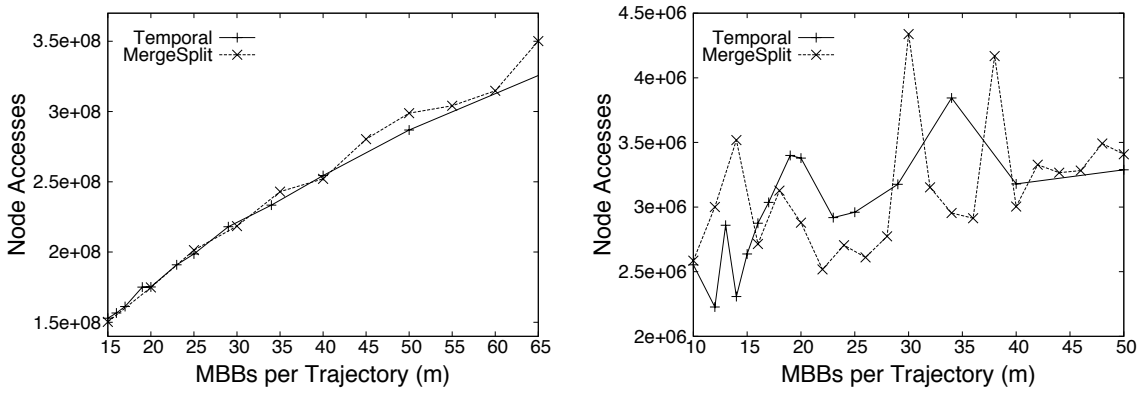
(a) *Galaxy*-1M (b) *Random*-1M

Figure 4.17. L1 (a) and L2 (b) cache misses vs. m for the static temporal splitting strategy and *MergeSplit* for the *Galaxy*-1M dataset (S6) with $d = 5$.

are used as a metric to estimate I/O activity. Figure 4.18 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 perform significantly better than the simple temporal splitting strategy in an out-of-core implementation. For example, in Figure 4.18 (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 4.6.4, 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 [22], or other works that consider volume reduction in the context of query sizes, such as [51], may not be entirely applicable to distance threshold searches.

4.6.6 Multi-core Execution with OpenMP

In Section 4.6.4, 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.

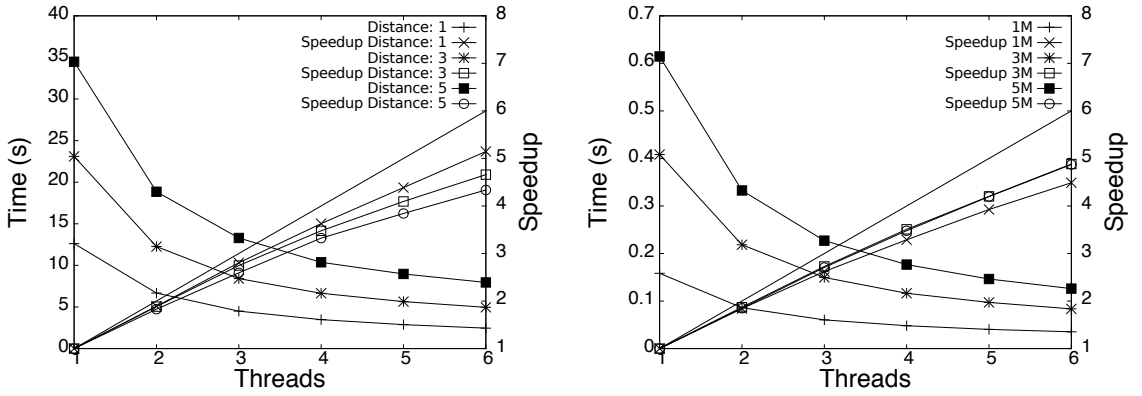


(a) *Galaxy*-1M (b) *Random*-1M

Figure 4.18. 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 $d = 15$.

The iterations of the loop on line 3 of TRAJDISTSEARCH in Algorithm 1 are independent, each iteration can be assigned to a different thread. In what follows, we show results up to 6 threads, which corresponds to the 6 cores on the CPU on the platform. Figure 4.19 shows the response time 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 4.12). 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 4.19 (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.

Distance threshold searches, and perhaps other spatiotemporal searches on trajectories can be parallelized in a straightforward manner in a shared-memory environment because the



(a) *Galaxy* (b) *Random* datasets

Figure 4.19. 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 $r = 10$.

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.

4.7 Conclusions

In this chapter, we proposed an implementation motivated by the milieu of spatiotemporal databases that performs query processing using the CPU. In contrast to most works in the spatiotemporal database community, we considered a database that is entirely in memory. Therefore, the target of optimization is at faster levels of the memory hierarchy. We find that trajectory splitting strategies should not necessarily focus on volume reduction for in-memory databases. This is because volume reduction does not limit the number of trajectories that may be stored in a single MBB; therefore, a bounded grouping of trajectories may be preferable for in-memory databases. We demonstrate that a potentially good metric for assessing trajectory splitting strategies is L1 cache misses. Finally, we illustrate that high parallel efficiency is obtained using a multithreaded OpenMP implementation.

Chapter 5

GPU Indexing Scheme and Algorithms for Memory-Constrained GPGPU Distance Threshold Searches

Given the arbitrarily large number of moving distance calculations required for distance threshold searches, the GPU is an attractive alternative to the CPU. However, memory on the GPU is limited in comparison to the amount of main memory that may be available on the host. In this chapter, we advance algorithms and an index for a GPGPU execution of the distance threshold search. To fit within memory constraints, we incrementally process a query set in batches by invoking a series of kernels. To perform this batched execution, our proposed methods consider the non-negligible overhead of kernel invocations. Additionally, our GPU-friendly indexing scheme and associated GPU kernel are tailored to this batched execution. The performance of the batching strategy is dependent upon creating efficient query batches; therefore, we develop several algorithms to create such batches. Finally, we develop an empirical response time model for a periodic batched execution that can predict query response time by a reasonable margin, which makes it possible to estimate a good batch size. The performance of this implementation is compared to the performance of the CPU implementation described in Chapter 4. We find that the GPU yields a significant speedup over the CPU implementation.

5.1 General Purpose Computing on the Graphics Processing Unit

Unlike the CPU, which has few cores and threads that are executed in a multithreaded implementation (i.e., one thread per physical core), the GPU has many cores and can run hundreds of threads concurrently. The CPU focuses on executing individual threads very quickly; whereas, the GPU attempts running many threads, each with a slower execution speed. The GPU is attached to the PCI express bus, which is a present-day bottleneck in GPGPU computing. To execute a program to be run on the GPU, the *host* program, which runs on the CPU, sends instructions and data to the GPU over the PCI express bus. The program is executed by the GPU and it is called a *kernel*. When the GPU is finished, the host retrieves the results from the GPU over the bus. The host has a much higher bandwidth between CPU and main memory. Therefore, there is an overhead to using the GPU, which makes the architecture not efficient for all applications.

There are two frameworks used to program the GPU. There is the Compute Unified Device Architecture (CUDA) framework, developed by NVIDIA, that can be used to program NVIDIA GPUs. The other framework is the Open Computing Language (OpenCL) framework, that is developed by the broader community and can be used to program heterogeneous architectures in general. In this work, we develop our implementations in OpenCL. However, when referring to the architecture of the GPU and the logical representation of the framework, we use the more common CUDA terminology (GPU as opposed to device, kernel as opposed to program, thread as opposed to work-item, etc.).

Figure 5.1 illustrates the architecture of the GPU. Each GPU contains a memory space. Also, each GPU contains multiple multiprocessors, each of which have multiple scalar cores that perform the computations. The scalar cores execute the *threads* within the single instruction, multiple data (SIMD) environment.

Figure 5.2 illustrates the conceptual memory hierarchy in CUDA. Global memory is the largest of the memory spaces and is the only memory space that can be accessed by the host to retrieve the results of a job that has finished running on the GPU. In most programs, the majority of the data is stored in global memory. Threads are executed in batches of *blocks*. For each block, there is a shared memory space that can be utilized by the threads. Therefore, thread synchro-

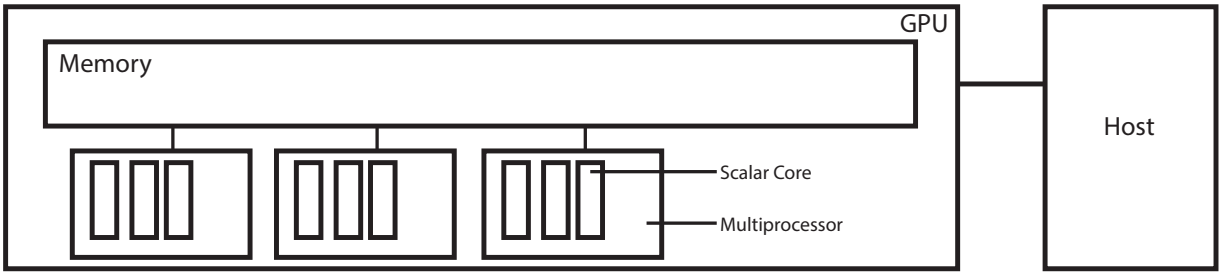


Figure 5.1. A semi-physical representation of the architecture of the GPU in the CUDA nomenclature.

nization is only possible within a block, and not between blocks. There is no guaranteed order of execution amongst threads because the scheduling of blocks onto cores is not transparent to the developer. Each thread has its own *local* memory space.

Threads in a conventional CPU may execute different sections of code concurrently. In contrast, in the SIMD architecture, threads belonging to the same warp (each warp has 32 threads) execute together in lock-step at each instruction. Branch instructions, such as an if-statement, can cause divergence in the instruction flow within warps. If this occurs, the threads executing together serialize, which causes a loss of parallel efficiency. Therefore, branch conditions that are expected to have lots of divergence (as a function of the data elements) should be avoided if possible. This architectural feature is an additional reason why the GPU may not be suited to all applications.

5.2 Problem Definition

Let D be a spatiotemporal database that contains n 4-dimensional (3 spatial dimensions + 1 temporal dimension) line segments. A line segment l_i, $i = 1, \ldots, n$, is defined by a spatiotemporal starting point (x_i^{start}, y_i^{start}, z_i^{start}, t_i^{start}), a spatiotemporal ending point (x_i^{end}, y_i^{end}, z_i^{end}, t_i^{end}), a segment id and a trajectory id. Segments belonging to the same trajectory have the same trajectory id and are ordered temporally by their segment ids. We call $t_i^{end} - t_i^{start}$ the *temporal extent* of l_i and the line segments in D, *entry segments*.

We consider *historical continuous searches* that search for entry segments within a distance d of a query Q, where Q is a set of line segments that belong to a moving object's trajectory. We call the line segments in Q *query segments*. The search is continuous, such that an entry seg-

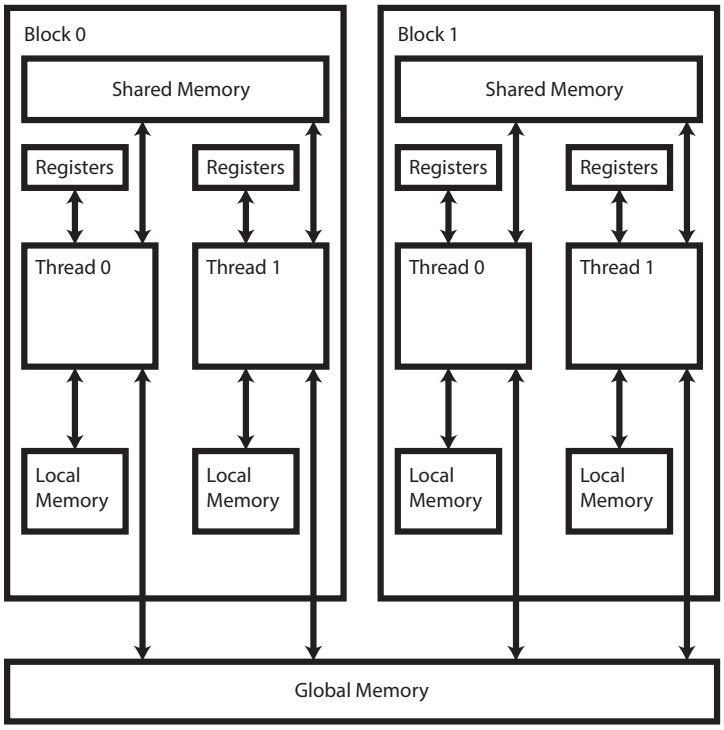


Figure 5.2. Conceptual CUDA memory hierarchy.

ment may be within the distance threshold d of particular query segment for only a subinterval of that segment's temporal extent. The result set thus contains a set of entry segments, and for each segment, a time interval. For example, for a query segment with temporal extent [0,1], the search may return (l_1,[0.1,0.3]) and (l_2,[0.6,0.9]).

We consider a platform that consists of a host, with RAM and CPUs, and a GPU device with its own RAM (global, shared, and local) and compute units. We consider an *in-memory database*, meaning that D is stored once and for all in global memory on the GPU. We focus on an *online* scenario where the objective is to minimize the response time for an arbitrary set of queries. This is the typical objective considered in other spatiotemporal database works such as the ones reviewed in Chapter 2. We consider the case in which D and Q cannot fit together on the GPU, with a twofold rationale. First, the memory on the GPU is limited and in practice a single database is subjected to a large number of queries. Second, memory for the result set must be allocated statically since dynamic memory allocation is not permitted on the GPU. However, the result set size is non-deterministic and depends on the spatiotemporal nature of the data. As a result, memory

allocation for the result set must be conservative and overestimate the amount of memory required. This overestimated size grows linearly with $|Q|$, thereby creating even more memory pressure on the GPU. For these two reasons we partition Q in batches that are processed in sequence. Such incremental query processing is also useful when multiple users query the database simultaneously, and would thus compete for memory space on the GPU. Note that by using relatively small batch sizes, the yielded result set fits within the memory of the GPU.

5.3 Trajectory Indexing

Many efficient indexing methods have been proposed in the spatiotemporal database literature assuming that processing takes place on a CPU. The GPU architecture is markedly different from that of the CPU, in particular due to its SIMD execution model. Therefore, limiting the amount of conditional branching in GPU implementations is important to achieve good performance. As a result, efficient CPU implementation approaches (which can benefit from branch prediction techniques) are likely to be vastly inefficient when applied directly to the GPU.

Chapter 4 uses an in-memory R-tree index for processing distance threshold searches on the CPU. For a given query, the search phase of the computation finds candidate segments as stored in MBBs in the leaf nodes of the R-tree, and the refine phase reduces these candidates to find those that should be part of the result set. The majority of the computation is spent in the search phase, which has many branch instructions to follow R-tree node pointers from the root to the leaves, which should be avoided on the GPU. Similar observations have been made in the literature when indexing spatial and spatiotemporal databases on the GPU [72, 71]. The authors in [72] note that it is not clear whether index-trees should be used at all. The work in [72, 71] utilizes grid files, or "flatly structured grids," data structures in which polylines are converted to MBBs and are assigned to cells on a grid to spatially partition and index the data as an alternative to using index trees. In this work, we design a GPU-friendly indexing method for scenarios in which large query sets must be partitioned into batches that are processed iteratively.

In light of the above, we propose the following approach to index the database. We first sort the entry segments by non-decreasing t_{start} values. Without loss of generality we assume that the entry segments are numbered in that order (i.e., $t_i^{start} \leq t_{i+1}^{start}$). The full temporal extent of database D is $[t_{min}, t_{max}]$ where $t_{min} = \min_{l_i \in D} t_i^{min}$ and $t_{max} = \max_{l_i \in D} t_i^{max}$. We divide

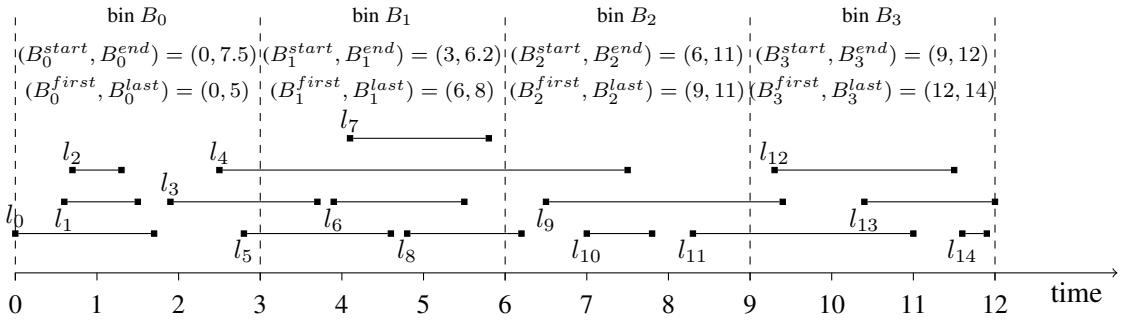


Figure 5.3. Example indexing of line segments into bins.

this temporal extent logically into m bins of fixed length $b = (t_{max} - t_0)/m$. We say that an entry segment l_i, $i = 1, \ldots, n$, belongs to bin B_j, $j = 1, \ldots, m$, if $\lfloor t_i^{start}/b \rfloor = j$. For bin B_j we can then define $B_j^{start} = j \times b$ and $B_j^{end} = \max((j+1) \times b, \max_{l_i \in B_j} t_i^{end})$. We call $[B_j^{start}, B_j^{end}]$ the temporal extent of bin B_j. We then define $B_j^{first} = \arg\min_{i|l_i \in B_j} t_i^{start}$ and $B_j^{last} = \arg\max_{i|l_i \in B_j} t_i^{start}$. $[B_j^{first}, B_j^{last}]$ is thus the index range of the entry segments in B_j. Bin B_j is thus fully described as (B_j^{start},B_j^{end}, B_j^{first}, B_j^{last}). The set of bins is the "index" of the database.

Figure 5.3 shows an example for a database with 14 entry segments along the time axis with a total temporal extent of 12 (segments are simply shown as non-overlapping horizontal lines as we do not depict their spatial dimensions or orientations). The temporal extent of the database is logically divided into 4 bins, and for each bin we indicate the B^{start},B^{end}, B^{first} and B^{last} values. For instance, bin B_1 contains the three entry segments with t^{start} in the $[3, 6)$ interval, i.e., l_6,l_7 and l_8. Therefore, $B_1^{first} = 6$ and $B_1^{last} = 8$. Among the three entry segments in bin B_1, l_8 has the highest t^{end} value at 6.2. Therefore, $B_1^{start} = 3$ and $B_1^{end} = 6.2$.

Given the database and set of bins, we consider a query set Q. We first sort the query segments by non-decreasing t_{start} values in $O(|Q| \log |Q|)$ time, which gives the temporal extent of the query (the combined temporal extent of the query segments). We then determine the set of (contiguous) bins that temporally overlap the temporal extent of the query. We do this determination in $O(\log m)$ time by using an index-tree in which we store the bins' temporal extents. Given this set of bins, $\mathcal{B}$, we compute $first = \min_{B \in \mathcal{B}} B_j^{first}$ and $last = max_{B \in \mathcal{B}} B_j^{last}$ in $O(1)$ time. We thus obtain $E_Q = \{l_i \in D | first \leq i \leq last\}$, the set of the candidate entry segments that may

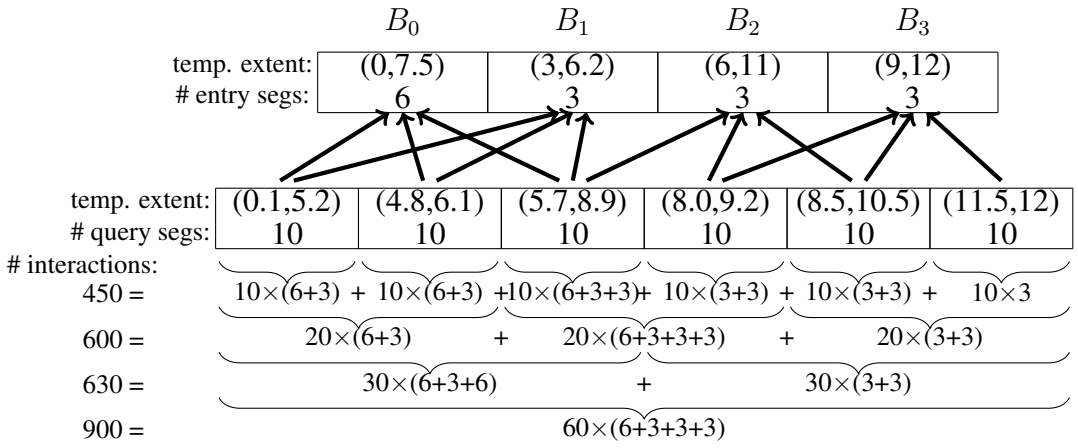


Figure 5.4. Example matching between query batches and entry bins.

be part of the result set. Each query segment must then be compared to each candidate segment in E_Q. We term each such a comparison an *interaction*, and we have a total of $|Q| \times |E_Q|$ interactions.

Some of the computed interactions are certain to not add the candidate to the result set. For instance, in the context of the example in Figure 5.3, consider a query with a single query segment with temporal extent [8,10]. The query segment overlaps the temporal extents of bin B_2 and bin B_3, meaning that it will be compared to $l_9, \ldots, l_{14}$. And yet, l_{10}, l_{13}, and l_{14} cannot overlap the query segment's temporal extent. More generally, the larger $|Q|$, the larger the number of interactions, and thus the larger the number of "wasteful" interactions. This observation provides a motivation for processing query segments in relatively small batches (in addition to the fact that using batches is necessary because memory for the result set must be allocated statically–see Section 5.2).

Figure 5.4 shows an example of how using batches decreases the number of interactions. The top of the figure shows the same set of bins as in Figure 5.3, without showing the entry segments but indicating temporal extents and numbers of entry segments. The bottom of the figure shows a set of 60 query segments partitioned into 6 batches. For each query batch we indicate its temporal extent and its number of segments. An arrow is drawn between a query batch and an entry bin if the query segments in the batch must be compared to the entry segments in the bin. Below the batches we show the number of interactions necessary to process the query. For instance, batch 2 has a temporal extent (5.7,9,1), which overlaps with the temporal extents of bins B_0, B_1, and B_2, which contain 6, 3, and 3 entry segments, respectively. Therefore, the processing of batch 2

entails 10×(6+3+3)=120 interactions. Using 10-segment query batches results in a total of 450 interactions. The figure also shows the number of interactions using larger batches. For instance, while processing 10-segment batch 2 requires 120 interactions and processing 10-segment batch 3 requires 60 interactions, processing the aggregate 20-segment batch leads to 20×(6+3+3+3) = 300 $>$ 180 interactions. In this example, processing all query segments as a single 60-segment batch would lead to 900 interactions, twice the number of interactions when using 10-segment batches. Processing each query segment individually (batch size of 1) minimizes the number of segment interactions. However, processing each batch incurs the non-negligible overhead of sending data from the host to the GPU and of invoking a GPU kernel. Consequently, one of the questions we investigate in this work is that of choosing batch sizes that minimize query response time.

Note that more advanced indexing methods could be envisioned that inform each individual entry what queries temporally overlap to avoid computing wasteful interactions. However, these methods lead to more data transfer overhead between the host and the GPU. Preliminary results show that in practice this overhead leads to significant increases in total response time in spite of reducing the number of wasteful interactions.

Given the above, we propose the following general approach for implementing distance threshold searches on the GPU. The entry segments in D, sorted by non-decreasing t^{start} values are stored contiguously in the global memory of the GPU. The database index, i.e., the description of the bins, and the query segments in Q (sorted by non-decreasing t^{start} values) are stored in RAM on the host. The query segments are partitioned in batches (not necessarily of identical sizes). For each batch, the index range of the candidate entry segments is calculated using the bins. The query segments in a batch and the index range, which is encoded as two integers, are sent from the host to the GPU. The candidate entry segments are then compared to the query segments, generating a result set that is returned to the CPU. Our indexing method guarantees that these candidate entry segments are stored contiguously in memory, which allows for efficient memory transfers between global, local and private memory spaces on the GPU, and which reduces the use of branches. The search is complete when all batches have been processed in this manner. In Section 5.4 we describe our GPU kernel for performing the search, while in Section 5.5 we describe approaches for picking good batch sizes.

5.4 Search Algorithm

In this section we describe an algorithm, GPUTRAJDISTSEARCH, that performs distance threshold searches using the indexing and search techniques outlined in Section 5.3. This algorithm is implemented as a GPU kernel using OpenCL, and optimized to use as few branch instructions as possible. To take advantage of the high number of hardware threads on the GPU and of its fast context-switching we simply use one GPU thread for each candidate entry segment. Each thread then compares its candidate entry segment to all query segments in the batch. Using Q_{batch} to denote a query batch, which is a subset of Q, each thread then computes $|Q_{batch}|$ interactions. Another option would be to use one thread per query segment, but it runs the risk of not fully utilizing all available hardware threads since $|Q_{batch}|$, unlike $|D|$, is not large. More specifically, the kernel takes as input: (i) Q_{batch}, an array of query trajectory segments sorted by t^{start} values; (ii) *firstCandidate*, the index in D of the first candidate entry segment (recall that the entire database D is stored on the GPU once and for all); (iii) *numCandidates*, the number of candidate entry segments; (iv) d, the threshold distance; and (v) *setID*, a global index that keeps track of the location in memory where the next result set item should be written. Q_{batch}, *firstCandidate*, *numCandidates* are computed on the host before executing the kernel and transferred to the GPU along with d and *setID*. The kernel returns a set of time intervals annotated by trajectory ids.

The pseudo-code of the kernel is shown in Algorithm 2. The threads in OpenCL are numbered using a global id (*gid*≥ 0). As we use only *numCandidates* threads, all threads with *gid* larger than *numCandidates* do not participate in the computation (lines 2-5). Once the result set is initialized to the empty set (line 6), the relevant candidate segment is copied into the thread's private memory (variable *entrySegment*) line 7. The algorithm then loops over all query segments to compute interactions between the candidate segment and the query segments (line 8). Given the candidate segment and the current query segment, function *temporalIntersection()* generates new candidate and query segments that span the same time interval (line 9). The algorithm then computes the interval of time during which these two segments are within a distance d of each other (line 10), which involves computing the coefficients of and solving a degree two polynomial [20] (see Appendix B). If this interval is non-empty, then *setID* is incremented atomically (line 12). The interval is annotated with the trajectory id and added to the result set (line 13). The full result set is returned once all interactions have been computed.

Algorithm 2 Pseudo-code for the GPUTRAJDISTSEARCH kernel algorithm.

```
1:  procedure GPUTRAJDISTSEARCH (Q_batch, firstCandidate, numCandidates, d, setID)
2:      gid ← getGlobalId()
3:      if gid≥numCandidates then
4:          return
5:      end if
6:      resultSet ← ∅
7:      entrySegment ← D[firstCandidate + gid]
8:      for all querySegment ∈ Q_batch do
9:          (entrySegment, querySegment) ← temporalIntersection(
                entrySegment,querySegment)
10:         timeInterval ← calcTimeInterval(
                entrySegment,querySegment,d)
11:         if timeInterval ≠ ∅ then
12:             resultID ← atomic_inc(setID)
13:             resultSet[resultID] ← resultsSet[resultID] ∪ timeInterval
14:         end if
15:     end for
16:     return resultSet[0:setID]
17: end procedure
```

The size of the result set for a kernel invocation could be as high as the number of interactions, $|Q_{batch}| \times$*numCandidates*. However, in practice, only a small fraction of the interactions are added to the result set. Since memory for the result set must be allocated statically, in our experiments we conservatively allocate enough memory for a result set with as many items as there are entries in the dataset. In practice, one could allocate much less memory, and in the rare cases in which more memory is needed one would simply re-attempt the kernel execution with more allocated memory.

5.5 Generation of Query Batches

As explained in Section 5.3, an important question is that of choosing appropriate, perhaps optimal, query batch sizes. Using small batches increases the total number of kernel invocations, and each such invocation has a non-negligible overhead. Conversely, using large batches increases the number of wasteful interactions. This increase was demonstrated in Figure 5.4 as an example. Figure 5.5 shows the actual number of interactions per query segment vs. the number of queries per batch for a total of 40,000 query segments over the GALAXY dataset with 10^6

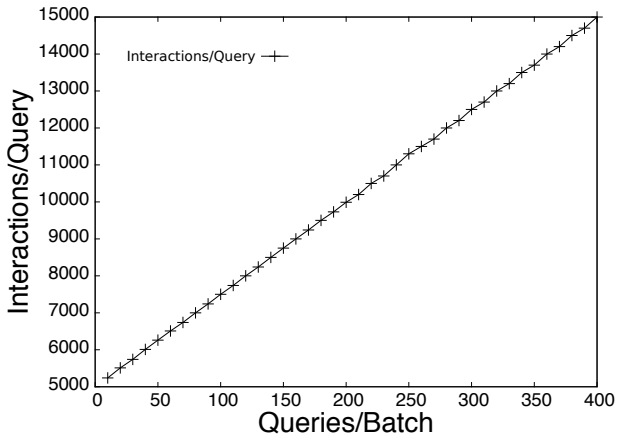


Figure 5.5. The number of interactions per query vs. batch size, for the GALAXY dataset (10^6 entry trajectory segments), with 40,000 query trajectory segments.

entry segments (see Section 5.6.1 for details on the datasets and queries used for experimental evaluations). As expected, the number of computed interactions, and thus the number of wasteful interaction computations, grows almost perfectly linearly with the batch size.

Beyond the above trade-off between between high overhead and high numbers of wasteful interactions, the temporal properties of the dataset should guide how one groups the query segments into batches. For instance, consider the example shown in Figure 5.4. The first and second sets of 10 query segments both overlap with the same set of entry segments (entry bins B_0 and B_1). Therefore, it is likely a good idea to group the first 20 query segments in a batch, since no extra wasteful interactions will be generated by this grouping (a total of 180 interactions). Consider now grouping together the third set of 10 query segments (which requires $10\times$(6+3+3)=120 interactions) and the fourth set of 10 query segments (which requires $10\times$(3+3)=60 interactions). This grouping leads to $20\times$(6+3+3+2)=280 interactions, for 280-120-60=100 extra wasteful interactions. As seen in this example, while picking a good batch size is important, it is also important to group together query segments that together do not overlap too many entry bins. In light of these considerations, in what follows we propose several algorithms to group query segments into batches.

5.5.1 Periodic

A simple approach to define query batches is to pick a single batch size, s, as in Figure 5.4. Each consecutive subsets of s queries in Q are then grouped together in a batch, for a total of $b = |Q|/s$ batches and thus b kernel invocations. We call this approach PERIODIC.

5.5.2 SetSplit

We propose a class of $O(|Q|^2)$ algorithms, called SETSPLIT, that attempt to group query segments together in a way that reduces wasteful interactions while yielding batches that are not too small.

The first algorithm, SETSPLIT-FIXED (Algorithm 3), produces a specified number of batches. More specifically, SETSPLIT-FIXED takes as input a set of query segments, Q, and the number of batches to generate, $numBatches$, and outputs a set of batches. The first step is to create a list of batches, B, in which each element is a single query segment (line 2). While the number of batches is larger than $numBatches$, the algorithm iteratively merges two adjacent batches into a single batch (loop at line 3). For each possible such merge (loop at line 5), we compute the sum of the numbers of interactions of two adjacent batches (line 6) and the number of interactions of the merge of these two batches (line 7). We determine the potential merge operation that would lead to the smallest increase in number of interactions (line 8), keeping track of the index of the first batch in that merge, $bestMerge$. We then replace batch $bestMerge$ by a batch obtained by merging batch $bestMerge$ and batch $bestMerge + 1$, and remove batch $bestMerge + 1$ (lines 13 and 14). The algorithm returns an array built from list B.

A drawback of SETSPLIT-FIXED is that it can produce many small batches, and potentially many batches that contain a single query segment, and thus lead to high overhead. Using a lower $numBatches$ value leads to more merge operations and thus lower overhead. However, it is unclear how to pick the best value for this parameter since it depends on the temporal properties of the datasets. To address these shortcomings, we propose another algorithm, SETSPLIT-MINMAX, that generates batches while imposing constraints on minimum and maximum batch sizes.

The pseudo-code of SETSPLIT-MINMAX is shown in Algorithm 4. SETSPLIT-MINMAX takes as input a set of query segments, Q, a lower bound on the batch size, min, and an upper bound on the batch size, max. It outputs a set of batches. The first phase of the algorithm (lines 2-21) is

Algorithm 3 Pseudo-code for the SETSPLIT-FIXED algorithm.

```
 1: procedure SETSPLIT-FIXED (Q, numBatches)
 2:     B ← list(Q)
 3:     while |B| > numBatches do
 4:         minDiff ← +∞
 5:         for i = 0, ..., |B| − 2 do
 6:             numIntsUnmerged ← numInts(B[i]) + numInts(B[i + 1])
 7:             numIntsMerged ← numInts(merge(B[i],B[i + 1]))
 8:             if numIntsMerged - numIntsUnmerged < minDiff then
 9:                 minDiff ← numIntsMerged - numIntsUnmerged
10:                 bestMerge ← i
11:             end if
12:         end for
13:         B[bestMerge] ← merge(B[bestMerge],B[bestMerge+1])
14:         B.removeElementAt(bestMerge+1)
15:     end while
16:     return array(B)
17: end procedure
```

similar to Algorithm 3 but for the fact that merges that would lead to a batch with more than max query segments are ignored (line 6). The second phase of the algorithm (lines 22-40) loops until no batch remains that contains fewer than min query segments. For each such batch, the algorithm considers the merge with the predecessor batch if any (lines 23-27), and with the successor batch if any (lines 28-32). The algorithm then performs the merge that leads to the smallest increase in number of interactions (lines 33-39). The algorithm returns an array built from list B.

We also consider an algorithm, SETSPLIT-MAX, that is a special case of SETSPLIT-MINMAX with $min = 1$, i.e., with no constraint imposed on the minimum batch size.

5.5.3 GreedySplit

In this section, we present a class of $O(|Q|)$ algorithms, called GREEDYSETSPLIT. Like SETSPLIT, GREEDYSETSPLIT also attempts to avoid small batches and to reduce wasteful interactions, but with lower complexity. The main idea behind GREEDYSETSPLIT is to first do all the "free" merges, i.e., those merges that do not increase the number of interactions, and then to merge contiguous batches using a single pass through the set of batches. The GREEDYSETSPLIT-MIN algorithm imposes a lower bound on the minimum batch size, while the GREEDYSETSPLIT-MAX algorithm imposes an upper bound on the maximum batch size. We consider a single constraint (ei-

Algorithm 4 Pseudo-code for the SETSPLIT-MINMAX algorithm.

```
1:  procedure SETSPLIT-MINMAX (Q, min, max)
2:      B ← list(Q)
3:      while true do
4:          minDiff ← +∞
5:          for i = 0, ..., |B| − 2 do
6:              if numSegments(merge(B[i],B[i + 1])) > max then
7:                  continue
8:              end if
9:              numIntsUnmerged ← numInts(B[i]) + numInts(B[i + 1])
10:             numIntsMerged ← numInts(merge(B[i],B[i + 1]))
11:             if numIntsMerged - numIntsUnmerged < minDiff then
12:                 minDiff ← numIntsMerged - numIntsUnmerged
13:                 bestMerge ← i
14:             end if
15:         end for
16:         if minDiff = +∞ then
17:             break
18:         end if
19:         B[bestMerge] ← merge(B[bestMerge],B[bestMerge+1])
20:         B.removeElementAt(bestMerge+1)
21:     end while
22:     while there exists B[i] such that numSegments(b) < min do
23:         if i > 0 then
24:             numIntsLeft = numInts(merge(B[i − 1],B[i]))
25:         else
26:             numIntsLeft = ∞
27:         end if
28:         if i < |B| − 1 then
29:             numIntsRight = numInts(merge(B[i],B[i + 1]))
30:         else
31:             numIntsRight = ∞
32:         end if
33:         if numIntsLeft < numIntsRight then
34:             B[i] ← merge(B[i − 1],B[i])
35:             B.removeElementAt(i − 1)
36:         else
37:             B[i] ← merge(B[i],B[i + 1])
38:             B.removeElementAt(i + 1)
39:         end if
40:     end while
41:     return array(B)
42: end procedure
```

ther minimum or maximum batch sizes), as designing a GREEDYSETSPLIT algorithm that would impose both constraints and terminates is difficult (a batch that is too large may need to be broken into batches that may then be too small).

GREEDYSETSPLIT-MIN takes as input a set of query segments, Q, and a lower bound on the batch size, $bound$, and it outputs a set of batches. Its pseudo-code is shown in Algorithm 5. In the first phase of the algorithm (lines 3-11), the algorithm traverses the set of batches, B, and merges two adjacent batches if this merge does lead to an increase in number of interactions. In a second phase (lines 12-20), the algorithm iteratively merges a batch with its successor if the batch contains fewer than min query segments (line 14). In the GREEDYSETSPLIT-MAX, line 14 is replaced by a "*numSegments(*$B[i]$*)>bound*" test and the if and else clauses are swapped. The algorithm returns an array built from list B.

Algorithm 5 Pseudo-code for the GREEDYSETSPLIT-MIN algorithm.

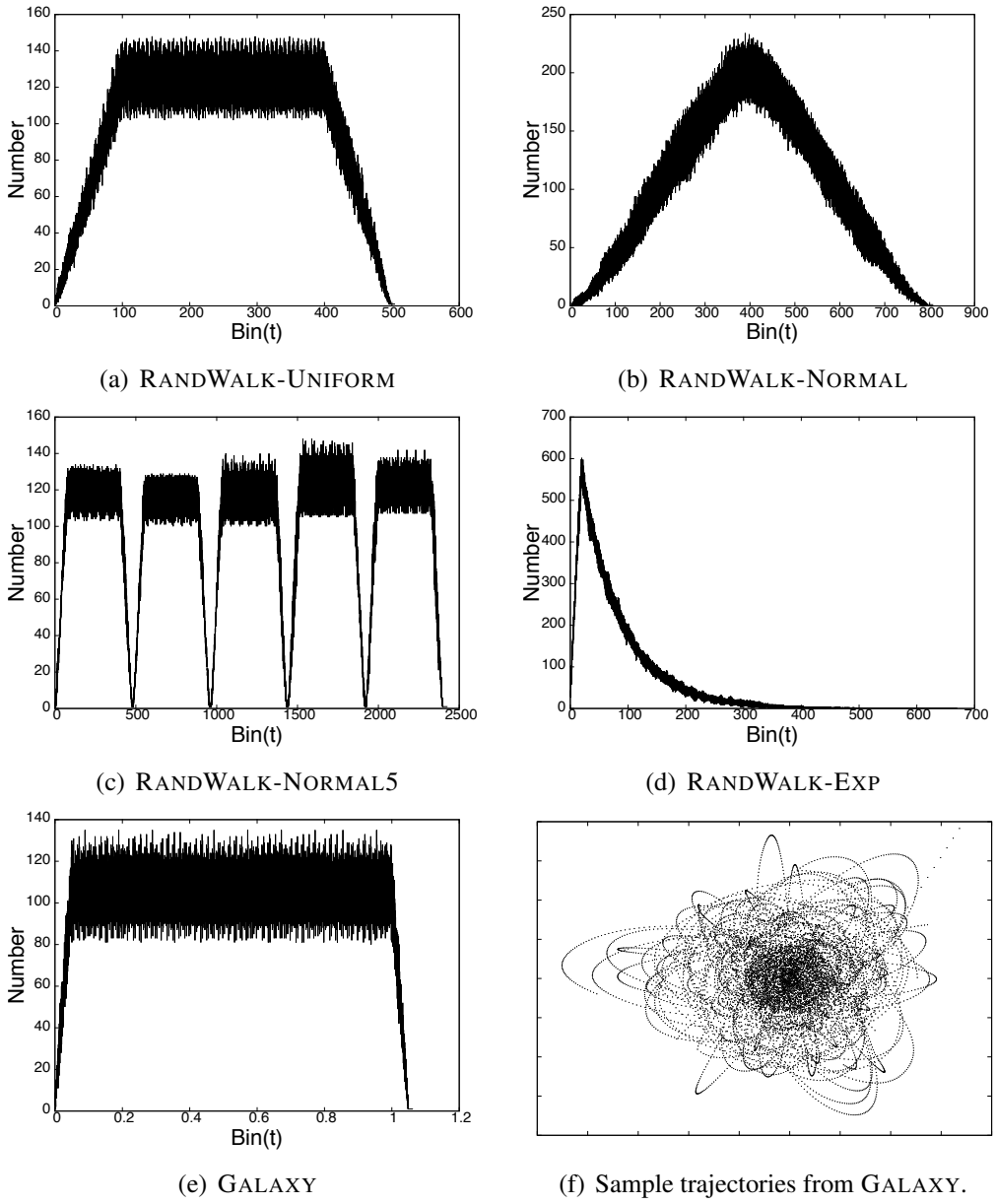
```
 1: procedure GREEDYSETSPLIT (Q, bound)
 2:     B ← list(Q)
 3:     i ← 0
 4:     while i < |B| − 1 do
 5:         if numInts(merge(B[i],B[i + 1])) = numInts(B[i]) + numInts(B[i + 1]) then
 6:             B[i] ← merge(B[i],B[i + 1])
 7:             B.removeElementAt(i + 1)
 8:         else
 9:             i ← i + 1
10:         end if
11:     end while
12:     i ← 0
13:     while i < |B| − 1 do
14:         if numSegments(B[i]) < bound then
15:             B[i] ← merge(B[i],B[i + 1])
16:             B.removeElementAt(i + 1)
17:         else
18:             i ← i + 1
19:         end if
20:     end while
21:     return array(B)
22: end procedure
```

5.6 Experimental Evaluation

5.6.1 Datasets

We evaluate our query processing scheme using several datasets, all of which are 4-dimensional (3 spatial dimensions, 1 temporal dimension). Our first dataset, called GALAXY, contains trajectories of stars moving in the Milky Way's gravitational field (as generated by the astronomy application described in Chapter 3). More specifically, this dataset contains 10^6 trajectory segments, corresponding to 2,500 trajectories of 400 timesteps each. Since each trajectory has the same number of timesteps and about the same temporal extent, the temporal profile of active trajectories is roughly uniform. However, since our approach relies on temporal data partitioning, we also generate synthetic datasets with various temporal profiles of the number of active trajectories. Such profiles occur, for instance, in datasets of vehicular traffic trajectories with nighttime, daytime, and rush hour patterns.

Our random datasets contain trajectories with random movements, similar to Brownian motion. The RANDWALK-UNIFORM dataset consists of 400-timestep trajectories whose start times are sampled from a uniform distribution over the [0,100] interval. The RANDWALK-NORMAL dataset is similar but uses a normal distribution to generate start times, with a mean of 200 and standard deviation of 200, truncated to the [0,400] interval. The RANDWALK-EXP dataset consists of trajectories with numbers of timesteps that are sampled from an exponential distribution with lambda=1/70, truncated to the [2,1000] interval, with start times sampled from a uniform distribution over a the [0,20] timestep interval. The RANDWALK-NORMAL5 dataset is generated but one of 5 different normal distributions is randomly selected when generating trajectories. This dataset thus exhibits distinct active and inactive phases, as occurs in datasets such as the vehicular traffic example above. The various parameter values for generating these datasets were picked so as to produce distinct patterns of numbers of entry segments assigned to entry bins. These patterns are shown in Figure 5.6 (a)-(e) for each dataset. In addition, Figure 5.6 (f) shows a sample of trajectories for the GALAXY dataset. Table 5.1 lists the number of trajectories and of entry segments in the datasets.



(a) RANDWALK-UNIFORM

(b) RANDWALK-NORMAL

(c) RANDWALK-NORMAL5

(d) RANDWALK-EXP

(e) GALAXY

(f) Sample trajectories from GALAXY.

Figure 5.6. Temporal distributions of active entry trajectory line segments in the datasets are shown in panels (a) through (e). The time corresponding to the midpoint of the bin is plotted on the horizontal axis, and the number of segments in the bin is shown on the vertical axis. A sample of the trajectories in the GALAXY dataset is shown (f).

Table 5.1. Characteristics of Datasets

Dataset	Trajectories	Entries
RANDWALK-UNIFORM	2,500	997,500
RANDWALK-NORMAL	2,500	1,000,000
RANDWALK-NORMAL5	2,500	1,000,000
RANDWALK-EXP	10,000	684,329
GALAXY	2,500	1,000,000

5.6.2 Experimental Methodology

The GPU-side implementation is developed in OpenCL, and the host-side implementation is developed in C++. The host-side implementation is executed on one of the 6 cores of a dedicated 3.46 Ghz Intel Xeon W3690 processor with 12 MB L3 cache, while the GPU side runs on an Nvidia Tesla C2075 card. We measure query response times averaged over 3 trials (standard deviation over the trials is negligible). In all experiments the number of entry bins in our index is set to 10,000. The implementations have been validated to ensure correctness. To guarantee that we do not obtain false positive or negative results, we compare the results of our implementation to an alternate implementation that utilizes a brute force approach.

In our experiments, we utilize the following trajectory searches:

- S1: From the GALAXY dataset, 100 trajectories are processed with $d = 1$, and with a total of 40,000 query line segments.
- S2: From the GALAXY dataset, 100 trajectories are processed with $d = 5$, and with a total of 40,000 query line segments.
- S3: From the RANDWALK-UNIFORM dataset, 100 trajectories are processed with $d = 5$, and with a total of 39,900 query line segments.
- S4: From the RANDWALK-UNIFORM dataset, 100 trajectories are processed with $d = 25$, and with a total of 39,900 query line segments.
- S5: From the RANDWALK-NORMAL dataset, 100 trajectories are processed with $d = 50$, and with a total of 40,000 query line segments.
- S6: From the RANDWALK-NORMAL dataset, 100 trajectories are processed with $d = 150$, and with a total of 40,000 query line segments.

- S7: From the RANDWALK-NORMAL5 dataset, 100 trajectories are processed with $d = 50$, and with a total of 40,000 query line segments.
- S8: From the RANDWALK-NORMAL5 dataset, 100 trajectories are processed with $d = 150$, and with a total of 40,000 query line segments.
- S9: From the RANDWALK-EXP dataset, 1000 trajectories are processed with $d = 50$, and with a total of 52,044 query line segments.
- S10: From the RANDWALK-EXP dataset, 1000 trajectories are processed with $d = 100$, and with a total of 69,881 query line segments.

For a given entry set the response time depends on the query set. This is because the spatiotemporal features of the queries determine the number of interactions to compute. However, we find that in all of our results, regardless of the query set, all of our candidate algorithms lead to response times with a relatively narrow range. For instance, for the GALAXY dataset and 10 different sample query sets, and for a query distance $d = 5$, the relative response time difference between the fastest and the slowest algorithm is only 1.99% on average and at most 3.08%. While the ranking of the particular algorithms may differ from one query set to another, these variations do not translate to large response time differences. Consequently, we only present results for a single query set.

5.6.3 Sequential Implementation and Multi-core OpenMP

While this work focuses on distance threshold searches on the GPU, Section 4.6 shows the results for our CPU implementation. To reiterate, the CPU implementation uses an R-tree index to store trajectory segments inside MBBs. One interesting question is how to "split" a trajectory, i.e., deciding on which (contiguous) segments should be stored in the same MBBs. In Section 4.6 we propose a trajectory splitting strategy that achieves a trade-off between the number of entries in the index, the volume of the space occupied by the MBBs, and the computational cost of candidate trajectory segment processing. Figure 5.7 shows average query response time vs. the number of segments indexed per MBB, for the GALAXY dataset for query distances $d = 1, \ldots, 5$, when executed on the host described in Section 5.6.2. In this case, indexing 12 segments per MBB yields the lowest average response time. The sequential CPU implementation can be easily parallelized

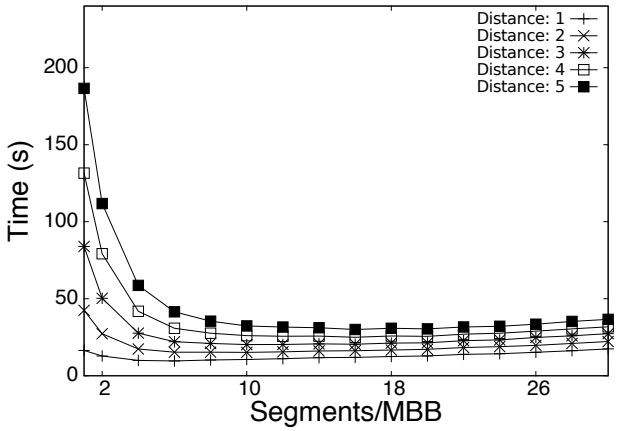


Figure 5.7. Response time vs. segments per MBB (r) for the GALAXY dataset with the same query set outlined in S1, but with $d = 1, 2, 3, 4, 5$.

using OpenMP. Figure 5.8 shows the response time vs. the number of threads for the GALAXY dataset, with 12 trajectory segments per MBB. On our 6-core host parallel efficiency is high (78%-90%), with parallel speedup between 4.69 and 5.44 with 6 threads. In what follows, we draw some comparisons between the performance of this CPU-only implementation and the performance of our GPU implementation. Note that these results are slightly different than those in Section 4.6 because we ran the experiments on a different platform.

5.6.4 Performance Evaluation

Let us first compare the performance of GPUTRAJDISTSEARCH to that of the sequential and parallel CPU implementations described in the previous section. We find that the relative performance of the GPU and CPU implementation is consistent across experimental scenarios. Let us consider experimental scenario S2 and only the PERIODIC algorithm for creating batches (using a batch size of 120). Our GPU implementation achieves average response time as low as 2.08 s, while for the same experimental scenario our sequential CPU implementation (using the best number of query segments per MBB for that scenario) leads to an average response time of 31.62 s. Our GPU implementation thus gives a speedup of 15.2 over the sequential CPU implementation. When compared to the OpenMP parallel CPU implementation, the average response time is 6.88 s, so that our GPU implementation achieves a speedup of 3.3. While these results are tied to the hardware characteristics of our experimental platform, we conclude that a GPU implementation of

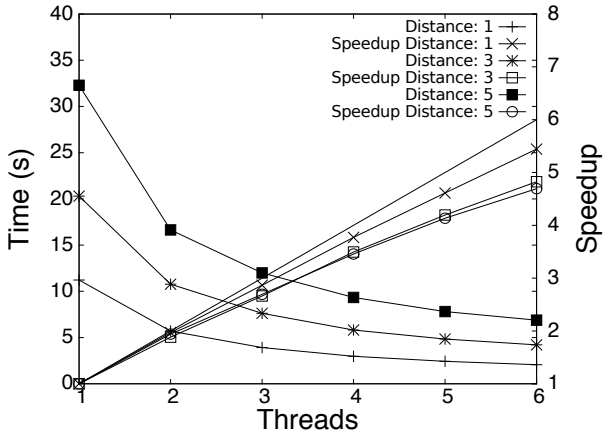


Figure 5.8. Response time vs. number of threads for the GALAXY dataset with the same query set outlined in S1 with $d = 1, 3, 5$ and $r = 12$. Results obtained using six cores on the platform.

distance threshold query processing is worthwhile and can yield substantial improvement over a CPU-only version.

We now evaluate the relative merit of the algorithms for creating query segment batches (PERIODIC, SETSPLIT, GREEDYSETSPLIT). The results are similar across experimental scenarios; therefore, note that most of the following figures appear in the appendix. Response time results for experimental scenarios S1 to S10 are shown as follows: Figure 5.9 (for S1 and S2), Figure A.1 (for S3 and S4), Figure A.2 (for S5 and S6), Figure A.3 (for S7 and S8), and Figure A.4 (for S9 and S10). For each experimental scenario, the response time of the PERIODIC algorithm is plotted versus the batch size on the left-hand side of the figure. A zoomed-in version of each plot is shown on the right-hand side, which shows the neighborhood of the best batch size for PERIODIC, as well as the response times of the SETSPLIT and GREEDYSETSPLIT algorithms (which are shown as horizontal lines). These results correspond to a "best case" for the SETSPLIT and GREEDYSETSPLIT algorithms, for two reasons. First, the response time results do not include the time necessary to compute the query batches. This time is negligible for PERIODIC, but can be significant for SETSPLIT and even for GREEDYSETSPLIT, as discussed at the end of this section. Second, using an exhaustive search, for each experimental scenario we have determined the best parameter configuration for the SETSPLIT and GREEDYSETSPLIT algorithms (i.e., the best number of batches for SETSPLIT-FIXED, the best maximum batch size for SETSPLIT-MAX, the best minimum and maximum batch size for SETSPLIT-MINMAX, the best minimum batch size for

Table 5.2. Percentage response time difference relative to the lowest response time for all algorithms and experimental scenarios. Results for the algorithm with the lowest response time shown in boldface.

Algorithm	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
GREEDYSETSPLIT-MAX	0.15	0.15	0.15	**0.00**	**0.00**	0.60	1.44	0.34	1.29	0.16
GREEDYSETSPLIT-MIN	**0.00**	0.24	**0.00**	0.15	0.52	0.11	**0.00**	**0.00**	0.93	0.10
SETSPLIT-FIXED	1.11	1.69	1.03	1.02	0.92	1.03	2.34	1.52	562.90	0.62
SETSPLIT-MAX	1.50	1.97	1.02	0.35	1.51	1.54	3.37	2.78	0.90	0.69
SETSPLIT-MINMAX	0.24	0.33	0.10	0.17	0.69	**0.00**	0.77	0.93	**0.00**	**0.00**
PERIODICBEST	0.37	**0.00**	0.23	0.50	0.83	1.03	1.11	0.09	1.50	1.69
PERIODICGOOD	3.21	2.47	1.64	3.56	2.15	1.43	2.21	1.05	2.52	2.69

GREEDYSETSPLIT-MIN, and the best maximum batch size for GREEDYSETSPLIT-MAX). The results in Figures 5.9-A.4 are summarized in Table 5.2, which shows for each algorithm, and each experimental scenario, the percentage response time difference relative to the response time of the best algorithm for that experimental scenario. We show two versions of the PERIODIC algorithm. PERIODICBEST corresponds to PERIODIC when using the batch size that leads to the lowest response time for the experimental scenario at hand. PERIODICGOOD corresponds to PERIODIC but using the worst batch size in a -20/+20 neighborhood of the best batch size (i.e., the batch size in that interval that leads to the highest response time).

Some trends are clearly seen in the results. Over the 10 experimental scenarios, the SETSPLIT and GREEDYSETSPLIT algorithms all lead to response times that are close to each other (within 3.4%). One exception is for SETSPLIT-FIXED, which leads to a significantly larger response time for S9 (about a factor 10 larger than the other algorithms). Recall that SETSPLIT-FIXED creates a fixed number of batches without any constraint on the maximum batch size. S9 contains entries with exponentially distributed temporal extents, which causes SETSPLIT-FIXED to create a few very large batches at the tail of the distribution (which leads to the smallest *minDiff* value - line 8 in Algorithm 3). These large batches are the reason for the high response time of SETSPLIT-FIXED. This problem does not occur for experimental scenario S10 due to the larger total number of query segments. An interesting finding is that the GREEDYSETSPLIT algorithms, even though they use a single pass through the query segments, do well. GREEDYSETSPLIT-MAX, resp. GREEDYSETSPLIT-MIN, leads to the lowest response time in 2, resp. 4, of the 10 experimental scenarios. Overall, the GREEDYSETSPLIT algorithms are among the 3 best algorithms for each

experimental scenario. This suggests that the quadratic complexity of the SETSPLIT algorithm to attempt a less local optimization is in fact unnecessary.

The key observation from our results is that PERIODIC leads to good performance. As seen in Figure 5.9, the response time of PERIODIC can be high for some batch sizes. However, when using the best batch size, PERIODIC can produce response time on par or even better than that of the GREEDYSETSPLIT and SETSPLIT algorithms. Overall, for each experimental scenario PERIODICBEST leads to response times at most 1.69% larger than that of the best GREEDYSETSPLIT or SETSPLIT algorithm for that scenario. It even leads to the lowest response time for experimental scenario S2. Even when PERIODIC does not use the best batch size it leads to good results. PERIODICGOOD still leads to response times at most 3.56% larger than the best GREEDYSETSPLIT or SETSPLIT algorithm over the 10 experimental scenarios.

As explained above, our results do not include the time to compute the batches. Due to quadratic complexity, for the SETSPLIT algorithms this time is large, factors larger than the query response time for our experimental scenarios. Overall, when adding the time to compute the batches (on the CPU), we find that the SETSPLIT algorithms lead to average response time more than 4.69 times larger than PERIODICBEST and up to 8.84 times larger (discounting SETSPLIT-FIXED for experimental scenario S9, which leads to response time 12.76 times larger). The GREEDYSETSPLIT algorithms fare better when compared to PERIODICBEST, with response times only up to 2.9% larger over all experimental scenarios. This is because these algorithms have linear complexity.

We conclude that although computing batches that reduce wasteful interactions, as in the SETSPLIT and GREEDYSETSPLIT algorithms, is an appealing idea, in practice it does not outperform a simple periodic approach. This is because the small response time benefit due to the use of better batches is offset by the CPU time overhead of computing these batches. One drawback of PERIODIC is that one must specify a good batch size, i.e., a batch size in a neighborhood of the best batch size. In the next section, we propose performance modeling techniques that can be used to determine such a good batch size.

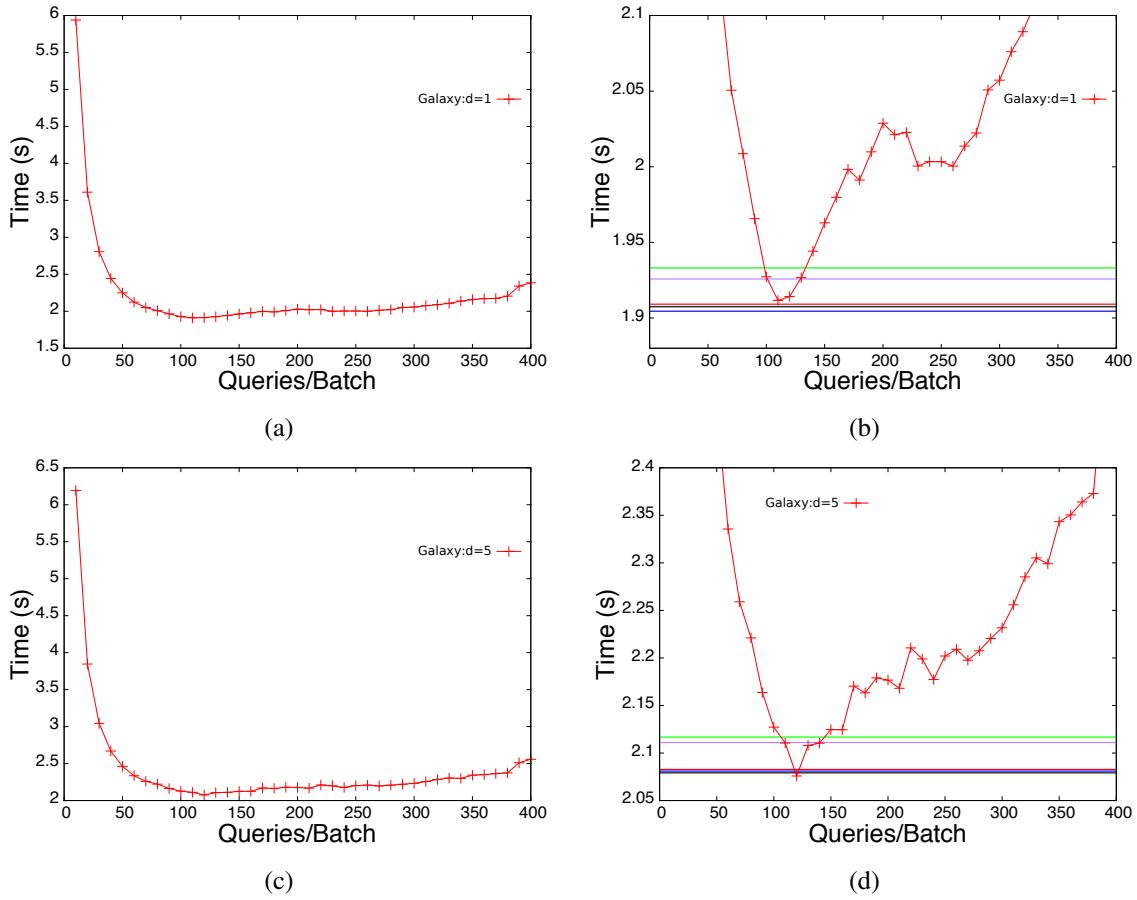


Figure 5.9. Response time vs. queries/batch (s) for the periodic query batch method for S1 (a) and S2 (c) (GALAXY dataset). Panels (b) and (d) correspond to zoomed in versions of (a) and (c) respectively, to highlight the minimum response times. The colored lines correspond to the best response time from the query splitting algorithms, where SETSPLIT-FIXED is purple, SETSPLIT-MAX is green, SETSPLIT-MINMAX is red, GREEDYSETSPLIT-MIN is blue, and GREEDYSETSPLIT-MAX is black.

5.7 Performance Modeling

Previous work on spatiotemporal database querying, and in particular works that consider distance threshold searches [5], and the work in Chapter 4 rely on index-trees, such as R-trees. These index-trees have complex performance behavior as the traversal time depends on the set of pointers followed on a path toward a leaf node, which is highly data dependent. As a result, predicting query response time is challenging. An added difficulty in the case of distance threshold searches is that one query may lead to a large result set while another may lead to an empty result set.

Because designed for GPU execution, the indexing scheme proposed in this work (Section 5.3 and 5.4) does not rely on index-trees. While not completely free of data-dependent behavior, the more deterministic behavior of this scheme makes it possible to predict query response time. And in particular, such prediction is sufficiently accurate to determine a good batch size for the PERIODIC algorithm.

The model consists of a GPU component and CPU component. The GPU component accounts for the invocation overhead and execution time of each individual kernel invocation, so that summing over all invocations gives the estimated GPU time for processing the entire set of query segments. The CPU component accounts for the time to perform memory allocations, set kernel parameters, send query data to the GPU, receive result sets from the GPU, marshal data, and perform other CPU-side computations (e.g., counter and pointer updates). Figure 5.10 shows response time results for the S1 experimental scenario, showing both the CPU and GPU components. The GPU curve shows an initial decrease as the batch size, s, increases in the interval $10 \leq s \leq 40$. This decrease is because for low s values the GPU device is underutilized and the kernel invocation overhead is large due to many such invocations. For $s \geq 50$, the GPU time increases due to the increasing number of interactions that must be computed (as explained in Section 5.5). The CPU time curve shows a steady decrease as s increases. This is because the smaller the value of s the more kernel invocations and thus the more work done on the CPU. We show two portions of the CPU time. The time necessary to perform kernel invocations, including the transfer of query segments from the CPU to the GPU, is shown as a shaded cyan portion below the CPU curve. The shaded blue portion corresponds to the time necessary for transferring result sets from the GPU back to the CPU.

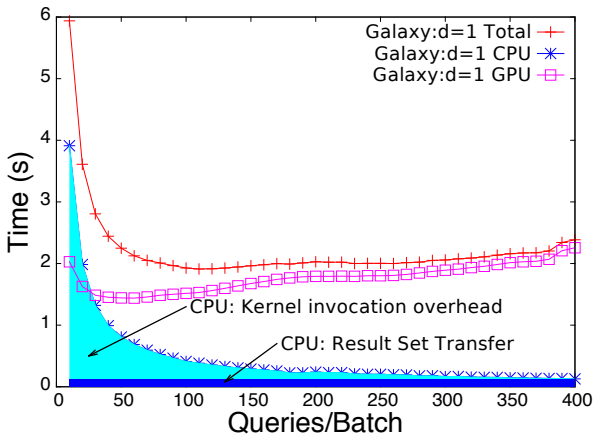


Figure 5.10. Response time vs. queries/batch (s) for S1 (GALAXY dataset with $d = 1$). The individual CPU and GPU components are shown.

5.7.1 GPU Component

Model

In this section, we derive an empirical model for the GPU component of our performance model. Let us use $T^{GPU}(i, c)$ to denote the GPU time for a kernel invocation that computes the i interactions necessary for comparing a batch of i/c query segments against c candidate segments (using c GPU threads). Let us use $\Theta^{GPU}(i, c)$ to denote the overhead of launching a no-op kernel for q query segments and i interactions (the overhead depends both on the number of queries and on the number of GPU threads). Given the i interactions to be computed, we denote by α the fraction of these interactions that lead to an item being added to the result set (i.e., both a temporal hit and a spatial hit), by β the fraction of these interactions for which the entry segment does not overlap the query segment temporally (i.e., a temporal miss), and by γ the fraction of these interactions for which the entry segment overlaps the query segment temporally but not spatially (i.e., a temporal hit but a spatial miss). We have $\alpha + \beta + \gamma = 1$. We distinguish these three cases because the computational cost is different in each. Candidate segments that are temporal misses can be determined with only a few instructions (i.e., comparing temporal extremities of query and candidate segments). Candidate segments that are temporal hits but spatial misses require more instructions (i.e., spatial extremities comparisons). Candidate segments that should be added to the result set require even more instructions to be performed (i.e., determining the actual overlapping

temporal interval). One can view the computation of an interaction as a set of comparisons and moving distance calculations, but these comparisons and computations are short-circuited whenever a segment is found to be a temporal or spatial miss.

We denote by $T_1(i, c)$, $T_2(i, c)$, and $T_3(i, c)$ the time for a kernel invocation with i interactions so that all c candidate segments are temporal and spatial hits, temporal misses, and temporal hits and spatial misses, respectively. This leads us to the following model:

$$T^{GPU}(i, c) = T_1^{GPU}(\alpha i, c) + T_2^{GPU}(\beta i, c) + T_3^{GPU}(\gamma i, c) - 2\Theta^{GPU}(i, c).$$

The first three terms above each include a $\Theta^{GPU}(i, c)$ component, hence the subtracted fourth term. $T^{GPU}(i, c)$ is computed for each batch, and the sum gives the total GPU time assuming the batch size is s:

$$T^{GPU}(s) = \sum_{j=0}^{|Q|/s} T^{GPU}(i_j, i_j/s).$$

Q is the total set of query segments (for simplicity this equation assumes that s divides $|Q|$). i_j is the number of interactions that must be computed for the j-th query batch, which is determined based on the entry segment bins (see Section 5.3). Therefore i_j/s is the number of candidate entry segments for the s query segments in the batch.

In this model, parameters α, β and γ depend on the dataset and the query. They must thus be determined empirically for typical scenarios. By contrast, the functions Θ^{GPU}, T_1^{GPU}, T_2^{GPU}, T_3^{GPU} depend only the hardware characteristics of the platform. In what follows we describe how we estimate these parameters. Note that this estimation is done for each batch of s queries.

Estimating α, β and γ

Recall that α is, for a kernel invocation on a query batch, the fraction of interactions that lead to a new item being added to the result set. Given that it is dataset dependent, we use a pragmatic approach to estimate α for a particular dataset once and for all, i.e., before the dataset is being queried in "production" use. Depending on the temporal distribution of the entry segments, there may be time periods with few active trajectories and some with many, resulting in a non-uniform distribution of query hits throughout time. To estimate α, we divide the dataset into $numEpochs$ temporal epochs. For each epoch we select a batch of s sample queries that fall within

the epoch. We do this by randomly selecting s consecutive query segments from a representative query dataset. We then execute our kernel and calculate the fraction of interactions that produced result items. We perform this over enough trials such that the predicted total number of result set items is within 5% of the total true number of result set items. This procedure yields an α estimate for each epoch. This estimate may be inaccurate if the sample queries are not representative of queries that will be processed in production. Also, if too low a value of $numEpochs$ is used, then the α estimates are more likely to be inaccurate since transient temporal patterns are then averaged over larger epochs. Using $numEpochs = 1$ is a degenerate case in which our model would assume that for any kernel invocation the query hit probability is the same. This may be accurate for a temporally uniform dataset, but vastly inaccurate for datasets that exhibit temporal transience. In all the experiments presented hereafter we use $numEpochs = 50$.

Unlike α, β can be computed precisely. For a given set of s query segments, one can determine which entry segments they may temporally overlap using the bins in our indexing scheme (see Section 5.3). Then, with two nested loops one can simply compare the temporal extremities of each query segment to that of each entry segment, yielding an exact value for β. Parameter γ is computed as $1 - \alpha - \beta$.

To summarize, for a given dataset we compute once and for all a set of α estimates for each epoch and for the full range of (reasonable) s values. Then, for each batch of s queries we compute an α estimate, β and γ. Therefore, for each candidate s value we can plug appropriate values of these three parameters into the T^{GPU} performance model.

Estimating T_1^{GPU}, T_2^{GPU}, T_3^{GPU} and Θ^{GPU}

The T_1^{GPU}, T_2^{GPU}, T_3^{GPU} and Θ^{GPU} functions depend only on the implementation of the kernel and the hardware characteristics of the platform. As a result, we can empirically estimate these time components based on benchmark results. Let us consider T_1^{GPU}, i.e., the kernel response time when all interactions are both temporal hits and spatial hits. The same approach is used to estimate $T_2^{GPU}(i, c)$, $T_3^{GPU}(i, c)$ and $\Theta^{GPU}(i, c)$.

We generate a synthetic dataset and query set in which all interactions are guaranteed to be both temporal and spatial hits. Figure 5.11 shows a subset of our benchmark results as response time vs. number of interactions for various numbers of candidate entry segments, as measured on our target platform. Given a number of interactions, i, and a number of candidate entries, c,

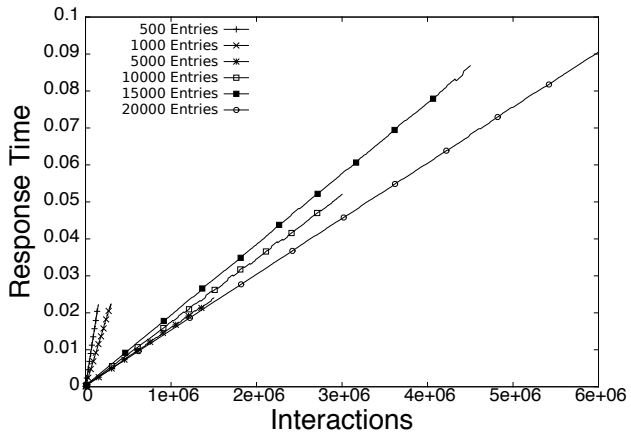
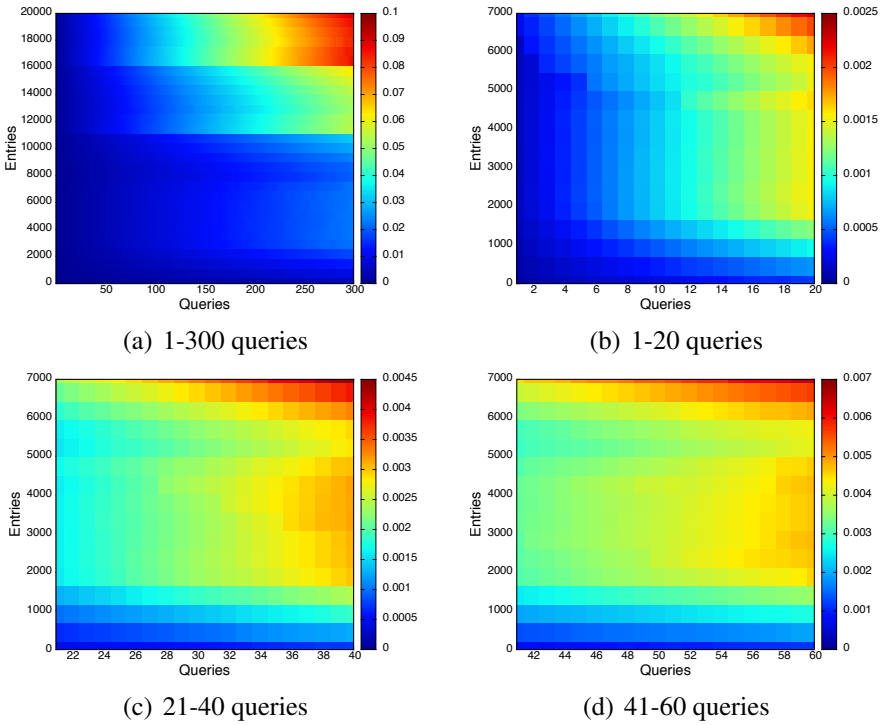


Figure 5.11. Interactions vs. response time for a selection of entries. The data shown corresponds to a range of 1-300 benchmarked queries.

we simply use linear interpolation to determine a response time prediction $T_1^{GPU}(i, c)$ from the benchmark results.

Figure 5.12 (a) shows a broader range of benchmark results, shown as heat maps of the response time vs. the number of candidate entries, c and the number of queries $q = i/c$. In Figure 5.12 (a) we observe that there are discontinuities in response time. We attribute these discontinuities mainly to thread scheduling factors on the GPU. Regardless, they will be a source of modeling error due to our use of linear interpolation. Figure 5.12 (b), (c), and (d) show plots with queries in the range of 1 to 60. We see somewhat smoother response time trends, particularly in Figure 5.12 (c) and (d) suggesting that in that range the use of linear interpolation should lead to less error. Since we know that the batch size, and thus the number of candidate segments, should be relatively small, then one may expect that modeling error due to linear interpolation could also be small.

The above modeling approach assumes that the kernel response time can be estimated from the benchmark-based models of T_1^{GPU}, T_2^{GPU}, T_3^{GPU}, executed separately. However, an actual kernel execution consists of a mix of temporal and spatial hits, temporal misses, and temporal hits but spatial misses. One may thus wonder whether the notion of separating the model into three components can lead to reasonable response time predictions. To answer this question we compared executions of the three benchmark kernels to a mixed execution, using various synthetic datasets and query sets with $\alpha = \beta = \gamma = 1/3$.



(a) 1-300 queries

(b) 1-20 queries

(c) 21-40 queries

(d) 41-60 queries

Figure 5.12. Benchmark of interactions that are all within the query distance. Panel (a) shows a large selection of the data, across a range of 300 queries, and (b), (c) and (d) show detailed versions of the data between 1-60 queries.

Figure 5.13 shows the response time vs. test case, where the first two histograms correspond to the separate and mixed tests, respectively, and the rightmost histogram shows the ratio of mixed to separate response times. The test cases are identified as "Separate" or "Mixed", followed by the number of entry segments and the batch size. Since $\alpha = \beta = \gamma = 1/3$, then the number of queries of each type (α, β, γ) is equal to the total number of queries in an experimental scenario divided by 3. For example, consider the scenario where there are 9 queries in total, Separate-100E/3Q (3 queries of each type) is compared to Mixed-100E/9Q, where we compare three kernel executions with queries of type α, β, and γ each with a query batch size of 3, to one kernel execution with 9 queries, which are a mixture of query types α, β, and γ. To allow for fair comparisons we have discounted the kernel invocation overhead from all results (this overhead occurs three times in the Separate results but only once in the Mixed results).

The first observation is that the Mixed executions have lower response times than the Separate counterparts. This may seem counter-intuitive because the Mixed executions, unlike

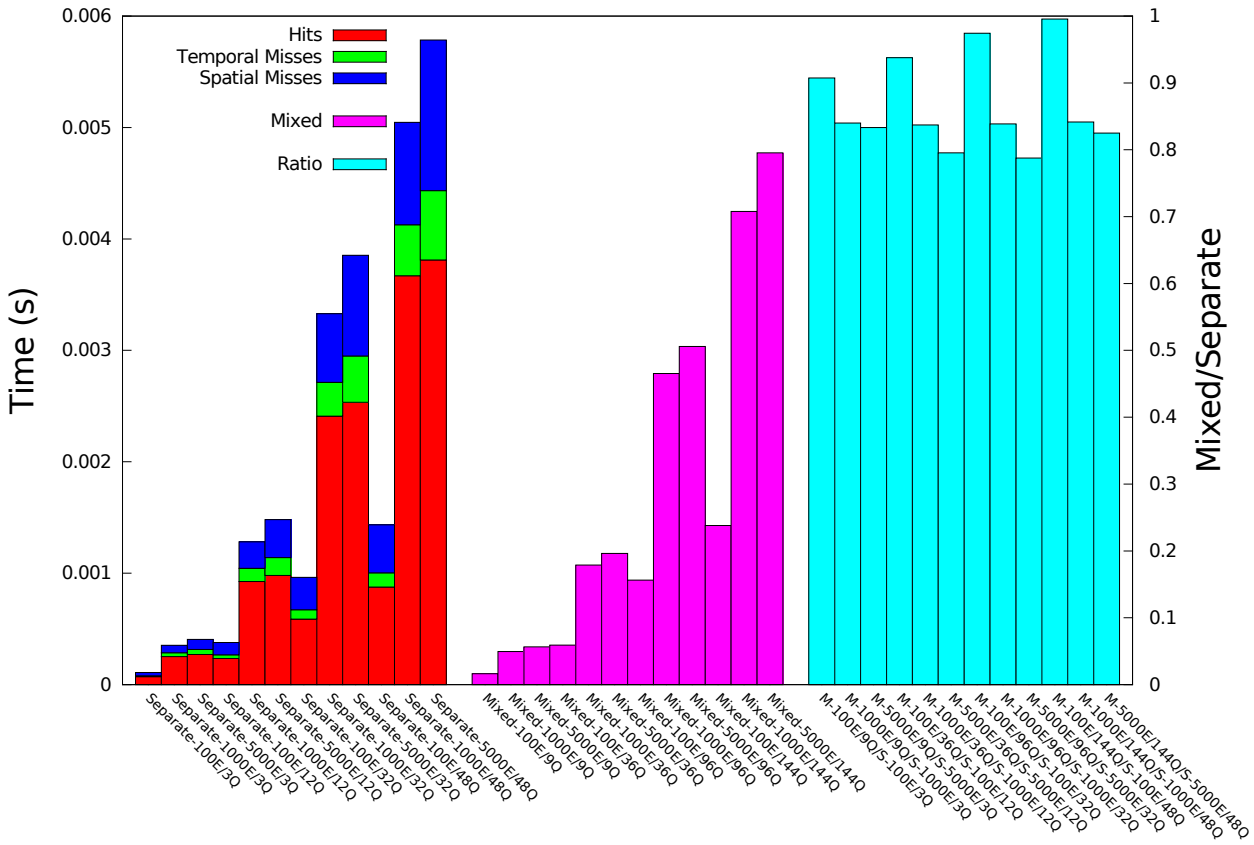


Figure 5.13. GPU response time vs. test cases of mixed and separated kernel invocations.

the Separate executions, should have a high degree of branch divergence, thus causing partially serialized thread executions on the GPU [23]. However, in Mixed executions the entry segments are retrieved from the GPU's global memory and stored into private memory once, and are then reused. This does not occur in the Separate executions as entry segments have to be reloaded from global memory into private memory at each kernel invocation. Regardless, the rightmost histogram in Figure 5.13 shows that the error due to using Separate executions is relatively consistent and in the 1%-20% range. As a result, we expect that our modeling approach should lead to reasonable response time predictions. Furthermore, since the error is consistent, the same bias should apply when comparing the estimated response time for various candidate batch sizes.

5.7.2 CPU Component

To model the CPU time, we propose an empirical model for each of the two portions of the CPU time shown in Figure 5.10 as shaded cyan and blue areas. These models consist of simple curve fitting based on benchmark results (R^2 values for the fits are above 0.9999).

To estimate $T_1^{CPU}(s)$, the portion of the CPU time that corresponds to the kernel invocation overhead for a batch size s (the cyan area in Figure 5.10), we generate a synthetic dataset with $\alpha \approx 0$. With a very low value of α, the result set has negligible size. As a result, kernel response time is approximately equal to the aggregate kernel invocation overhead. We thus obtain a kernel invocation overhead curve for the full range of possible batch sizes. This benchmark must be executed for various total numbers of query segments so that for a query set Q, our model uses benchmark results obtained for approximately $|Q|$ query segments. In practice, one would thus run the benchmark for various numbers of query segments, obtaining a family of CPU response time curves. In all experiments hereafter, $|Q|$=40,000, and we thus use 40,000 queries as well in our benchmark. We obtain the following response time fitted curve:

$$T_1^{CPU}(s) = -0.0017 + 32.2946 \times s^{-0.9528}. \tag{5.7.1}$$

To estimate T_2^{CPU}, the portion of the CPU time that corresponds to the transfer of the result set from the GPU to the CPU (the blue area in Figure 5.10), we rely on the α parameter defined in the GPU component of our performance model and estimated as described in Section 5.7.1. Using α, we can determine the number of result set items generated by each kernel invocation. Summing over all kernel invocations and multiplying by the size in bytes of a result set item yields the total size of the result set, which we denote by σ. Assuming that T_2^{CPU} does not depend on s, we can then estimate it by dividing σ by the GPU-CPU bandwidth measured on the platform. On our target platform the model is as follows:

$$T_2^{CPU}(\sigma) = 1.54 \times 10^{-8} \times \sigma. \tag{5.7.2}$$

In the end, the total CPU time is modeled as: $T^{CPU}(s, \sigma) = T_1^{CPU}(s) + T_2^{CPU}(\sigma)$, and the total response time is modeled as $T^{CPU}(s) + T^{GPU}(s, \sigma)$.

Table 5.3. Model Results.

Search	Model	Actual	Slowdown
S1	80	110	4.8%
S2	80	120	6.3%
S3	80	120	4.5%
S4	80	110	4.5%
S5	100	140	2.8%
S6	100	140	2.3%
S7	140	160	0.8%
S8	150	160	0.58%
S9	170	220	0.1%
S10	200	210	0.97%

5.7.3 Model Evaluation

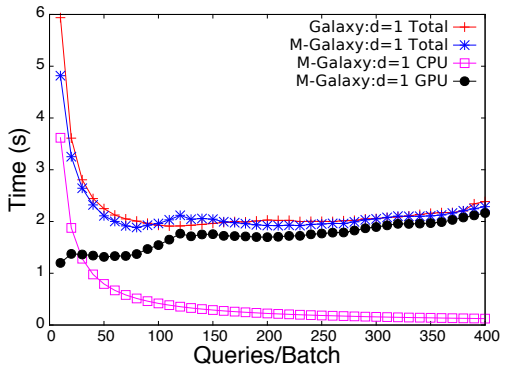
Figure 5.14 shows actual and modeled response times vs. the batch size for a selection of our experimental scenarios. The CPU and GPU model components are shown with separate curves. The general trends of the actual response time are respected by the model. In some instances, the model tracks the actual response time well, while on other it exhibits some deviations. However, the main purpose of our model is not to predict response time perfectly, but to produce a sufficiently coherent prediction so that a good batch size can be selected. Figure 5.14 (b) presents the model for the GALAXY dataset with $d = 5$. The model suggests that $s = 80$ yields the best response time; however, the actual best response time occurs when $s = 120$. Had $s = 80$ been chosen, $s = 120$ would be 6.3% faster. Such results are summarized in Table 5.3 for our 10 experimental scenarios, where the Model column gives the batch size based on the model, the Actual column gives the empirically best batch size, and the Slowdown column gives the response time slowdown due to using the model-driven batch size, as a percentage. Note that S3 has few elements in its result set, and thus has a low value of α across all epochs. For this scenario, it was not possible to assure that the total estimated number of result set items is within 5% of the actual number across all values of s.

In these results we find that the worst slowdown is less than 7% and that in many cases the slowdown is negligible. We conclude that, in spite of data dependency challenges, our model is useful for determining a good batch size to use with the PERIODIC algorithm and for predicting

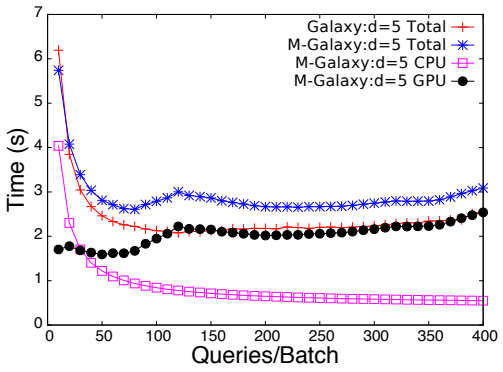
the total query response time. An interesting question is whether our modeling approach can be used for response time prediction purposes for other spatiotemporal queries.

5.8 Conclusions

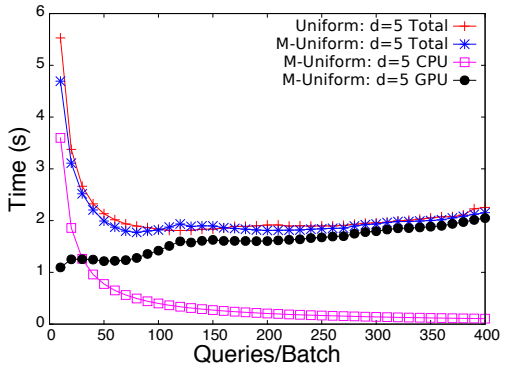
In this chapter, we have advanced and index and algorithms for GPU executions of the distance threshold search. We have considered the case where there are memory constraints on the GPU. We proposed a GPU-friendly indexing scheme used in the context of incrementally processing the query set by executing a series of kernel invocations. Although we developed algorithms to make good query batches under a range of experimental scenarios, in practice we find that a fixed batch size is sufficient. We developed an empirical query response time model that can predict response time to a reasonable degree, such that it can be used to predict the best batch size to be used on a given dataset. We find that the GPU affords a significant speedup over the multithreaded CPU implementation. It is expected that the bandwidth bottleneck between the host and GPU will decrease in upcoming years, which will further boost the advantage of our GPU implementations.



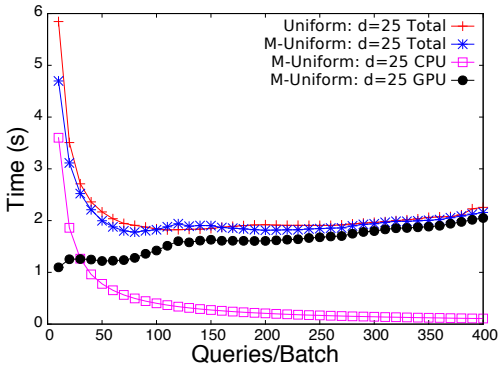
(a) S1: GALAXY.



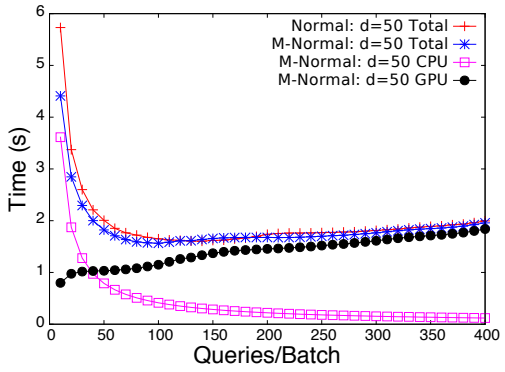
(b) S2: GALAXY.



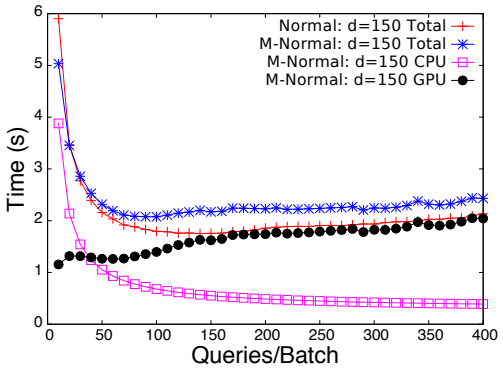
(c) S3: RANDWALK-UNIFORM.



(d) S4: RANDWALK-UNIFORM.

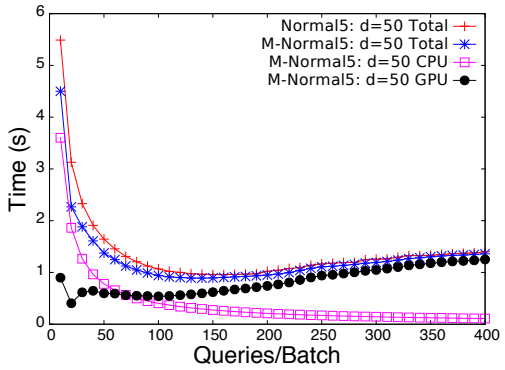


(e) S5: RANDWALK-NORMAL.

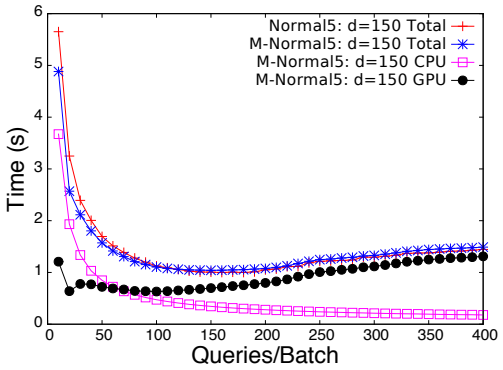


(f) S6: RANDWALK-NORMAL.

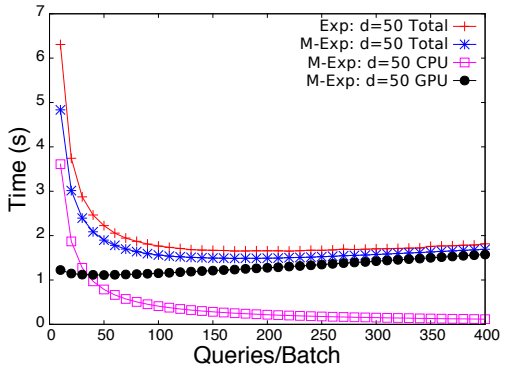
Figure 5.14. Modeled response times vs. queries per batch (s) for searches on each dataset. The red curve shows the actual response time, the blue curve shows the modeled total response time, where the CPU (magenta) and GPU (black) model components added together equal the modeled total (blue) curve.



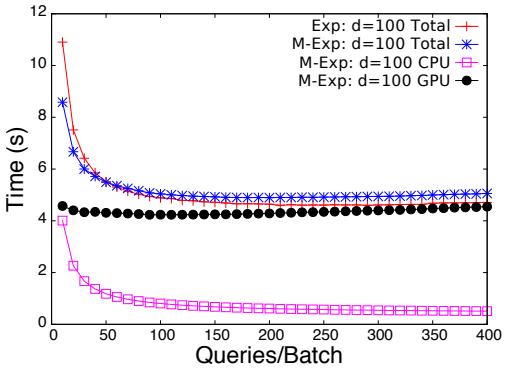
(g) S7: RANDWALK-NORMAL5.



(h) S8: RANDWALK-NORMAL5.



(i) S9: RANDWALK-EXP.



(j) S10: RANDWALK-EXP.

Figure 5.14. continued.

Chapter 6

GPU Indexing Schemes and Algorithms for Non-Memory-Constrained GPGPU Distance Threshold Searches

The previous chapter focuses on GPU executions of the distance threshold search with memory constraints. In contrast, in this chapter, we assume a scenario where those memory constraints are removed, and the query set can fit on the GPU with the database. Our intended scenario is that of a distributed memory environment in which a number of GPU-equipped compute nodes are reserved by a user, thus there is sufficient memory for the entire query set. With memory constraints removed, the opportunity arises to investigate different trajectory indexing schemes. We propose three trajectory indexing schemes for the distance threshold search on the GPU that have spatial, temporal, and spatiotemporal selectivity. For each of these indexes, we develop GPU kernels that minimize branch instructions to achieve good parallel efficiency. We compare our implementations to the R-tree CPU implementation in Chapter 4 and we find that the GPU yields a significant speedup.

6.1 Problem Statement

6.1.1 Problem Definition

The problem definition in this chapter is the same as that in Section 5.2. However, we consider the case in which both D and Q can fit in GPU memory. This may be the case in a distributed memory environment where there is sufficient memory across a series of GPU-equipped compute nodes and the GPUs are reserved by a single user.

6.1.2 Memory Management on the GPU

Chapters 4 and 5 focus on indexing trajectories in two environments. CPU implementations (as that in Chapter 4) rely on in-memory index trees that have been used traditionally for out-of-core implementations, such as the R-tree [21]. Each thread traverses the tree and creates a candidate segment set to be further processed to create the final result set. Although many candidate segments are not within distance d of the query segments due to the "wasted space" in the index (an unavoidable consequence of using MBBs shown in Chapter 4), memory must still be allocated to store these candidate segments. Furthermore, the size of the final result set is non-deterministic as it depends on the spatiotemporal nature of the data. Consequently, memory allocation for the result set must be conservative and overestimate the memory required (this overestimation grows linearly with $|Q|$). On the CPU these memory management issues are typically not problematic in practice since the number of threads is limited (e.g., set to the number of physical cores) and the memory is large.

As discussed in Chapter 5, there are memory limitations when using the GPU (which are not germane to the CPU). In contrast to the scenario in Chapter 5, where both D and Q cannot simultaneously fit in memory, in this chapter, we assume that both D and Q fit in memory. However, to address the issue of non-deterministic storage requirements for the result set, on the GPU one must define a fixed size for a statically allocated memory buffer. If the memory requirements exceed this buffer then it is necessary to perform a series of kernel invocations so as to incrementally "batch" the generation of the final result set. Note that in most instances, only a single kernel invocation is required. Thus, in practice, only a small number of batches are required to generate

the final result set. In contrast, in Chapter 5, there are a significantly greater number of batches that are required to process an entire result set.

6.2 Indexing Trajectory Data

In this section we outline three trajectory indexing techniques for the GPU. For each we discuss shortcomings and possible solutions regarding the memory management issues discussed in Section 6.1.2.

6.2.1 Spatial Indexing: Flatly Structured Grids

Previous work has proposed the use of grid files, or "flatly structured grids" (FSG), to index trajectory data on the GPU spatially [71]. In that work the authors focus on 2-D spatial data (and Hausdorff distance) while our context is 3-D spatiotemporal data (and Euclidean distance). An interesting question is whether spatial indexing with FSGs is effective even when the data has a temporal dimension. In what follows we describe an FSG indexing scheme and accompanying search algorithm for the GPU. We call this approach GPUSPATIAL.

Trajectory Indexing

We define a FSG as a 3-D rectangular box partitioned into cells with $grid_x$, $grid_y$, $grid_z$ cells in the x, y, and z spatial dimensions, respectively, for a total of $grid_x \times grid_y \times grid_z$ cells. Each line segment l_i in D is contained in a spatial MBB defined by two points MBB_i^{min} and MBB_i^{max} where,

$$MBB_i^{min} = (\min(x_i^{start}, x_i^{end}), \min(y_i^{start}, y_i^{end}) \min(z_i^{start}, z_i^{end})),$$

and

$$MBB_i^{max} = (\max(x_i^{start}, x_i^{end}), \max(y_i^{start}, y_i^{end}), \max(z_i^{start}, z_i^{end})).$$

Each line segment is assigned to the FSG by rasterizing its MBB to grid cells. Figure 6.1 shows a 2-D example for two line segments and a 5×4 FSG. Each line segment may occupy more than one grid cell, and some grid cells can remain empty. We store the FSG as an array of non-empty

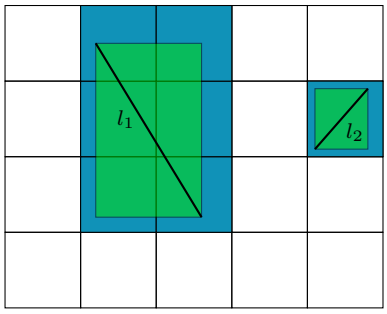


Figure 6.1. 2-D example rasterization of two line segment MBBs (green) to grid cells (blue) in a 4×5 FSG. l_1: a long line segment whose MBB spans six grid cells; l_2: a short line segments whose MBB spans one grid cell.

cells, G. Each cell is denoted as C_h, $h = 1, \ldots, |G|$, where h is a linearized coordinate computed from the cell's x, y, and z coordinates using row-major order.

Each cell C_h is defined by h, and by an index range $[A_h^{min}, A_h^{max}]$ in an additional integer "lookup" array, A. $A[A_h^{min} : A_h^{max}]$ contains the indices of the line segments whose MBBs overlap cell C_h (the notation $X[a : b]$ is used to denote the "slice" of array X from index a to index b, inclusive). In other terms, if l_i's MBB overlaps C_h, then $i \in A[A_h^{min} : A_h^{max}]$. Since the MBB of line segment l_i can overlap multiple grid cells, i can occur multiple times in array A. Figure 6.2 shows an example to highlight the relationship between G, A, and D. This example is discussed later on.

One of the objectives of the above design is to reduce the memory footprint of the index. This is why we only index non-empty grid cells, and why for each cell C_h we do not store its spatial coordinates but instead compute h whenever needed (thereby trading off memory space for computation time). Furthermore, the use of lookup array A makes it possible for array G to consist of same-size elements (even though some cells contain more line segments than others). Without this extra indirection through lookup array A, it would have been necessary to store entry segment ids directly into the elements of G. However, it would have been necessary to pick an element size large enough to accommodate the cell with the largest number of entry segments, thereby wasting memory space. D, A, and G are stored in GPU memory before query processing begins.

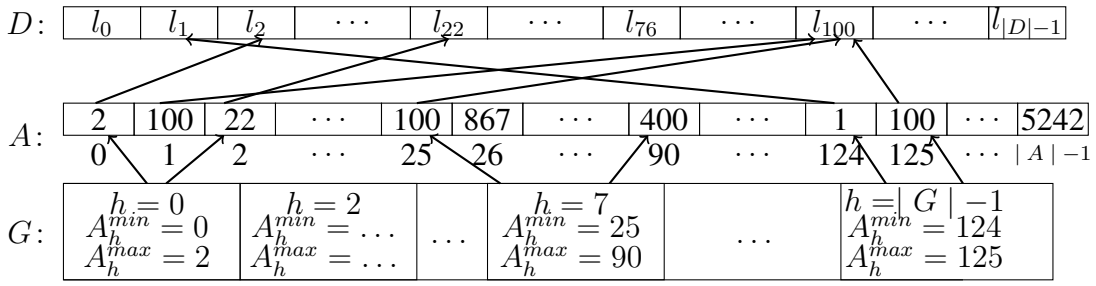


Figure 6.2. Example relationship between the grid (G), the lookup array (A) and the database of entry line segments (D) in the GPUSPATIAL approach.

Search Algorithm

The trajectory segments in Q are not sorted by any spatial or temporal dimension. This is because sorting segments temporally would not be effective when using a spatial index. Regarding spatial sorting, it is not clear by which dimension the segments should be sorted. However, segments that are part of the same query trajectory are stored contiguously, thus providing a natural advantageous ordering of data elements. Each query segment q_k is assigned to a GPU thread. The kernel first calculates the MBB for q_k and the FSG cells that overlap this MBB. Given the x, y, z coordinates of each such cell in the FSG, the kernel computes its linearized coordinate (h) using a row-major order. A binary search is used to find whether cell C_h occurs in array G, in $O(\log |Q|)$ time. In this manner the kernel creates a list of non-empty cells that overlap q_k's MBB. For each cell C_h in this list, the indices of the entry segments it contains are computed as $A[A_h^{min} : A_h^{max}]$. These indices are appended to a buffer U_k. A key point here is that with a spatial indexing scheme there is no good approach for storing index entry segments in a contiguous manner (since one would have to arbitrarily pick one of the spatial dimensions). This is why we must resort to using buffer U_k as opposed to, for instance, a 2-integer index range in a contiguous array of entry segments. Each entry in U_k is then compared to the query segment q_k to see if it is within the threshold distance; however, note that while the segments are expected to be relatively nearby each other spatially (given their FSG overlap), they may not overlap temporally.

Consider the example in Figure 6.2, which shows partial contents of arrays G, A, and D. Consider a query q_1 (not shown in the figure), which overlaps grid cells C_0, C_1, and C_7. Cell C_1 is not in G, meaning that it contains no entry segments. Therefore, the only two cells to consider are C_0 and C_7, which have [A^h_{min},A^h_{max}] values of [0,2] and [25,90], respectively. In lookup array A,

we find that [0,2] corresponds to entries 2, 100, and 22, while [25,90] corresponds to entry indices 100, 867, . . ., 400. These indices are copied from A into buffer U_k. Note that in this step the search algorithm does not remove duplicate indices (such as entry index 100 in this example) and thus may perform some redundant entry segment processing. Removing duplicates would amount to sorting buffer U_k, as done for instance in [71], which thus comes at an additional computational cost that may offset the benefits of removing redundant segment processing.

Since the number of entry segments that overlap q_k's MBB can be arbitrary large (it depends on the spatial features of D and Q, and on the query distance d), the use of buffer U_k creates memory pressure, especially since both D (along with G and A) and Q are stored on the GPU. This same issue has been encountered in previous work, e.g., when using a parallel R-tree index on the GPU [36]. We define an overall buffer size, s, that is split equally among all queries ($|U_k| = s/|Q|$). If the processing of query q_k exceeds the capacity of U_k, then the thread terminates, and stores the query id into an array that is sent back to the host. Once the kernel execution finishes, the host re-attempts the execution of those queries that could not complete due to memory pressure. In this re-attempt, memory pressure is lower because fewer queries are executed (i.e., $|U_k|$ is larger). This method implicitly has the effect that threads with similar (large) amounts of work to execute together, resulting in improved load-balancing.

The pseudo-code of the search algorithm is shown in Algorithm 6. It takes the following arguments: (i) the FSG array (G); (ii) the lookup array (A); (iii) the database (D); (iv) the set of queries (Q); (v) an array that contains the ids of the queries to be reprocessed (*queryIDs*), which is empty for the first kernel invocation; (vi) buffer space (U); (vii) the query distance (d); (viii) an output array in which the kernel stores the ids of the queries that must be reprocessed (*redo*); and (ix) the memory space to store the result set (*resultSet*). Arguments that lead to array transfers between the host and the GPU, either as input or output, are shown in boldface. Other arguments are either pointers to pre-allocated zones of (global) GPU memory or integers. The algorithm begins by checking the global thread id and aborts if it is greater than Q or |queryIDs|, depending on whether this is a first invocation or a re-invocation (lines 3-4). The id of the query assigned to the GPU thread is then acquired from Q or using an indirection via *queryIDs* (lines 6-8). Function *getCandidates* searches the FSG and returns a boolean that indicates whether buffer space was exceeded and the (possibly empty) set of candidate entry segment ids (line 10). If buffer space was exceeded, then the query id is atomically added to the *redo* array and the thread terminates

Algorithm 6 GPUSPATIAL kernel.

```
1:  procedure SEARCHSPATIAL (G, A, D, Q, queryIDs, U, d, redo, resultSet)
2:      gid ← getGlobalId()
3:      if queryIDs = ∅ and gid≥|Q| return
4:      if queryIDs ≠ ∅ and gid≥|queryIDs| return
5:      if queryIDs = ∅ then
6:          queryID ← gid
7:      else
8:          queryID ← queryIDs[gid]
9:      end if
10:     (overflow , candidateSet) ← getCandidates(G, A, D, Q[queryID], U, d)
11:     if overflow then
12:         atomic: redo ← redo ∪ { queryID }
13:         return
14:     end if
15:     for all entryID ∈ candidateSet do
16:         result ← compare(D[entryID],Q[queryID])
17:         if result ≠ ∅ then
18:             atomic: resultSet ← resultSet ∪ result
19:         end if
20:     end for
21:     return
22: end procedure
```

(line 11-13). The algorithm then loops over all candidate entry segment ids (line 15), compares each entry segment spatially and temporally to the query (line 16) and atomically adds a query result, if any, to the result set (line 18). Once all GPU threads have completed, *resultSet* and *redo* are transferred back to the host. If |redo| is non-zero, then the kernel is re-invoked, passing *redo* as *queryIDs*. Duplicates in the result set are filtered out on the host.

6.2.2 Temporal Indexing

In this section we propose a purely temporal partitioning strategy, which we call GPUTEMPORAL.

Trajectory Indexing

We begin by sorting the entries in D by ascending t_{start} values, re-numbering the entry segments in this order, i.e., $t_i^{start} \leq t_{i+1}^{start}$. The full temporal extent of D is $[t_{min}, t_{max}]$ where $t_{min} = \min_{l_i \in D} t_i^{start}$ and $t_{max} = \max_{l_i \in D} t_i^{end}$. We divide this full temporal extent so as to create m logical bins of fixed length $b = (t_{max} - t_{min})/m$. We assign each entry segment, l_i, $i = 1, \ldots, |D|$, to a bin, where l_i belongs to bin B_j, $j = 1, \ldots, m$, if $\lfloor t_i^{start}/b \rfloor = j$. There can be temporal overlap between the line segments in adjacent bins. For each bin B_j we defined its start times as $B_j^{start} = j \times b$ and its end time as $B_j^{end} = \max((j+1) \times b, \max_{l_i \in B_j} t_i^{end})$. B_j^{start} does not depend on the line segments in bin B_j, but B_j^{end} does. The temporal extent of bin B_j is defined as $[B_j^{start}, B_j^{end}]$. Given the definitions of B_j^{start} and B_j^{end}, the union of the temporal extents of the bins is equal to the full temporal extent of D. We define $B_j^{first} = \arg\min_{i|l_i \in B_j} t_i^{start}$ and $B_j^{last} = \arg\max_{i|l_i \in B_j} t_i^{start}$, i.e., the ids of the first and last entry segments in bin B_j, respectively. $[B_j^{first}, B_j^{last}]$ forms the index range of the entry segments in B_j. Bin B_j is thus fully described as (B_j^{start},B_j^{end}, B_j^{first}, B_j^{last}). The set of bins forms the temporal database index.

Figure 6.3 shows an example of how line segments may be assigned to a set of temporal bins. In this example, 15 entry segments are assigned to 4 temporal bins over a database temporal extent of 12 time units (spatial dimensions are ignored, and thus line segments are simply represented as horizontal lines in the figures). The B^{start},B^{end}, B^{first} and B^{last} values are shown for each bin. For instance, three entry segments are assigned to bin B_2: l_9, l_{10}, and l_{11}. Thus, $B_2^{first} = 9$ and $B_2^{last} = 11$. $B_2^{start} = 2 \times (12/4) = 6$ and $B_2^{end} = t_{11}^{end}$.

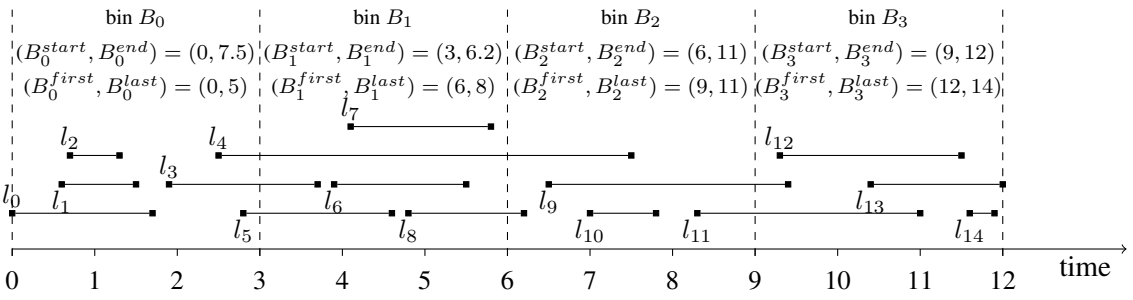


Figure 6.3. An example assignment of entry line segments to temporal bins in the GPUTEMPORAL approach.

Search Algorithm

Before performing the actual search, the following pre-processing steps must be performed. First, query segments in Q are sorted by non-decreasing t_{start} values, in $O(|Q| \log |Q|)$ time. For each query segment q_k, we calculate the index range of the contiguous bins that it overlaps temporally. A naïve algorithm for computing this overlap would be to scan all bins in $O(m)$ time. A binary search could be used to obtain a logarithmic time complexity. In practice, however, there are many temporally contiguous query segments and each overlaps only a few bins. Since segments in Q are sorted by non-decreasing t_{start}, the search can be done efficiently by using the first temporal bin that overlaps the previous query segment as the starting point for the scan for the temporal bins that overlap the next query segment. The search thus typically takes near-constant time. Let $\mathcal{B}_k$ denote the set of contiguous bins that temporally overlap query segment q_k, as identified by the above search. In constant time we can now compute the index range of the entry line segments that may overlap and must be compared with q_k: $E_k = [\min_{B \in \mathcal{B}_k} B_j^{first}, \max_{B \in \mathcal{B}_k} B_j^{last}]$. We term the mapping between q_k and E_k the *schedule*, S. Each GPU thread compares a single query to the line segments in D whose indices are in the E_k range. Assuming that $|Q|$ is moderately large, one is then insured that all GPU cores can be utilized.

In our implementation, all preprocessing described in the previous paragraph is performed on the CPU. Some of this preprocessing could be performed on the GPU (e.g., sorting the query segments). In an initial implementation, we performed the calculation of E_k on the GPU; however, this did not result in any performance improvement. As explained earlier, on the host the search for temporally overlapping bins can be drastically improved by relying on the

same search for the previous query segment. However, this cannot be implemented on the GPU as it would require thread synchronization and communication, which cannot be performed across thread blocks. In all of our experiments, the time to compute S on the CPU is a negligible portion of the overall query response time.

Algorithm 7 GPUTEMPORAL kernel.

```
 1: procedure SEARCHTEMPORAL (D, Q, S, d, resultSet)
 2:     gid ← getGlobalId()
 3:     if gid≥|Q| return
 4:     queryID ← gid
 5:     entryMin ← S[gid].EntryMin
 6:     entryMax ← S[gid].EntryMax
 7:     for all entryID ∈ {entryMin,...,entryMax} do
 8:         result ← compare(D[entryID],Q[queryID])
 9:         if result ≠ ∅ then
10:             atomic: resultSet ← resultSet ∪ result
11:         end if
12:     end for
13:     return
14: end procedure
```

The pseudo-code of the search algorithm is shown in Algorithm 7. It takes the following arguments: (i) the database (D); (ii) the query set (Q); (iii) the schedule (S); (iv) the query distance (d); and (v) the memory space to store the result set (*resultSet*). As in Algorithm 6, arguments that lead to array transfers between the host and the GPU are shown in boldface. The algorithm first checks the global thread id and aborts if it is greater than $|Q|$ (line 3). The query assigned to the thread is then acquired from Q (line 4). Next, the algorithm retrieves the minimum and maximum entry segment indices from the schedule (lines 5-6). From line 7 to 13 the algorithm then operates as Algorithm 6.

6.2.3 Spatiotemporal Indexing

In the two previous sections we have proposed a purely spatial and a purely temporal indexing scheme. The spatial scheme leads to segments in Q and D being compared that are spatially relevant but may be temporal misses (no temporal overlap). Likewise, the temporal indexing scheme compares temporally relevant segments in Q and D, but these segments may be spatial misses (no spatial overlap). Therefore, either approach can outperform the other depending on the

spatiotemporal characteristics of Q and D. Assuming for the sake of discussion that these characteristics do not give any such particular advantage to either one of the two indexing approaches, we can reason about their relative performance. First, the spatial indexing approach requires buffer space to store the spatially overlapping trajectory segments. In contrast, because the temporal indexing scheme is indexed in a single dimension, the temporally overlapping entry segments can be defined by an index range in D, which represents significant memory space savings. The same method could possibly be used with a spatial indexing scheme if considering only one of the spatial dimensions, making the index no longer a multi-dimensional grid, but instead a linear array. This approach would however drastically decrease the spatial selectivity of the search, leading to large increases in wasted computational effort (i.e., comparisons of segments that have no overlap in one or two of the spatial dimensions). Second, to minimize the memory footprint on the GPU, the spatial scheme requires two additional arrays (G and A), thus leading to two indirections in global GPU memory. In contrast, the temporal scheme requires a single indirection. Moreover, the entry segments are stored contiguously in the temporal scheme, while this is not the case in the spatial scheme.

Given the features of both the spatial and the temporal indexing scheme, we attempt to find an alternate spatiotemporal index that retains the benefit of both schemes without some of the drawbacks mentioned above. We term this approach GPUSPATIOTEMPORAL.

Trajectory Indexing

GPUSPATIOTEMPORAL adopts a temporal index so as to avoid the buffering and multiple indirection issues of spatial indexing, but subdivides each temporal bin into spatial subbins to achieve spatial selectivity. Entry segments in D are assigned to m temporal bins exactly as for GPUTEMPORAL. We then compute the spatial extent of D in each dimension. For instance, in the x dimension the extent of D is:

$$[x_{min}, x_{max}] = [\min_{l_i \in D}(\min(x^i_{start}, x^i_{end})), \max_{l_i \in D}(\max(x^i_{start}, x^i_{end}))] \ .$$

Spatial extents in the y and z dimensions are computed similarly. We then compute the maximum spatial extent in each dimension of the entry segments, which for the x dimensions is $\max_{l_i \in D} |x^i_{start} - x^i_{end}|$. Maximum spatial extents are computed similarly for the y and z dimen-

sion. For each of the temporal bins, we create v spatial subbins along each dimension, with the constraints that these subbins are larger than the maximum spatial extent of the entry segments. For instance, in the x dimension, this constraint is expressed as $v \leq (x_{max} - x_{min}) / \max_{l_i \in D} |x^i_{start} - x^i_{end}|$. We place this constraint for two reasons, which will be clarified when we describe the search algorithm: (i) to eliminate duplicates in the result set, and (ii) to reduce the amount of redundant information in the index. In total we have $m \times v$ subbins and we denote each subbin as $\hat{B}_{i,j}$, with $i = 1, \ldots, m$ and $j = 1, \ldots, v$.

The part of Figure 6.4 above the dashed line shows an example of how entry line segments are logically assigned to bins and subbins. The very top of the figure shows $m = 3$ temporal bins, B_0 to B_2. Each temporal bin contains the segments with ids in the range $[B_j^{first}, B_j^{last}]$. For instance, $B_1^{first} = 4$ and $B_1^{last} = 7$. Each entry segment is described by an id and 2 spatial (x, y, z) extremities. For instance, segment l_6 is in temporal bin B_1 and its spatial extremities are $(8, 9, 10)$ and $(10, 9, 8)$. Temporal dimensions are omitted in the figure. Below the temporal bins, we depict 9 temporal spatial subbins, $\hat{B}_{0,0}$ to $\hat{B}_{2,2}$ ($v = 3$ subbins per temporal bin). For each subbin, we indicate its spatial range in the x, y, and z dimension. Each subbin spans 4 spatial units in the x and y dimensions, and 5 spatial units in the z dimension. Given segment lengths in the database these subbin dimensions meet the constraints described in the previous paragraph. For each subbin and each dimension, we show the overlapping entry segment ids. For instance, consider subbin $\hat{B}_{0,1}$. It is overlapped in the x dimension by l_0, l_2 and l_3, in the y dimension by l_3, and in the z dimension by l_1 and l_3.

The part of Figure 6.4 below the dashed line shows how the logical assignment of segments to spatial subbins is implemented physically in memory. We create three integer arrays, X, Y, and Z, depicted at the bottom of the figure. Each array stores the ids of the line segments that overlap the subbins in one spatial dimension. The ids for a subbin are stored contiguously, for the subbins $\hat{B}_{i,j}$'s sorted by (j, i) lexicographical order. This is illustrated using colors in the figure and amounts to storing contiguously all ids in the first subbins of the temporal bins, then all ids in the second subbins of the temporal bins, etc. For instance, for the y dimension, the Y array in our example consists of $v = 3$ chunks. The first chunk corresponds to the ids in subbins $\hat{B}_{0,0}$ (l_0,l_1,l_3), $\hat{B}_{1,0}$ (l_4, l_5, l_8), and $\hat{B}_{2,0}$ (l_9), the second chunk corresponds to the ids in subbins $\hat{B}_{0,1}$ (l_2,l_3), $\hat{B}_{1,1}$ (l_5, l_7), and $\hat{B}_{2,1}$ (l_8), and the third chunk corresponds to the ids in subbins $\hat{B}_{0,2}$ (none), $\hat{B}_{1,2}$ (l_6), and $\hat{B}_{2,2}$ (l_8). The reason for storing the ids in this manner is as follows. Consider a query segment

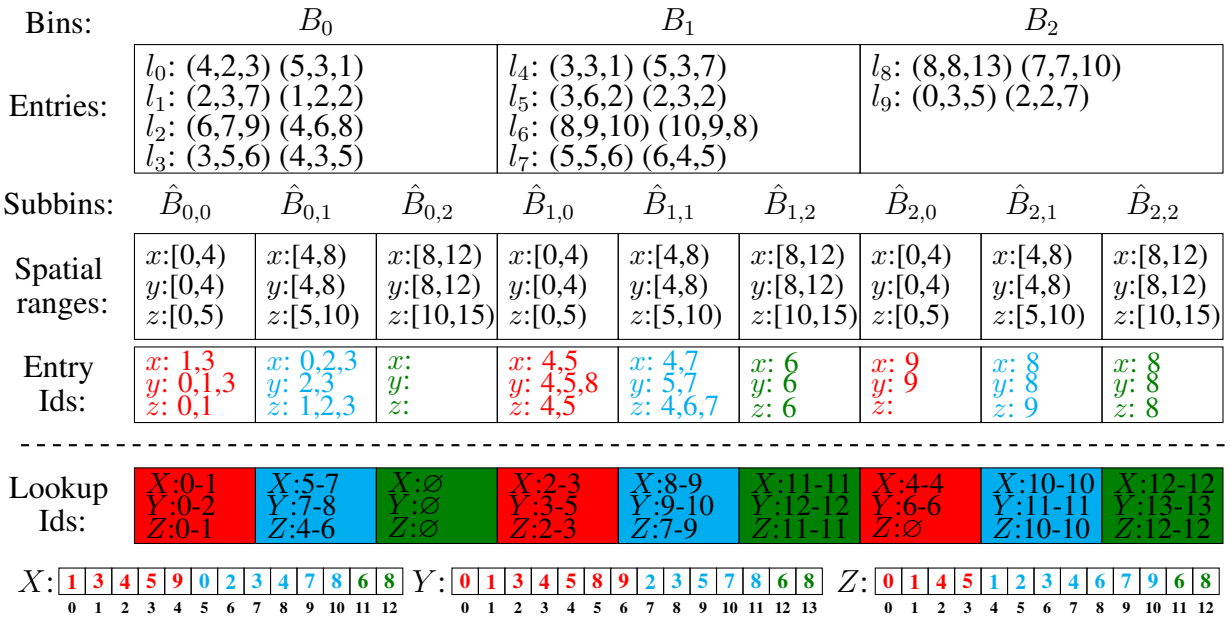


Figure 6.4. Example spatiotemporal indexing of a dataset with 10 entry segments. Above the dashed line is the logical assignment of the segments to the spatial subbin. Below the dashed line is the physical realization of this assignment in GPU memory.

with some spatial and temporal extent. This query may overlap several contiguous temporal bins (as shown in Section 6.2.2). However, because of the way in which we choose the sizes of the spatial subbins, most queries will not overlap multiple subbins in all three dimensions. Identifying potential overlapping entry segments then amounts to examining the i-th subbin of contiguous temporal bins, for some $0 \leq i \leq v$. In other words, based on the example in Figure 6.4, this amounts to examining sequences of same-color subbins.

Given the X, Y, and Z array, each spatial subbin is then described with the index range of the entries in those arrays, i.e., 6 integers. For instance, consider subbin $\hat{B}_{0,1}$ in our example. Its description is index range $5-7$ in the x dimension (i.e., it overlaps with segments $l_{X[5]}$ to $l_{X[7]}$ in the x dimension), index range $7-8$ in the y dimension (i.e., it overlaps with segments $l_{Y[7]}$ and $l_{Y[8]}$ in the Y dimension), and index range $4-6$ in the z dimension (i.e., it overlaps with segment $l_{Z[4]}$, $l_{Z[5]}$ and $l_{Z[6]}$ in the z dimension). Using this indirection, each spatial subbin is of fixed size. When compared to the purely temporal index, this spatiotemporal indexing scheme requires only additional space in GPU memory for the X, Y, and Z integer arrays, which corresponds to $\gtrsim 3|D| \times 4$ bytes.

Search Algorithm

On the host, as in the GPUTEMPORAL approach, we first sort Q and for each query segment calculate the temporally overlapping entries from the temporal bins. We also compute the set of spatially overlapping subbins in each dimension. This computation also takes place on the host, where the description of the bins and subbins are stored. Arrays X, Y, and Z are stored on the GPU. One option would be to compute the intersection of entry segments that belong to these subbins so as to select only spatially relevant entry segments. This turns out to be inefficient because we would then have to send a list of entry segment indices to the GPU, which has high overhead. Instead, we seek a solution in which we send a fixed and small number of indices to the GPU. As a result, we opt for a poorer but easier to encode selection of the candidate entry segments. Among the three spatial dimensions we pick the one in which the number of entry segments that overlap the query segment is the smallest. We then simply send an index range, 2 integers, in the X, Y, or Z array, depending on the dimension that was picked. This approach may lead to wasteful computation on the GPU (i.e., evaluation of entry segments that do not overlap with the query segment in one of the other two spatial dimensions), but the overhead of these wasteful computations is offset by the gain from the reduced amount of data that is sent to the GPU. Let us demonstrate how this approach exploits the way in which the X, Y, and Z arrays are constructed in the previous section. For the example in Figure 6.4, consider a query segment that overlaps temporal bins 0 and 1, and overlaps spatially with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the x dimension (entries 0,2,3,4,7), with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the y dimension (entries 2,3,5,7), and with subbins $\hat{B}_{0,1}$ and $\hat{B}_{1,1}$ in the z dimension (entries 1,2,3,4,6,7). Because only 4 entries are overlapped in the y dimension we compare the queries with those entries. The entry indices are stored *contiguously* in array Y, at indices 7 to 10. So we simply compare the query to the entry segments stored in array Y from index 7 to index 10, which is encoded in constant space. We perform a wasteful comparison with entry segment 5 due to our non-perfect spatial selectivity of entry segments.

On the host, we generate a schedule S, which contains for each query segment q_k a specification of which lookup array to use (0 for X, 1 for Y, or 2 for Z) and an index range into that array, which we encode using 4 integers (which preserves alignment). GPUSPATIOTEMPORAL requires only 1 extra indirection in comparison to GPUTEMPORAL, and avoids storing the

overlapping entry indices in a buffer like in GPUSPATIAL. We then sort S based on the lookup array specification so as to avoid thread serialization due to branching as much as possible. As for GPUTEMPORAL (Section 6.2.2), calculating S on the host takes negligible time.

As explained in the previous section, we enforce a minimum size for the spatial subbins. Ensuring that subbins are not too small is necessary for two reasons. First, with small subbins each entry segment could overlap many subbins with high probability. As a result, the query id would occur many times in arrays X,Y, and/or Z, thereby wasting memory space on the GPU and causing redundant calculations. Second, given our indexing scheme and search algorithm described hereafter, a query that overlaps multiple subbins along all three spatial dimensions may lead to duplicates in the result set. These duplicates would then need to be filtered out (either on the GPU or the CPU). To avoid duplicates, we simply default to the purely temporal scheme whenever duplicates would occur. While this behavior wastes computation (i.e., we lose spatial filtering capabilities), the constraint on subbin size described in the previous section ensure that it occurs with low probability.

The pseudo-code of the search algorithm is shown in in Algorithm 8. It takes the following arguments: (i) the X, Y, and Z arrays; (ii) the database (D); (iii) the query set (Q); (iv) the schedule (S); (v) the query distance (d); and (vi) the memory space to store the result set (*resultSet*). As in Algorithm 7, arguments that lead to array transfers between the host and the GPU are shown in boldface. The algorithm begins by checking the global thread id and aborts if it is greater than $|Q|$ (line 3). The query assigned to the thread is acquired from Q (line 4). A helper array is constructed that holds pointers to the X, Y, and Z arrays (line 5). If schedule S does not give a specification for one of the X, Y, or Z arrays (S[gid].arrayXYZ = -1) then the algorithm defaults to the temporal scheme (line 17). Otherwise, the algorithm retrieves the pointer to the correct X, Y, or Z array (line 7) and determines the index range for the entry segments (lines 8-9). It then processes the entry segments (line 10) as Algorithm 7.

Algorithm 8 GPUSPATIOTEMPORAL kernel.

```
1:  procedure SEARCHSPATIOTEMPORAL (X,Y,Z,D,Q,S,d, resultSet)
2:      gid ← getGlobalId()
3:      if gid≥|Q| return
4:      queryID ← gid
5:      arraySelector ← {X, Y, Z}
6:      if S[gid].arrayXYZ ≠ -1 then
7:          arrayXYZ ← arraySelector[S[gid].arrayXYZ]
8:          entryMin ← S[gid].entryMin
9:          entryMax ← S[gid].entryMax
10:         for all i ∈ {entryMin, ..., entryMax} do
11:             entryID = arrayXYZ[i]
12:             result ← compare(D[entryID],Q[queryID])
13:             if result ≠ ∅ then
14:                 atomic: resultSet ← resultSet ∪ result
15:             end if
16:         end for
17:     else
18:         Lines 5-12 in Algorithm 7.
19:     end if
20:     return
21: end procedure
```

6.3 Experimental Evaluation

6.3.1 Datasets

To evaluate the performance of our various indexing methods we use 3 datasets (1 real world and 2 synthetic) of 4-dimensional trajectories (3 spatial + 1 temporal). In Chapter 5 we have evaluated a purely temporal indexing scheme that shares the general principles of the scheme described in Section 6.2.2 (but assuming that Q cannot fit in GPU memory). In that chapter, we evaluated the performance of distance threshold search for datasets with varying statistical temporal properties, and found the index to perform equally well across these datasets. In this chapter, based on our previous experience and because we consider spatial and spatiotemporal indexing schemes, we use trajectory datasets that vary in terms of their sizes and spatial properties (e.g., spatial trajectory density):

- *Random-1M*: a small, sparse synthetic dataset;

Table 6.1. Characteristics of Datasets

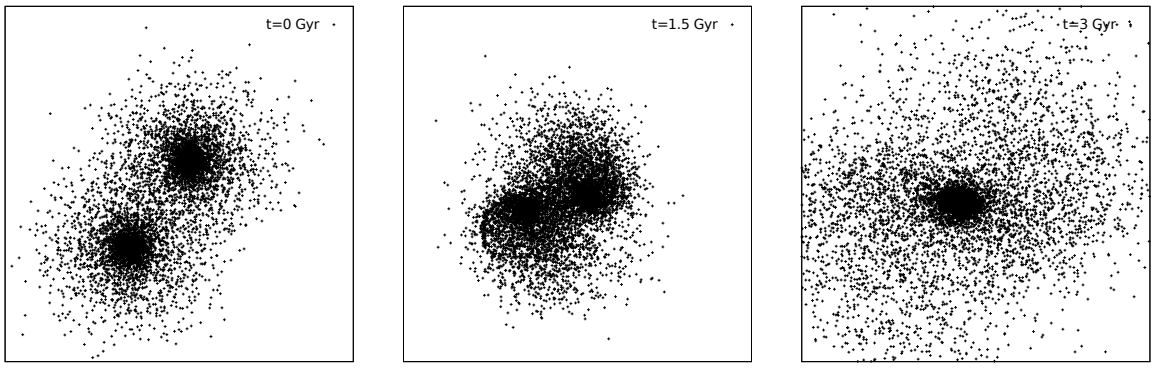
Dataset	Trajectories	Entries
Random-1M	2,500	997,500
Merger	131,072	25,165,824
Random-dense	65,536	12,582,912

- *Merger*: a large, real-world astronomical dataset;
- *Random-dense*: a high density synthetic dataset that is motivated by astronomy applications.

The *Random-1M* dataset consists of 2,500 trajectories generated via random walks over 400 timesteps, for a total of 997,500 entry segments. Trajectory start times are sampled from a uniform distribution over the [0,100] interval.

The *Merger* dataset, as described in Chapter 3, consists of particle trajectories that simulate the merger of the disks of two galaxies. It contains the positions of 131,072 particles over 193 timesteps for a total of 25,165,824 entry segments. Figure 6.5 depicts particles positions projected onto the $x - y$ plane at different times, showing the merger evolution.

The *Random-dense* dataset is generated as follows. Consider the stellar number density of the solar neighborhood, i.e., at galactocentric radius $R_\odot = 8$ kpc (kiloparsecs), of Reid et al. [52], $n_\odot = 0.112$ stars/pc^3. We develop a dataset with the same number of particles of one disk in the *Merger* dataset (65,536), and 193 timesteps. To match the density of [52], we require a volume of $65536/0.112 = 585142$ pc^3. This yields a cube with length, width and height dimensions of 83.64 pc. Note that we could have made the dataset more spatially dense by picking a region close to the galactic center, since the stellar density decreases as a function of R. We generate actual trajectories as random walks as in the *Random-1M* dataset, where all of the particles are initially populated within the aforementioned cube. We allow the trajectories to move a variable distance in each of the x,y,z dimensions at each timestep (between 0.001 and 0.005 kpc), and if a particle moves outside of the cube by 20% of the length of the cube in any dimension, the particle is forced back towards the cube. The particles, on average, cannot travel too far from the cube such that we maintain a fairly consistent trajectory density at each timestep. This dataset aims to represent a density consistent within the range of possible densities within the Milky Way that a single node might process. The characteristics of each dataset are summarized in Table 6.1.



(a) (b) (c)

Figure 6.5. Sample particle positions in the *Merger* dataset at times 0 Gyr (a), 1.5 Gyr (b) and 3 Gyr (c).

6.3.2 Experimental Methodology

For all our distance threshold search implementations the GPU-side is developed in OpenCL and the host-side is developed in C++. The host-side implementation is executed on one of the 6 cores of a dedicated 3.46 GHz Intel Xeon W3690 processor with 12 MiB L3 cache. The GPU-side implementation runs on an Nvidia Tesla C2075 card with 6GiB of RAM and 448 cores. In all experiments we measure query response time as an average over 3 trials (standard deviation over the trials is negligible). We allocate a buffer to hold the result set of the search on the GPU that can hold 5.0×10^7 items. In the description of the results we indicate when this buffer is overcome, thus requiring incremental processing of the query. The response time does not include the time to build the index or the time to store D and the index in GPU memory. These operations can be performed off-line before query processing begins. The implementations have been validated to ensure correctness. To guarantee that we do not obtain false positive or negative results, we compare the results of our implementation to an alternate implementation that utilizes a brute force approach.

We consider three experimental scenarios, each for one of our datasets:

- S1: The *Random-1M* dataset and a query with 100 trajectories each with 400 timesteps for a total of 39,900 query segments.
- S2: The *Merger* dataset and a query set with 265 trajectories each with 193 timesteps for a total of 50,880 query segments.

- S3: The *Random-dense* dataset and a query set with 265 trajectories each with 193 timesteps for a total of 50,880 query segments.

For each scenario we use ranges of query distances (in units of kpc for S2 and S3).

In addition to our GPU implementations, we also evaluate a CPU-only implementation. This implementation relies on an in-memory R-tree index, and is multithreaded using OpenMP. Threads traverse the R-tree in parallel, each for a different query segment, and return candidate entry segments. This implementation was developed in Chapter 4. In that work we investigated "trajectory splitting," i.e., the impact of the number of segments stored in each MBB in the R-tree index, r. There is a trade-off between the time to search the index (which decreases as r increases due to lower tree depth) and the time to process the candidate (which increases as r increases due to higher index overlap). All executions of the CPU implementation use 6 threads on our 6-core CPU. Results in Chapter 4 show that this implementation achieves high parallel efficiency. Like for the GPU implementation, our response time measurements do not include the time to build the index tree.

Although the experimental results in the following sections are constrained by the specifics of our platform, the results of the CPU implementation are used to demonstrate that the GPU can be used efficiently for distance threshold searches. A fundamental difference between the CPU implementation and the GPU implementation is that the former relies on index-tree traversal while the latter relies on flat indexing schemes. This is because tree traversals on the GPU are problematic, e.g., due to thread divergence slowdowns.

6.3.3 Results for the *Random-1M* Dataset

In this section, we present results for the *Random-1M* dataset, first giving results for individual implementations and then combining results that make it possible to compare the implementations. The *Random-1M* dataset is representative of small and sparse datasets in which few or no entry segments are expected to lie within distance d of a query segment, i.e., with a low number of *interactions*.

Figure 6.6 shows response time vs. the number of entry segments per MBB (r) for the CPU implementation for a range of query distances. Using a single entry segment per MBB ($r = 1$)

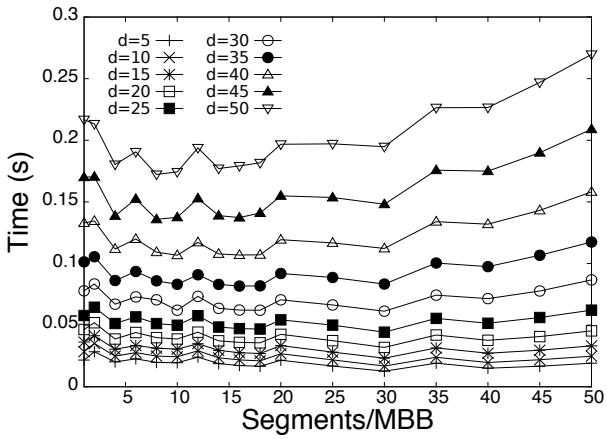


Figure 6.6. Response time vs. number of entry segments per MBB (r) for the CPU implementation in scenario S1 with $d = 5, 10, \ldots, 50$.

does not lead to the best response time. For this experimental scenario using, e.g., $r = 10$ leads to good response times across all query distances.

Figure 6.7 plots response time vs. d for GPUSPATIAL. Results are shown for a range of grid sizes. We use a total buffer size, $|U|$, of 2GiB to store overlapping entry segments. This is larger than the space necessary to store D. This is thus an optimistic configuration for the FSG index. Using too few grid cells leads, e.g., 10 per dimension, to poor performance due to poor spatial selectivity. With poor spatial selectivity (i) a large candidate set must be processed and (ii) many GPU threads overflow their entry buffers (U_k) thus requiring multiple query processing attempts. Likewise, using too many grid cells also leads to poor performance because entry segments overlap multiple cells. As a result there is duplication of index entries, and thus in the result set. Although filtering out these duplicates takes negligible time, transferring them from the GPU back to the host incurs non-negligible overhead. In these experiments, and among the FSG configurations we have attempted, using 50 cells per dimension leads to the best result.

Regardless of FSG configuration, we see rapid growth in response time as d increases. The disposition of the FSG index to prefer small d values has also been alluded to in [71]. This suggests that FSGs may not be particularly useful for spatiotemporal trajectory searches due to the large spatial extent of the data and absence of temporal discernment, unless query distances are small. However, a FSG index is likely to perform well with fewer requirements, such as indexing

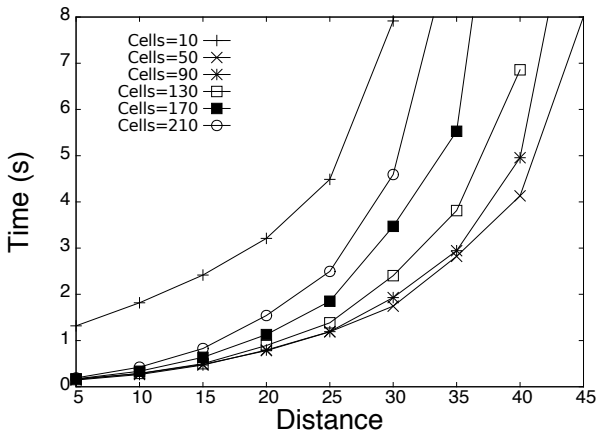


Figure 6.7. Response time vs. d for GPUSPATIAL in scenario S1. Different curves are shown for different numbers of spatial cells in the x, y, and z dimensions (i.e., "Cells=10" means a $10\times10\times10$ grid).

data with no temporal dimension, and/or focusing on point searches (instead of line segments), which will not cause data duplication when a large number of grid cells is used.

Figure 6.8 is shows response times vs. d for GPUTEMPORAL. Results are shown for a range of number of temporal bins. Unlike for GPUSPATIAL, this method is insensitive to the query distance. With too few temporal bins there is not enough temporal discrimination leading to large numbers of interactions. But as the number of bins increases the response time reaches a minimum (increasing beyond 10,000 bins does not differentiate entries as a function of temporal extent in the *Random-1M* dataset).

Figure 6.9 shows response time vs. the number of subbins for GPUSPATIOTEMPORAL, where 10,000 temporal bins are used. Curves are shown for a range of d values. For low d values a greater number of spatial subbins is desirable. This is because it is unlikely that a query will overlap multiple subbins, which would cause our algorithm to revert to the purely temporal method, which has no spatial selectivity. As d increases, queries overlap multiple spatial subbins with higher probability. As a result, better performance is achieved with fewer subbins. Recall that we require that a query fall within a single subbin so as to avoid duplication in the result set. Without this requirement, an increasing number of subbins would suggest an increase in the duplication of entries in the index, thereby increasing the number of candidates that need to be processed (the same trade-off discussed for GPUSPATIAL). There is thus a trade-off between having too few or too many subbins, even when duplicates in the result set are permitted.

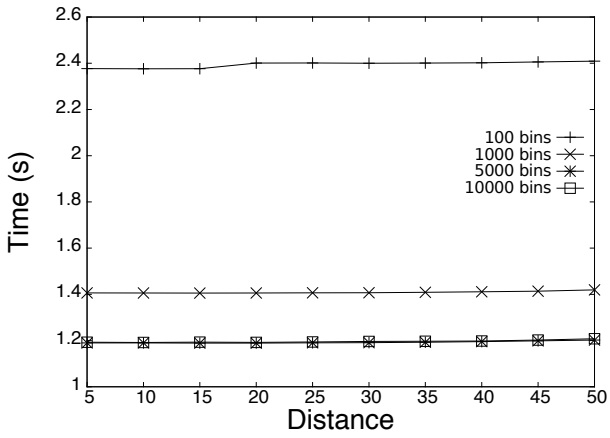


Figure 6.8. Response time vs. d for GPUTEMPORAL in scenario S1. Different curves are shown for different numbers of temporal bins (100, 1000, 5000, 10000).

We note that using 1 subbin in the spatiotemporal index is equivalent to using a purely temporal index with no spatial selectivity. Comparing results between GPUSPATIOTEMPORAL with 1 subbin and GPUTEMPORAL shows the effect of the additional indirection in the spatiotemporal index. At $d = 50$ (yielding the greatest number of indirections in S1), with 1 subbin in the spatiotemporal index, the response time is 1.36 s, whereas the response time is 1.21 s when using the temporal index without any indirection. This is a 12.4% increase in response time due to the indirection.

Figure 6.10 shows response time vs. d for our four implementations. Each implementation is configured with best or good parameter values based on previous results in this sections (see the caption of Figure 6.10 for details). The CPU implementation is best across all query distances. Comparing the GPU implementations, we see that GPUSPATIAL performs better than GPUTEMPORAL and GPUSPATIOTEMPORAL when $d < 20$, but that it does not scale well for larger d values. One may wonder whether this lack of scalability comes from the overhead of relaunching the kernel due to buffer overflows. Figure 6.10 plots an "optimistic" curve that discounts this overhead. We see that the same trend, if not as extreme, remains. The temporal and spatiotemporal indexing methods have consistent response times across query distances. Note that we could have selected the best number of subbins for each value of d from Figure 6.9, which would have improved results. Comparing GPUTEMPORAL and GPUSPATIOTEMPORAL, we see that having spatial selectivity in addition to temporal indexing provides performance gains, even on this small

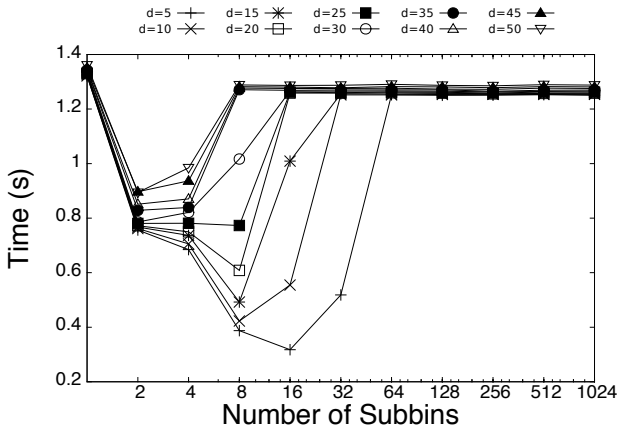


Figure 6.9. Response time vs. the number of subbins (v) for GPUSPATIOTEMPORAL in scenario S1. The number of temporal bins is set to 10,000. Different curves are shown for different query distances ($d = 5, 10, ..., 50$).

and sparse dataset. The CPU R-tree approach outperforms the GPU approaches on this small and sparse trajectory dataset.

6.3.4 Results for the *Merger* Dataset

In this section, we present results for our largest dataset, *Merger*, which contains over 25 million entry segments. From Section 6.3.3, we find that the purely spatial FSG method leads to extremely high response times for this larger dataset and as a result, we do not consider it. In GPU executions, for some values of d, we have to process Q incrementally due to to insufficient space for storing the full result set on the GPU. This is reflected in the measured response times.

Figure 6.11 shows response time vs. r for the CPU implementation for 3 query distances. With this large dataset, unlike with *Random-1M*, storing more than $r = 1$ segments per MBB leads to higher response time. A higher r value decreases the time to search the R-tree index, but this benefit is offset by the increase in candidate set size. This is an important result. There is a literature devoted to assigning trajectory segments to MBBs for improving response time [22, 51], including the work in Chapter 4. These works, however, do not consider large datasets. For these datasets, an intriguing future research direction is to take the opposite approach as that advocated in the literature: splice individual polylines to increase the size of the dataset (which can be thought of as setting $r < 1$). Thus, instead of assigning a single trajectory segment to an MBB, a trajectory

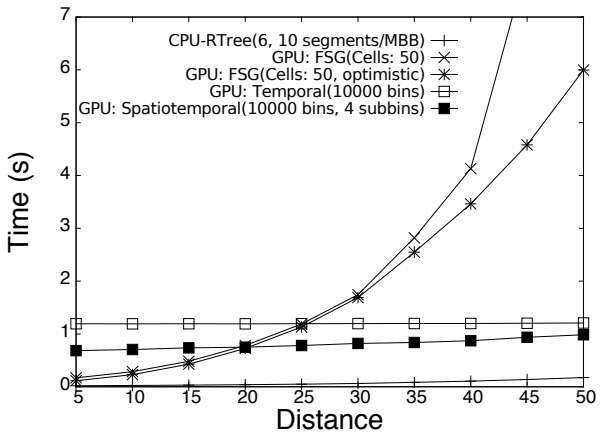


Figure 6.10. Response time vs. d for our implementations for scenario S1. For the CPU implementation we use $r = 10$ segments/MBB; for GPUSPATIAL we use 50 cells per spatial dimension; for GPUTEMPORAL we use 10,000 bins; and for GPUSPATIOTEMPORAL we use 10,000 temporal bins and $v = 4$ spatial subbins: For GPUSPATIAL we also plot an optimistic curve that ignores kernel re-launch overheads.

segment is split into multiple pieces, requiring multiple MBBs. This scheme would increase the number of entries in the index; however, the total volume occupied by the MBBs would decrease.

We do not show results for GPUTEMPORAL as they are similar to those for the *Random-1M* dataset. Using 1,000 temporal bins leads to the lowest response time, which is consistent across all query distances.

Figure 6.12 shows response time vs. number of subbins for GPUSPATIOTEMPORAL, where 1,000 temporal bins are used. Curves are shown for a range of d values. A good number of subbins is $v = 16$ across all query distances, and this value is in fact best for most query distances. This implies that picking a good v value can likely be done for a dataset regardless of the queries. Figure 6.9 shows a dependency between v and d for the *Random-1M* dataset. This dependency vanishes for a large dataset with many interactions.

Figure 6.13 compares the performance of the CPU implementation and GPUTEMPORAL and GPUSPATIOTEMPORAL (GPUSPATIAL is omitted). Each method is configured with best or good parameter values based on results in Figures 6.11 and 6.12. GPUSPATIOTEMPORAL outperforms GPUTEMPORAL across the board, with response times at least 23.6% faster. At low query distances the CPU implementation yields the lowest response times. It is overtaken by GPUSPATIOTEMPORAL at $d \sim 1.5$. At $d = 0.001$ the response time for the CPU implementation

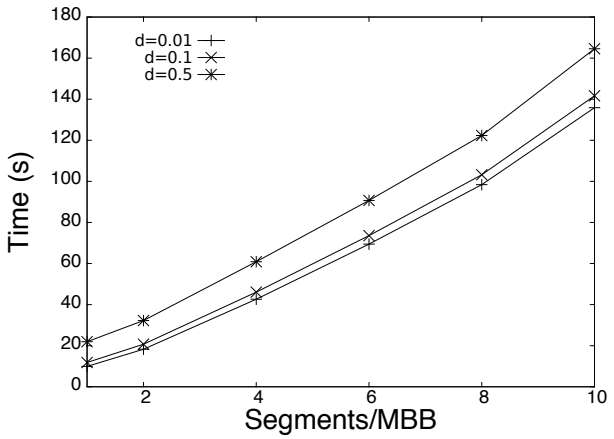


Figure 6.11. Response time vs. number of entry segments per MBB (r) for the CPU implementation in scenario S2 with $d = 0.01, 0.1, 0.05$.

is 9.70 s vs. 41.75 s for GPUSPATIOTEMPORAL (the GPU implementation is 330.4% slower). At $d = 5$ these response times become 184.4 s, and 116.09 s, respectively (the GPU implementation is 58.8% faster). We conclude that the GPU implementation outperforms the CPU implementation when using large datasets or when large query distances are considered.

6.3.5 Results for the *Random-dense* Dataset

We now present results for the *Random-dense* dataset, which is smaller than *Merger* and representative of scenarios in which many trajectories are located in a small spatial region, as motivated by the stellar number density at the solar neighborhood. Note that increasing the density by even $> 4\times$ would still be consistent with that resembling the disk in the inner Galaxy.

Figure 6.14 shows response time vs. query distance for the CPU implementation for $r = 1, 2, 4, 8$. Unlike for *Merger*, which has $2\times$ the number of entries as *Random-dense*, storing multiple segments/MBB improves response time. We find that $r = 4$ yields low response time values across all query distances.

As in the previous section, we do not show results for GPUTEMPORAL as they are similar to those for the *Random-1M* dataset. Using 1,000 temporal bins leads to the lowest response time, which is consistent across all query distances.

Figure 6.15 (a) shows response time vs. the number of subbins (v) for S3 for GPUSPATIOTEMPORAL. With this dataset, the use of subbins for reducing response time is only possible

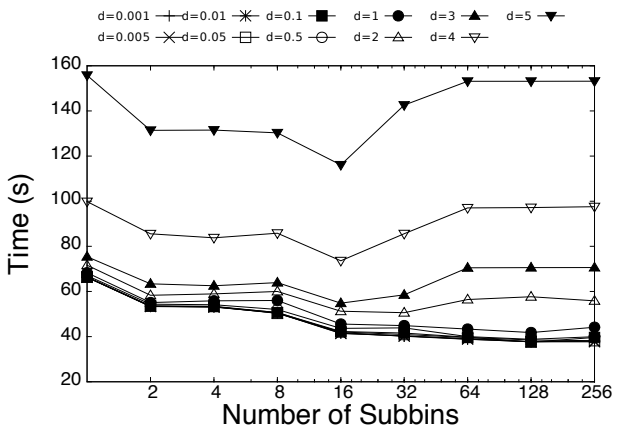


Figure 6.12. Response time vs. the number of subbins (v) for GPUSPATIOTEMPORAL in scenario S2. The number of temporal bins is set to 1,000. Different curves are shown for different query distances between $d = 0.001$ and $d = 5$.

for small query distances ($d = 0.001, 0.01, 0.03$). This is because the dataset is smaller than *Merger* and because with larger values of d, the queries are more likely to fall within multiple subbins (in which case the search algorithm degenerates into a purely temporal scheme). Figure 6.15 (b) shows the fraction of queries that utilized the entries provided by the subbins for $d = 0.001, 0.01, 0.03$. Only the smallest query distance, $d = 0.001$, permits usage of the spatiotemporal index across a sizable fraction of the number of subbins. For instance for $d = 0.03$ and $v = 2$, just over 60% of the queries use the spatiotemporal index over the pure temporal index, and when $v = 4$, the entries provided by the spatiotemporal index are not used. This explains why in Figure 6.15 (a), there is no performance improvement for $d > 0.03$ when v increases.

Given the density of the dataset, for larger values of d, only a fraction of the queries can be solved per kernel invocation as there is insufficient memory space for the result set. Since *Random-dense* has half as many entries as *Merger*, we can increase the size of the buffer on the GPU for the result set (from 5×10^7 elements for *Merger* to 9.2×10^7 elements for *Random-dense*). Figure 6.16 shows the response time vs. d for GPUTEMPORAL and GPUSPATIOTEMPORAL with two buffer sizes. Increasing the buffer size by 84% (thus requiring fewer kernel invocations) leads to decreases in response time due to fewer host-GPU communications. For instance, at $d = 0.09$ (which requires the greatest number of kernel invocations), the spatiotemporal index, with $v = 2$, using an increased buffer size for the result set has a response time that is 65.76% lower than with the initial buffer size. Although we could not run experiments with a larger buffer size for scenario

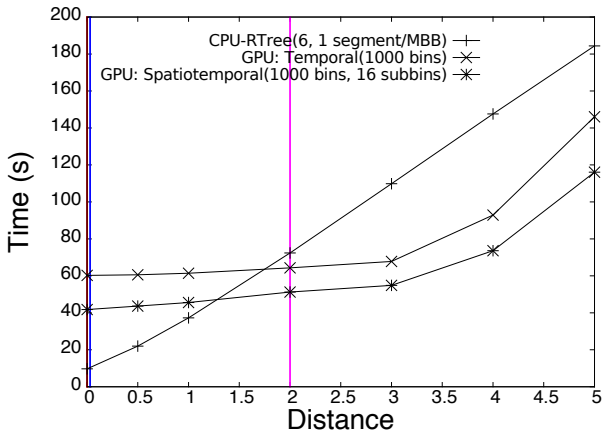


Figure 6.13. Response time vs. d for our implementations for scenario S2. For the CPU implementation we use $r = 1$ segments/MBB; for GPUTEMPORAL, we use 1,000 bins; for GPUSPATIOTEMPORAL, we use 1,000 temporal bins and $v = 16$ spatial subbins. We indicate three distance thresholds that would be interesting for the study of the habitability of the Milky Way based on such datasets. Red: close encounters between stars and planetary systems [25]; Blue: supernova events on habitable planetary systems [19], and Magenta: studying the effects of gamma ray bursts on habitable planets [63]. Both the Red and Blue lines are close to the vertical axis.

S2 (due to the large size of the *Merger* dataset), we expect similar performance gains. Since current trends point to improvements in host-to-GPU bandwidth, in the future, our indexing methods should provide even better performance improvements compared to CPU implementations.

Figure 6.17 shows response time vs. d for the CPU implementation and GPUTEMPORAL and GPUSPATIOTEMPORAL with the larger buffer sizes. The query distance range spans a wide range of result set sizes. When $d = 0.001 \approx 0\%$ of the entries are within the query distance, and when $d = 0.09$, 73.9% of the entries are within the query distance. For very small query distances $d \lesssim 0.02$, the CPU implementation yields the lowest response time, and is outperformed by the GPU implementations for larger d. For $d > 0.03$, GPUSPATIOTEMPORAL performs slightly worse than GPUTEMPORAL. This suggests that for dense datasets, when moderate to large query distances are required, the pure temporal indexing method performs the best. At $d = 0.05$, GPUTEMPORAL is 223% faster than the CPU implementation (with $r = 4$).

Comparing Figures 6.13 (*Merger* dataset) and 6.17 (*Random-dense* dataset), we see that the range of query distances for which the GPU method is preferable to the CPU method is much larger for the *Random-dense* dataset (considering the query distances that correspond to relevant application scenarios – the red, blue, and magenta vertical lines). In the astronomy application

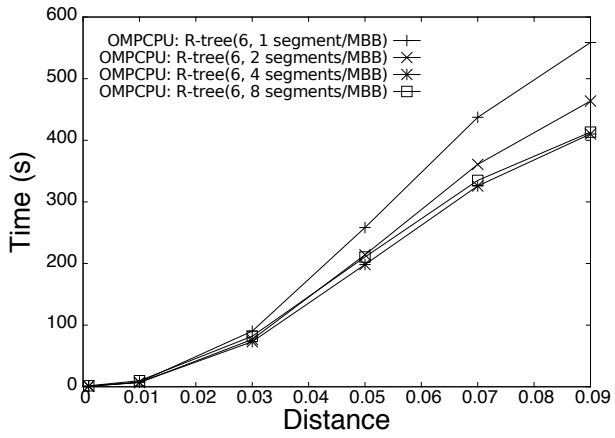


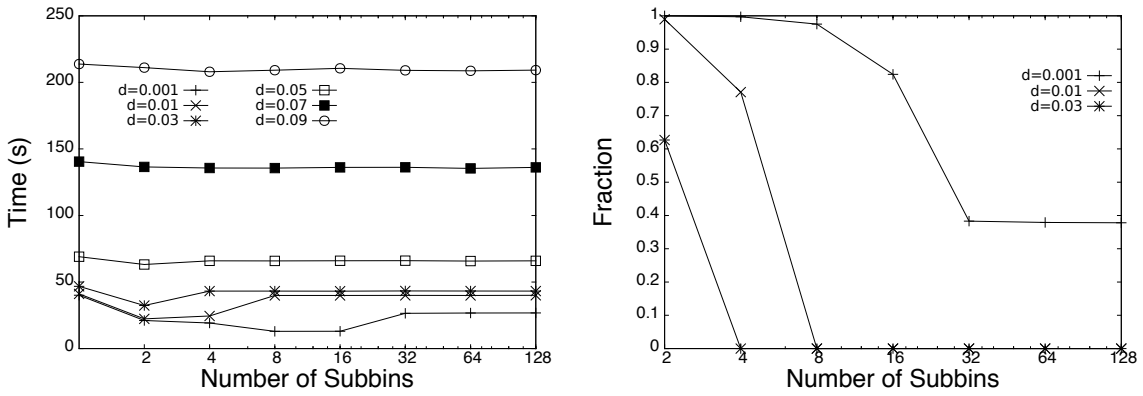
Figure 6.14. Response time vs. d for the CPU implementation in scenario S3. Different curves are shown for different values of r (1,2,4, and 8).

domain, datasets denser than the *Random-dense* dataset are relevant (i.e., to study the galactic regions at $R < 8$ kpc). For these datasets a GPU approach will provide even more performance improvement over a CPU implementation.

To summarize our results, Figure 6.18 shows the ratio of the response times of the GPU to CPU implementations for the 3 datasets for a few representative query distances. Data points below the $y = 1$ line correspond to instances in which the GPU implementation outperforms the CPU implementation. The main findings are that although the CPU is preferable for small and sparse datasets (Figure 6.18 (a)), the GPU leads to significant improvements for large and/or dense datasets (Figure 6.18 (b)) unless query distances are very small.

6.4 Conclusions

In this chapter, we have developed algorithms and indexes designed for GPU executions of the distance threshold search. We have considered a scenario where there are no memory constraints when storing the queries on the GPU with the database. We proposed three GPU indexing schemes and corresponding GPU kernels. We find that GPUSPATIAL which relies on a FSG does not achieve good performance for our trajectory searches. We find that our other two indexing methods for the GPU, GPUTEMPORAL and GPUSPATIOTEMPORAL, outperform multicore CPU implementations in a range of experimental scenarios. The main findings are that although the



(a) (b)

Figure 6.15. (a) Response time vs. number of subbins (v) for GPUSPATIOTEMPORAL for scenario S3 for a range of query distances. The number of temporal bins is set to 1,000. (b) The fraction of queries that use the entries provided by subbins vs. the number of subbins (v).

CPU is preferable for small and sparse datasets, the GPU leads to significant improvements for large and/or dense datasets unless query distances are small. When there are large query distances, or a dense dataset is considered, the parallelism afforded by the GPU is beneficial and the overhead of using the GPU is a small fraction of the total response time. However, when the dataset is sparse and/or the query distance is small, this overhead precludes performance gains when using the GPU. These results are encouraging, as large and dense datasets are both characteristics of our target astrobiological application. We also find that the notion of splitting trajectories and storing them inside of MBBs is not necessarily applicable for large datasets. This result should apply to other trajectory similarity searches, such as kNN searches.

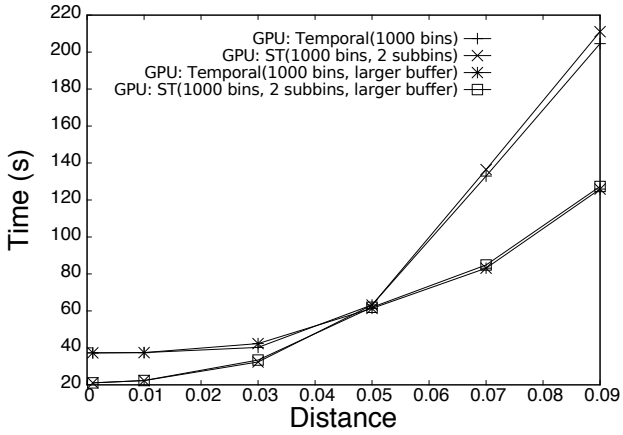


Figure 6.16. Response time vs. d for GPUTEMPORAL and GPUSPATIOTEMPORAL for scenario S3. Results are shown for the original buffer size (5×10^7) and for a larger buffer size (9.2×10^7).

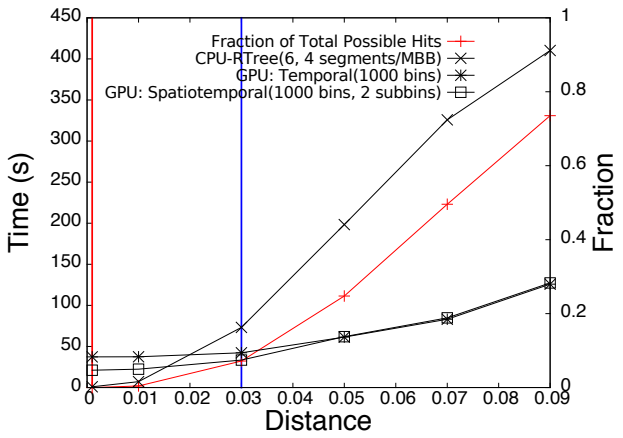
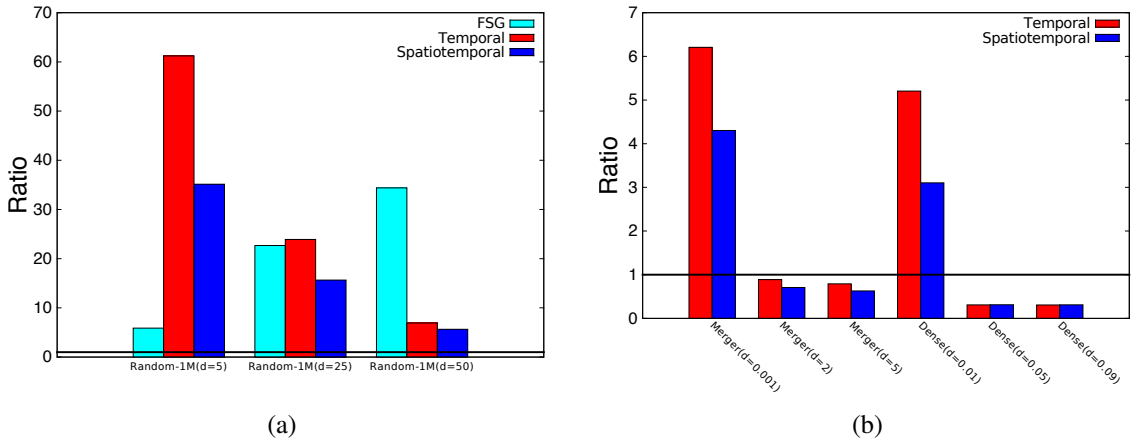


Figure 6.17. Response time (left vertical axis) and fraction of entries with distance d of the query (right vertical axis) vs. d for the CPU implementation, GPUTEMPORAL, and GPUSPATIOTEMPORAL for scenario S3. For the CPU we show results for $r = 4$. 1,000 temporal bins are used for both the temporal and spatiotemporal indexing methods. $v = 2$ spatial subbins are used for the spatiotemporal indexing method.



(a) (b)

Figure 6.18. Ratio of GPU to CPU response times across all datasets for (a) S1 and (b) S2 and S3. Values below the $y = 1$ line indicate improvements over the CPU implementation.

Chapter 7

Conclusions and Future Work

Distance threshold similarity searches on moving object trajectories are relevant in many application areas, including the application in astrobiology on the habitability of the Milky Way outlined in Chapter 1. In this dissertation, we have developed efficient algorithms for processing distance threshold trajectory similarity searches. The distance threshold search performs a join on a database of trajectories that are within a given query distance of a query trajectory.

The performance of the distance threshold search on trajectories is data dependent and leads to many challenges. In particular, the number of trajectories in a database, the search distance, d, and the spatiotemporal nature of the data all affect the effectiveness of a trajectory search to generate a candidate set of trajectory segments and the subsequent refinement process. Furthermore, achieving good performance in sequential and parallel implementations is also a function of the selected platform. We reiterate the following challenges in this work:

- Developing efficient trajectory indexes is difficult, as it needs to have a good degree of spatial and/or temporal selectivity for an arbitrary set of queries. Consequently, this means that there is no optimal way to index the data to achieve the best performance, as differing sets of query trajectories will have varying properties and thus, response times. Additionally, the underlying architecture, such as the CPU or the GPU and associated memory hierarchies need to be carefully considered to obtain a good data layout primarily for search algorithms, but also for refining the candidate set to obtain the final result set.

- The properties of the trajectories in a given dataset lead to non-deterministic execution and storage behavior. As a result, estimating the query response time across a range of trajectory configurations is difficult.

- GPU implementations may be unable to use indexes suitable for the CPU. Unlike parallelizing trajectory similarity searches on the CPU that may have access to a considerable amount of main memory, searches on the GPU have greater memory constraints that need to be considered. Indexes that achieve good performance for implementations using the CPU, such as index-trees, often have non-linear space-complexities and may require too much memory on the GPU. Furthermore, differences in architectures, in particular SIMD, necessitate algorithms that minimize branch instructions which reduce parallel efficiency.

Given these overarching challenges, we now turn to the contributions made in this work, and possible future research directions.

7.1 Contributions

Towards a resolution to the challenges in the previous section, we make the following contributions:

- In Chapter 4 we present in-memory sequential and parallel query processing algorithms for the CPU.

- In Chapter 5 we propose GPGPU algorithms that avoid index-trees altogether that are used in CPU implementations and instead features a GPU-friendly indexing scheme. Additionally, we advance a response time performance model of this execution.

- In Chapter 6 we develop three indexing strategies with spatial, temporal and spatiotemporal selectivity for the GPU that differ significantly from indexes suitable for the CPU, and show the conditions under which each index achieves good performance.

We detail these contributions in the following three sections.

7.1.1 Trajectory Searches using Sequential and Multithreaded Implementations

In-memory distance threshold searches for trajectory and point searches on moving object trajectories are significantly different from the well-studied kNN searches [16, 14, 17, 20]. 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 [16, 14, 17, 20].

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 showed 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 (72.2%-85.7% in our experiments).

7.1.2 Trajectory Searches using GPGPU with Memory Constraints

We have studied the efficient execution of in-memory distance threshold searches on the GPU. The objective is to minimize response time in an online setting in which a series of kernel invocations are performed to process a potentially large query set. We have shown that the par-

allelism afforded by the GPU, provided a GPU-friendly indexing method is used, can outperform multithreaded CPU implementations that use an in-memory R-tree index. We have proposed such a GPU-friendly indexing method. While conceptually simple, this method may be suitable for indexing spatial and spatiotemporal objects for parallel architectures in general, as described in [72]. We have proposed several algorithms for partitioning a query set into individual batches, so as to reduce memory pressure and computational cost on the GPU. We have found that, when considering the cost to compute the batches, a simple algorithm that partitions the query set into fixed-size batches leads to competitive response times.

Modeling the performance of algorithms that process moving objects is a challenge due to the spatiotemporal nature of data. Furthermore, in the context of spatiotemporal databases, where index-trees are paramount, the non-deterministic nature of tree traversals adds an additional source of performance uncertainty. The indexing method proposed in this work obviates some of this data-dependent uncertainty. As a result, we are able to derive a reasonably accurate response time model. This model, which considers both CPU and GPU time, is sufficient for predicting a good batch size for a given dataset. Furthermore, in some instances, the model is adequate to estimate the actual query response time across a range of query batch sizes. This result is encouraging, as it suggests that predicting query response time on the GPU, at least with some indexing techniques, is feasible, making it possible to assess the tractability of spatiotemporal queries across a range of application domains. In particular, such query response time prediction will be crucial for estimating the compute time for the astrophysical application that is the initial motivation for this work.

7.1.3 Efficient Indexing of Trajectories on the GPU

We have examined the performance of indexes used for efficient indexing of spatiotemporal trajectories for distance threshold similarity searches. We describe several algorithms to solve the distance threshold search on the GPU, as suited to each index. We show that GPU-friendly indexing methods can outperform a multicore CPU implementation that uses an in-memory R-tree index. Furthermore, the indexes proposed here can be applied to other types of searches on spatial and spatiotemporal data. We also find that when considering the R-tree implementation on the CPU, the problem of splitting a trajectory and storing it in multiple MBBs to achieve a trade-off

between the time to search the index and the time to process the set of candidate segments is annulled when a large dataset is utilized, as a small number of segments per MBB can lead to the best response time.

7.2 Future Work

The contributions made in this work can lead to numerous other interesting offshoots, whether to extend distance threshold search efficiency/throughput, or to test the utility of certain facets of distance threshold searches on other types of spatial or spatiotemporal searches on moving objects in general. In what follows, we elaborate on several future research directions.

7.2.1 Trajectory Indexing for In-memory CPU-based Implementations

In the in-memory implementations that rely on the R-tree index, we investigated efficient trajectory splitting strategies. 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 [8].

When we ran experiments with the largest of our datasets, we found that the utility of splitting trajectories and assigning them to MBBs was diminished because a more accurate search which minimized the size of the candidate set was preferable to faster searches at the expense of refining a larger candidate set. Since trajectory splitting strategies have been studied in the literature, it would be interesting to determine what set of generalizable conditions leads to this performance loss. Furthermore, if these conditions can be understood, then a resolution to this problem would be a significant achievement for the study of large-scale spatial and spatiotemporal applications.

Exploring analytical performance models is an interesting future research direction because query and entry segment data characteristics lead to non-deterministic trajectory searches and candidate set refinements. Models of query performance in these settings may be heavily dependent on modeling cache reuse.

7.2.2 Modeling the Performance of Searches on Spatiotemporal Objects Using GPGPU

One interesting result we obtained was obviating some of the data-dependent uncertainty associated with using index-trees in CPU implementations of distance threshold searches by using our GPGPU implementation and associated GPU-friendly index. Because we have advanced this index, we have developed performance models of the distance threshold search. Therefore, it would be interesting to extend our performance modeling technique to other spatiotemporal queries, such as the kNN query on trajectories, and other searches on spatiotemporal objects. If the model of response time is accurate in other settings, then performance bottlenecks will be easier to identify in other searches on moving objects.

7.2.3 Extension to kNN Searches on Trajectories

Regarding the in-memory R-tree index for the CPU, and trajectory splitting strategies, one may wonder whether the idea of assigning multiple segments to an MBB is applicable for kNN searches on trajectories [16, 14, 17, 20]. The kNN 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, kNN 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 kNN searches. An interesting problem is to reconcile the differences between kNN and distance threshold searches in terms of index resolution.

From our experiments, we found that CPU-based implementations perform poorly in comparison to the GPU implementations when there are many comparisons between queries and entries in the database, which occurs with large datasets and query distances. While the kNN search on trajectories is not expected to perform well for small values of k on the GPU, in large-scale applications that require a sufficiently large value of k, it might be worthwhile to perform the search on the GPU. Furthermore, since index-trees may not be a good indexing technique for the GPU, one option would be to perform a distance threshold search using one of the indexes proposed for the GPU in this work to obtain a candidate set of neighbors, order them by moving distance (shortest to farthest), and then select the k nearest ones to the search trajectory.

7.2.4 Hybrid CPU-GPU Implementations

Throughout this work, we have shown that the GPU and CPU implementations perform well under particular conditions, and that it is likely that the CPU will not be able to contend for the niche occupied by the GPU, and vice-versa. Therefore, one may wonder if a hybrid approach that considers a multithreaded CPU implementation that uses an index-tree combined with a GPU implementation would be able to achieve performance gains not realized in this work. Interestingly, this problem will likely focus on query scheduling, where those query segments deemed to have a higher probability of having a large result set are sent to the GPU, and those query segments that have few interactions are scheduled on the CPU. For such a hybrid approach to achieve good performance, load balancing will be of paramount importance. Given a set of $|Q|$ query segments, there will be $2^{|Q|}$ possible schedules, or ways to partition the queries to be run on either the CPU or GPU. Efficient scheduling heuristics, that rely on response time performance modeling, will need to be developed.

7.2.5 Distributed Memory Implementations

We did not consider distributed memory executions, and only considered shared-memory executions at a single compute node. Performing distance threshold similarity searches on hundreds of millions to billions of trajectories requires distributed memory implementations. Whether executed on a cluster consisting of multi-core CPUs, many-core GPUs, or both if using a hybrid approach, there will be additional obstacles related to distributed memory computing that will need to be addressed, such as: (i) partitioning the dataset over the nodes to achieve good load balancing, and (ii) reducing communication overheads between nodes. Such issues have been investigated for countless parallel applications and applying some underlying principles to distributed memory distance threshold searches will be necessary to scale up to large-scale datasets.

Appendix A

Performance Evaluation of Query Segment Batches

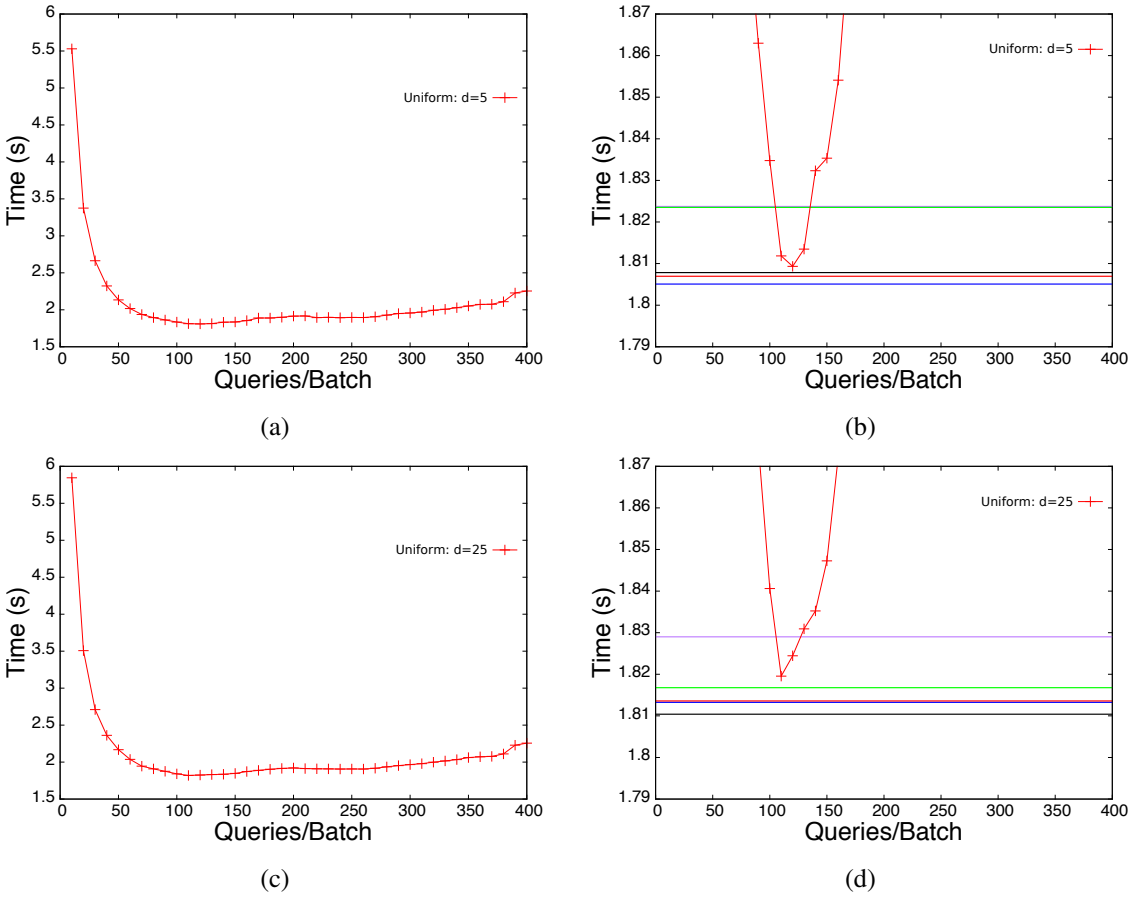


Figure A.1. Response time vs. queries/batch (s) for the periodic query batch method for S3 (a) and S4 (c) (RANDWALK-UNIFORM dataset). Panels (b) and (d) correspond to zoomed in versions of (a) and (c) respectively, to highlight the minimum response times. The colored lines correspond to the same algorithms as shown in Figure 5.9.

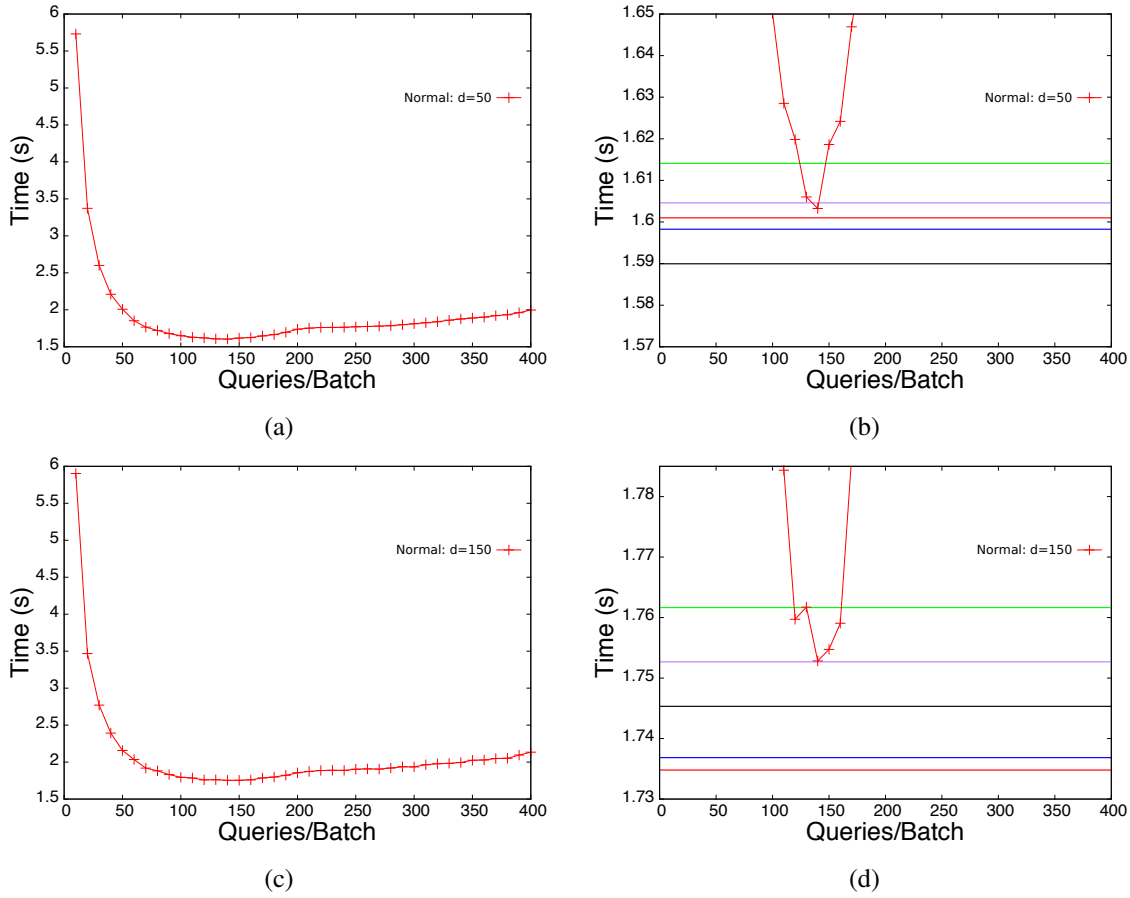


Figure A.2. Response time vs. queries/batch (s) for the periodic query batch method for S5 (a) and S6 (c) (RANDWALK-NORMAL dataset). Panels (b) and (d) correspond to zoomed in versions of (a) and (c) respectively, to highlight the minimum response times. The colored lines correspond to the same algorithms as shown in Figure 5.9.

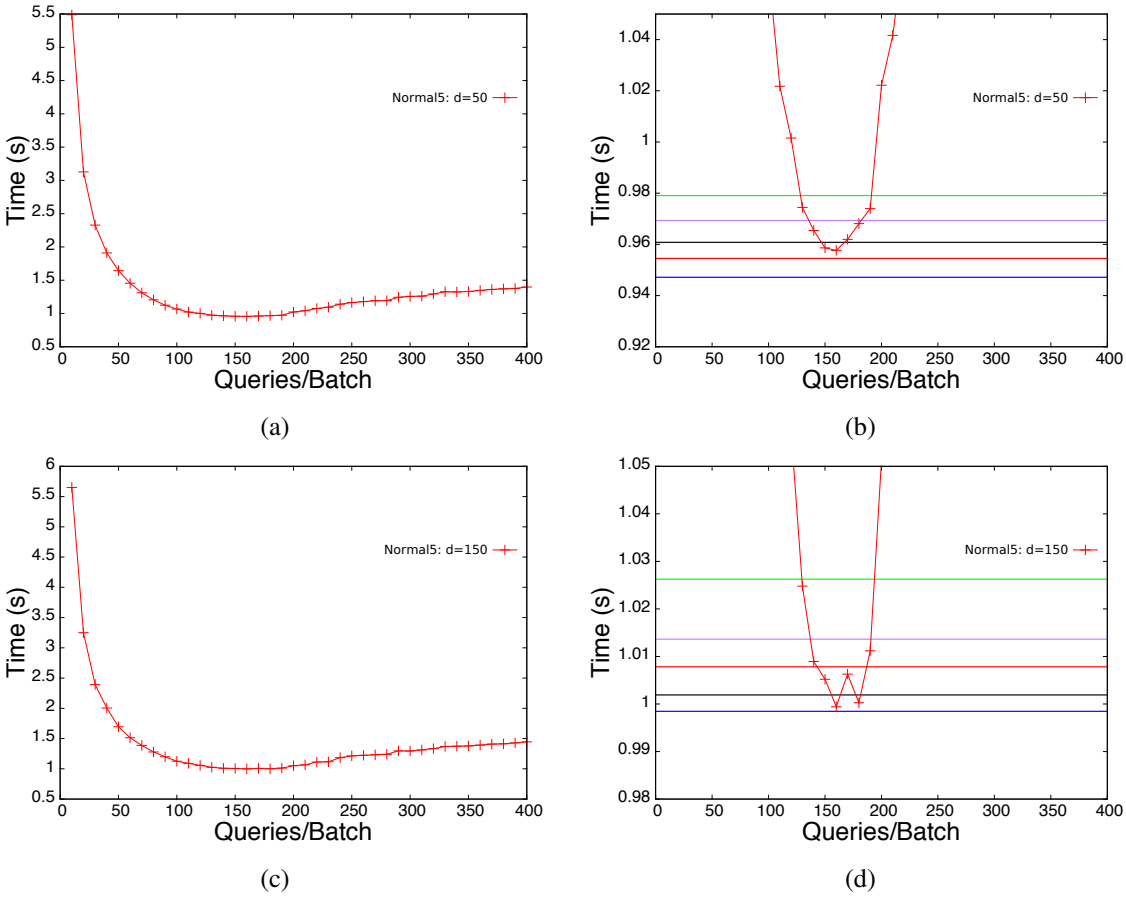


Figure A.3. Response time vs. queries/batch (s) for the periodic query batch method for S7 (a) and S8 (c) (RANDWALK-NORMAL5 dataset). Panels (b) and (d) correspond to zoomed in versions of (a) and (c) respectively, to highlight the minimum response times. The colored lines correspond to the same algorithms as shown in Figure 5.9.

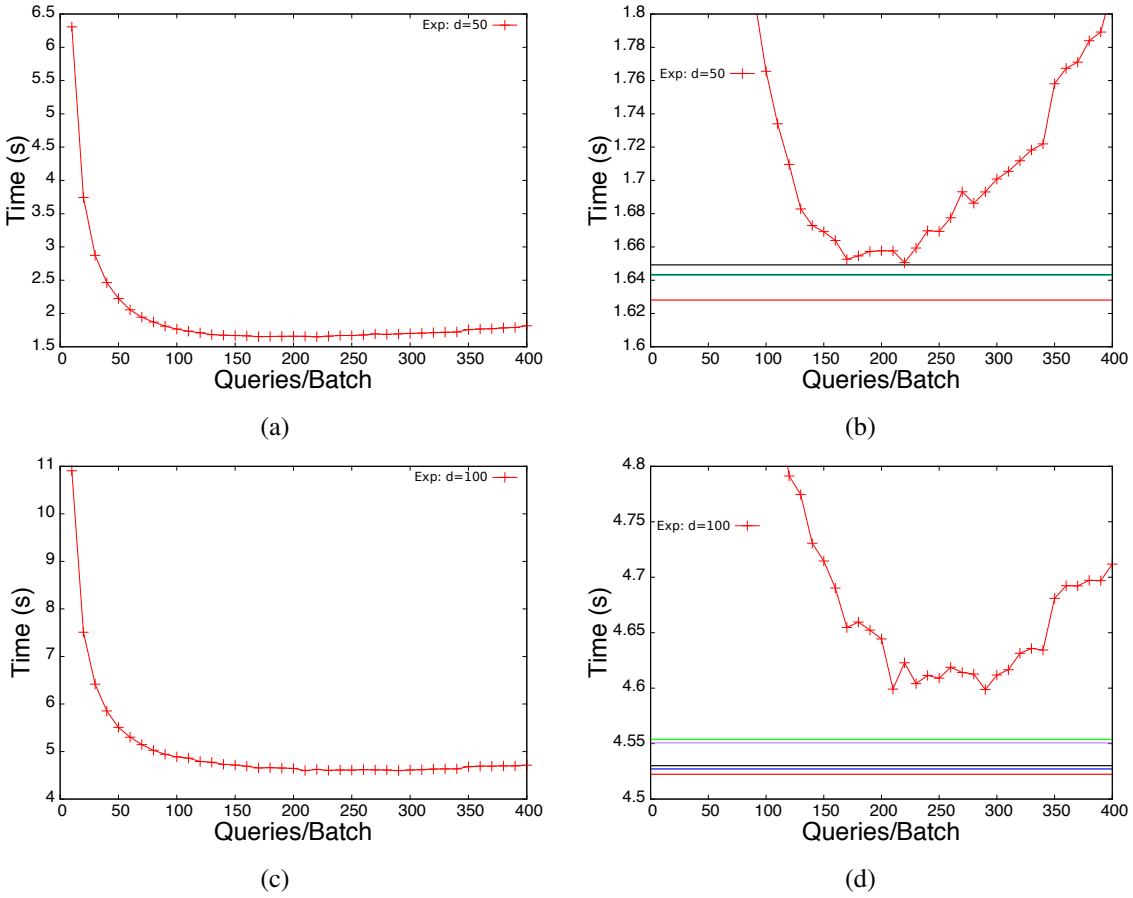


Figure A.4. Response time vs. queries/batch (s) for the periodic query batch method for S9 (a) and S10 (c) (RANDWALK-EXP dataset). Panels (b) and (d) correspond to zoomed in versions of (a) and (c) respectively, to highlight the minimum response times. The colored lines correspond to the same algorithms as shown in Figure 5.9.

Appendix B

Calculation of Moving Distance

In what follows, we illustrate the moving distance calculation. We assume that the speed is constant between the two points that define the spatiotemporal line segment. Let x_i^t, y_i^t, and z_i^t denote the coordinates of a point on spatiotemporal line segment l_i at time t, where t is normalized in the range [0,1]. Consider two line segments l_1 and l_2, with $l_1 = \{(x_1^0, y_1^0, z_1^0), (x_1^1, y_1^1, z_1^1)\}$, and $l_2 = \{(x_2^0, y_2^0, z_2^0), (x_2^1, y_2^1, z_2^1)\}$. The squared Euclidean distance between l_1 and l_2 at time t is computed as:

$$D_t^2 = (x_t^1 - x_t^2)^2 + (y_t^1 - y_t^2)^2 + (z_t^1 - z_t^2)^2, \tag{B.0.1}$$

where,

$$\begin{aligned}
x_1^t &= x_1^0 + t(x_1^1 - x_1^0) \\
y_1^t &= y_1^0 + t(y_1^1 - y_1^0) \\
z_1^t &= z_1^0 + t(z_1^1 - z_1^0) \\
x_2^t &= x_2^0 + t(x_2^1 - x_2^0) \\
y_2^t &= y_2^0 + t(y_2^1 - y_2^0) \\
z_2^t &= z_2^0 + t(z_2^1 - z_2^0).
\end{aligned}$$

$$
\begin{aligned}
D_2^t = \quad & ((x_1^0 + t(x_1^1 - x_1^0)) - (x_2^0 + t(x_2^1 - x_2^0)))^2 + ((y_1^0 + t(y_1^1 - y_1^0)) - (y_2^0 + t(y_2^1 - y_2^0)))^2 \\
& + ((z_1^0 + t(z_1^1 - z_1^0)) - (z_2^0 + t(z_2^1 - z_2^0)))^2 \\
= \quad & ((x_1^0 - x_2^0) + t(x_1^1 - x_1^0 - x_2^1 + x_2^0))^2 + ((y_1^0 - y_2^0) + t(y_1^1 - y_1^0 - y_2^1 + y_2^0))^2 \\
& + ((z_1^0 - z_2^0) + t(z_1^1 - z_1^0 - z_2^1 + z_2^0))^2 \\
= \quad & (x_1^0 - x_2^0)^2 + t^2(x_1^1 - x_1^0 - x_2^1 + x_2^0)^2 + 2t(x_1^0 - x_2^0)(x_1^1 - x_1^0 - x_2^1 + x_2^0) \\
& + (y_1^0 - y_2^0)^2 + t^2(y_1^1 - y_1^0 - y_2^1 + y_2^0)^2 + 2t(y_1^0 - y_2^0)(y_1^1 - y_1^0 - y_2^1 + y_2^0) \\
& + (z_1^0 - z_2^0)^2 + t^2(z_1^1 - z_1^0 - z_2^1 + z_2^0)^2 + 2t(z_1^0 - z_2^0)(z_1^1 - z_1^0 - z_2^1 + z_2^0)
\end{aligned}
$$

Let,

$$
\begin{aligned}
A_x &= (x_1^1 - x_1^0 - x_2^1 + x_2^0)^2 \\
A_y &= (y_1^1 - y_1^0 - y_2^1 + y_2^0)^2 \\
A_z &= (z_1^1 - z_1^0 - z_2^1 + z_2^0)^2 \\
B_x &= 2(x_1^0 - x_2^0)(x_1^1 - x_1^0 - x_2^1 + x_2^0) \\
B_y &= 2(y_1^0 - y_2^0)(y_1^1 - y_1^0 - y_2^1 + y_2^0) \\
B_z &= 2(z_1^0 - z_2^0)(z_1^1 - z_1^0 - z_2^1 + z_2^0) \\
C_x &= (x_1^0 - x_2^0)^2 \\
C_y &= (y_1^0 - y_2^0)^2 \\
C_z &= (z_1^0 - z_2^0)^2 \\
A &= A_x + A_y + A_z \\
B &= B_x + B_y + B_z \\
C &= C_x + C_y + C_z
\end{aligned}
$$

Thus,

$$
\begin{aligned}
D_t^2 &= A_x t^2 + B_x t + C_x + A_y t^2 + B_y t + C_y + A_z t^2 + B_z t + C_z \\
&= At^2 + Bt + C. \qquad \text{(B.0.2)}
\end{aligned}
$$

Modifying Equation B.0.2, calculating the distance within the threshold distance d amounts to solving the following quadratic equation for t:

$$
D_t^2 - d^2 = At^2 + Bt + E, \qquad \text{(B.0.3)}
$$

where $E = C - d^2$.

Let Δ be the discriminant of the quadratic equation:

$$
\Delta = B^2 - 4AE. \qquad \text{(B.0.4)}
$$

if $\Delta < 0$, then the line segments are not within the threshold distance d

if $\Delta = 0$, then the line segments are within the threshold distance d at time $t = \frac{-B}{2A}$

if $\Delta > 0$, then the line segments are within the threshold distance d within the time interval:

$$
\left[\frac{-B - \sqrt{\Delta}}{2A}, \frac{-B + \sqrt{\Delta}}{2A} \right] \cap [0, 1]
$$

Appendix C

Publications

- Gowanlock, M. & Casanova, H. Indexing of Spatiotemporal Trajectories for Efficient Distance Threshold Similarity Searches on the GPU. To appear in the proceedings of the *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2015)*. (Acceptance rate: 21.8%)

- Gowanlock, M. & Casanova, H. Efficient Indexing and Processing of Trajectory Similarity Searches for Two Memory Constraint Scenarios on the GPU. Proceedings of the *Student Research Symposium of the 21st annual IEEE International Conference on High Performance Computing (HiPC 2014)*, Goa, India, December, 2014. (Acceptance rate: 24%)

- Gowanlock, M. & Casanova, H. Distance Threshold Similarity Searches on Spatiotemporal Trajectories using GPGPU. Proceedings of the *21st annual IEEE International Conference on High Performance Computing (HiPC 2014)*, Goa, India, December, 2014. (Acceptance rate: 23%)

- Gowanlock, M. & Casanova, H. (2014) In-Memory Distance Threshold Queries on Moving Object Trajectories. Proceedings of the *Sixth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, Chamonix, France, April, 2014, pp. 41-50. (Acceptance rate: 30%)

- Gowanlock, M., Casanova, H. & Schanzenbach, D. (2014) Parallel In-Memory Distance Threshold Queries on Trajectory Databases. Proceedings of the *Sixth International Confer-*

ence on Advances in Databases, Knowledge, and Data Applications (DBKDA), Chamonix, France, April, 2014, pp. 80-83. (Acceptance rate: 30%)

- Gowanlock, M. & Gazan, R. (2013) Assessing Researcher Interdisciplinarity: A Case Study of the University of Hawaii NASA Astrobiology Institute. *Scientometrics* 94:133-161.

- Gowanlock, M. G., Patton, D. R., & McConnell, S. M. (2011) A Model of Habitability Within the Milky Way Galaxy. *Astrobiology* 11(9):855-873.

Bibliography

[1] `http://www.chorochronos.org/`. Accessed 5-February-2014.

[2] `http://www.superliminal.com/sources/sources.htm`. Accessed 5-February-2014.

[3] Indexing of moving objects for location-based services. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 463–, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. *J. Comput. Syst. Sci.*, 66(1):207–243, February 2003.

[5] S. Arumugam and C. Jermaine. Closest-Point-of-Approach Join for Moving Object Histories. In *Proc. of the 22nd Intl. Conf. on Data Engineering*, pages 86–95, 2006.

[6] E. F. Bell, D. B. Zucker, V. Belokurov, S. Sharma, K. V. Johnston, J. S. Bullock, D. W. Hogg, K. Jahnke, J. T. A. de Jong, T. C. Beers, N. W. Evans, E. K. Grebel, Ž. Ivezić, S. E. Koposov, H.-W. Rix, D. P. Schneider, M. Steinmetz, and A. Zolotov. The Accretion Origin of the Milky Way's Stellar Halo. *Astrophysical Journal*, 680:295–311, June 2008.

[7] Rimantas Benetis, S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[8] Elisa Bertino, Barbara Catania, and Boris Shidlovsky. Towards Optimal Indexing for Segment Databases. In *Proc. of the 6th Intl. Conf. on Advances in Database Technology*, pages 39–53, 1998.

[9] J. Binney and S. Tremaine. *Galactic Dynamics: Second Edition*. Princeton University Press, 2008.

[10] N. Bissantz and O. Gerhard. Spiral arms, bar shape and bulge microlensing in the Milky Way. *Monthly Notices of the Royal Astronomical Society*, 330:591–608, March 2002.

[11] G. Chabrier. Galactic Stellar and Substellar Initial Mass Function. *Publications of the Astronomical Society of the Pacific*, 115:763–795, July 2003.

[12] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *Proc. of the Conf. on Innovative Data Sys. Research*, pages 164–175, 2003.

[13] 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*, pages 109–120, 2010.

[14] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for Nearest Neighbor Search on Moving Object Trajectories. *Geoinformatica*, 11(2):159–193, 2007.

[15] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–330, 2000.

[16] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Nearest neighbor search on moving object trajectories. In *Proc. of the 9th Intl. Conf. on Advances in Spatial and Temporal Databases*, pages 328–345, 2005.

[17] Yun-Jun Gao, Chun Li, Gen-Cai Chen, Ling Chen, Xian-Ta Jiang, and Chun Chen. Efficient k-Nearest-Neighbor Search Algorithms for Historical Moving Object Trajectories. *J. Comput. Sci. Technol.*, 22(2):232–244, 2007.

[18] Fosca Giannotti, Mirco Nanni, Fabio Pinelli, and Dino Pedreschi. Trajectory pattern mining. In *Proc. of the 13th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 330–339, 2007.

[19] M. G. Gowanlock, D. R. Patton, and S. M. McConnell. A Model of Habitability Within the Milky Way Galaxy. *Astrobiology*, 11:855–873, 2011.

[20] Ralf Hartmut Güting, Thomas Behr, and Jianqiu Xu. Efficient k-nearest neighbor search on moving object trajectories. *The VLDB Journal*, 19(5):687–714, 2010.

[21] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.

[22] Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient indexing of spatiotemporal objects. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '02, pages 251–268, London, UK, UK, 2002. Springer-Verlag.

[23] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, pages 3:1–3:8, 2011.

[24] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *Proc. VLDB Endow.*, 1(1):1068–1080, August 2008.

[25] J. J. Jiménez-Torres, B. Pichardo, G. Lake, and A. Segura. Habitability in Different Milky Way Stellar Environments: A Stellar Interaction Dynamical Approach. *Astrobiology*, 13:491–509, 2013.

[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)*, pages 269–273, Dec 2012.

[27] M. Jurić, Ž. Ivezić, A. Brooks, R. H. Lupton, D. Schlegel, D. Finkbeiner, N. Padmanabhan, N. Bond, B. Sesar, C. M. Rockosi, G. R. Knapp, J. E. Gunn, T. Sumi, D. P. Schneider, J. C. Barentine, H. J. Brewington, J. Brinkmann, M. Fukugita, M. Harvanek, S. J. Kleinman, J. Krzesinski, D. Long, E. H. Neilsen, Jr., A. Nitta, S. A. Snedden, and D. G. York. The Milky Way Tomography with SDSS. I. Stellar Number Density Distribution. *Astrophysical Journal*, 673:864–914, February 2008.

[28] Kimikazu Kato and Tikara Hosino. Multi-gpu algorithm for k-nearest neighbor problem. *Concurrency and Computation: Practice and Experience*, 24(1):45–53, 2012.

[29] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 261–272, New York, NY, USA, 1999. ACM.

[30] George Kollios, Vassilis J. Tsotras, Dimitrios Gunopulos, Alex Delis, and Marios Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. on Knowl. and Data Eng.*, 13(5):758–777, September 2001.

[31] P. Kroupa. On the variation of the initial mass function. *Monthly Notices of the Royal Astronomical Society*, 322:231–246, April 2001.

[32] P. Kroupa, C. A. Tout, and G. Gilmore. The distribution of low-mass stars in the Galactic disc. *Monthly Notices of the Royal Astronomical Society*, 262:545–587, June 1993.

[33] Martin Kruliš, Tomáš Skopal, Jakub Lokoč, and Christian Beecks. Combining CPU and GPU architectures for fast similarity search. *Distributed and Parallel Databases*, 30(3–4):179–207, 2012.

[34] C. Leitherer, D. Schaerer, J. D. Goldader, R. M. G. Delgado, C. Robert, D. F. Kune, D. F. de Mello, D. Devost, and T. M. Heckman. Starburst99: Synthesis Models for Galaxies with Active Star Formation. *Astrophysical Journal*, 123:3–40, July 1999.

[35] Zhenhui Li, Ming Ji, Jae-Gil Lee, Lu-An Tang, Yintao Yu, Jiawei Han, and Roland Kays. Movemine: Mining moving object databases. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1203–1206, 2010.

[36] Lijuan Luo, M. D F Wong, and L. Leong. Parallel implementation of r-trees on the gpu. In *Design Automation Conf. (ASP-DAC), 2012 17th Asia and South Pacific*, pages 353–358, Jan 2012.

[37] P. J. McMillan. Mass models of the Milky Way. *Monthly Notices of the Royal Astronomical Society*, 414:2446–2457, July 2011.

[38] G. E. Miller and J. M. Scalo. The initial mass function and stellar birthrate in the solar neighborhood. *Astrophysical Journal*, 41:513–547, November 1979.

[39] Kyriakos Mouratidis, Dimitris Papadias, Spiridon Bakiras, and Yufei Tao. A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors. *IEEE Trans. on Knowl. and Data Eng.*, 17(11):1451–1464, 2005.

[40] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proc. of ACM SIGMOD Intl. Conf. on Danagement of data*, pages 634–645, 2005.

[41] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Proc. of the Department of Defense HPCMP Users Group Conf.*, pages 7–10, 1999.

[42] Mario A. Nascimento and Jefferson R. O. Silva. Towards historical r-trees. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, SAC '98, pages 235–240, New York, NY, USA, 1998. ACM.

[43] J. F. Navarro, C. S. Frenk, and S. D. M. White. The Structure of Cold Dark Matter Halos. *Astrophysical Journal*, 462:563, May 1996.

[44] Jinfeng Ni and Chinya V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Trans. on Knowl. and Data Eng.*, 19(5):663–678, May 2007.

[45] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. of the 12th Symp. on Principles of Database Sys.*, pages 214–221, 1993.

[46] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation. In *Proc. of the 19th ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, pages 211–220, 2011.

[47] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches in Query Proc. for Moving Object Trajectories. In *Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 395–406, 2000.

[48] Kriengkrai Porkaew, Iosif Lazaridis, and Sharad Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, SSTD '01, pages 59–78, London, UK, UK, 2001. Springer-Verlag.

[49] Shaojie Qiao, Changjie Tang, Shucheng Dai, Mingfang Zhu, Jing Peng, Hongjun Li, and Yungchang Ku. Partspan: Parallel sequence mining of trajectory patterns. In *Fifth Intl. Conf. on Fuzzy Systems and Knowledge Discovery*, volume 5, pages 363–367, Oct 2008.

[50] Katerina Raptopoulou, Michael Vassilakopoulos, and Yannis Manolopoulos. On past-time indexing of moving objects. *J. Syst. Softw.*, 79(8):1079–1091, August 2006.

[51] Slobodan Rasetic, Jörg Sander, James Elding, and Mario A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 934–945, 2005.

[52] I. N. Reid, J. E. Gizis, and S. L. Hawley. The Palomar/MSU Nearby Star Spectroscopic Survey. IV. The Luminosity Function in the Solar Neighborhood and M Dwarf Kinematics. *Astronomical Journal*, 124:2721–2738, November 2002.

[53] I. N. Reid, J. D. Kirkpatrick, J. Liebert, A. Burrows, J. E. Gizis, A. Burgasser, C. C. Dahn, D. Monet, R. Cutri, C. A. Beichman, and M. Skrutskie. L Dwarfs and the Substellar Mass Function. *Astrophysical Journal*, 521:613–629, August 1999.

[54] S. A. Rodionov and N. Y. Sotnikova. Optimal Choice of the Softening Length and Time Step in N-body Simulations. *Astronomy Reports*, 49:470–476, June 2005.

[55] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 71–79, 1995.

[56] E. E. Salpeter. The Luminosity Function and Stellar Evolution. *Astrophysical Journal*, 121:161, January 1955.

[57] Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. of the 7th Intl. Symp. on Advances in Spatial and Temporal Databases*, pages 79–96, 2001.

[58] Yufei Tao and Dimitris Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[59] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 287–298, 2002.

[60] Y. Theodoridis, T. Sellis, AN. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, pages 123–132, Jul 1998.

[61] 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*, pages 441–448, 1996.

[62] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. of the 6th Intl. Symp. on Advances in Spatial Databases*, pages 147–164, 1999.

[63] B. C. Thomas, A. L. Melott, C. H. Jackman, C. M. Laird, M. V. Medvedev, R. S. Stolarski, N. Gehrels, J. K. Cannizzo, D. P. Hogan, and L. M. Ejzak. Gamma-Ray Bursts and the Earth: Exploration of Atmospheric, Biological, Climatic, and Biogeochemical Effects. *Astrophysical Journal*, 634:509–533, 2005.

[64] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *Comput. J.*, 43(4):325–343, December 2000.

[65] Marcos R. Vieira, Petko Bakalov, and Vassilis 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*, pages 286–295, 2009.

[66] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29(2):331–342, May 2000.

[67] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid 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*, pages 643–654, 2005.

[68] Simin You, Jianting Zhang, and Le Gruenwald. Parallel spatial query processing on gpus using r-trees. In *Proc. of the 2nd ACM SIGSPATIAL Intl. Workshop on Analytics for Big Geospatial Data*, BigSpatial '13, pages 23–31, New York, NY, USA, 2013. ACM.

[69] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. Monitoring k-Nearest Neighbor Queries over Moving Objects. In *Proc. of the 21st Intl. Conf. on Data Engineering*, pages 631–642, 2005.

[70] Demetrios Zeinalipour-Yazti, Song Lin, and Dimitrios Gunopulos. Distributed spatio-temporal similarity search. In *Proc. of the 15th ACM Intl. Conf. on Information and Knowledge Management*, pages 14–23. ACM, 2006.

[71] Jianting Zhang, Simin You, and Le Gruenwald. U^{2}STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. In *Proc. of the 2012 ACM Workshop on City Data Management Workshop*, CDMW '12, pages 5–12, 2012.

[72] Jianting Zhang, Simin You, and Le Gruenwald. Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. *Information Systems*, 44(0):134–154, 2014.

[73] M. Zoccali, S. Cassisi, J. A. Frogel, A. Gould, S. Ortolani, A. Renzini, R. M. Rich, and A. W. Stephens. The Initial Mass Function of the Galactic Bulge down to $\sim$0.15 M_{solar}. *Astrophysical Journal*, 530:418–428, February 2000.