Parallel In-Memory Distance Threshold Queries on Trajectory Databases

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

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

***Abstract*—Spatiotemporal databases are utilized in many applications to store the trajectories of moving objects. In this context, we focus on in-memory *distance threshold queries* that return all trajectories found within a distance d of a fixed or moving object over a time interval. We present performance results for a sequential query processing algorithm that uses an in-memory R-tree index, and we find that decreasing index resolution improves query response time. We then develop a simple multithreaded implementation and find that high parallel efficiency (78%-90%) can be achieved in a shared memory environment for a set of queries on a real-world dataset. Finally, we show that a GPGPU approach can achieve a speedup over 3.3 when compared to the multithreaded implementation. This speedup is obtained by abandoning the use of an index-tree altogether. This is an interesting result since index-trees have been the cornerstone of efficiently processing spatiotemporal queries.**

Keywords-spatiotemporal databases; query parallelization.

I. INTRODUCTION

Many applications require analyzing moving object trajectories (e.g., users with Global Positioning System devices, animals in ecological studies, stellar bodies in astrophysical simulations). Two relevant queries over moving object trajectories, which we term *distance threshold queries*, are: (i) Find all trajectories within a distance d of a given fixed point over a time interval $[t_0, t_1]$; and (ii) Find all trajectories within a distance d of a given trajectory over a time interval $[t_0, t_1]$.

For instance, consider an astrobiology application that studies the habitability of the Milky Way [1]. The Milky Way is expected to host many rocky low-mass planets, some of which may be able to support complex life. The dangers to complex life include transient radiation events, such as supernovae, or close encounters with flyby stars. To model habitability one must quantify these events, which can be formulated as distance threshold queries: (i) Find all stars within a distance d of a supernova explosion, modeled as a fixed point, over a short time interval t; and (ii) Find the stars, and corresponding time periods, that host a habitable planet and are within a distance d of a moving star, s, over the star's lifetime t_s. Given that a dataset of the Milky Way may contain billions of stellar trajectories, distance threshold queries must be performed efficiently.

Our objective is to design efficient distance threshold query processing algorithms, and we make the following contributions:

1) We propose a sequential algorithm that relies on an efficient trajectory indexing strategy.
2) We investigate parallelization in a multi-proc/multi-core environment and a General-Purpose computing on Graphics Processing Units (GPGPU) environment. The contrasting architectures require different algorithms and data structures to achieve good performance.
3) We outline performance bottlenecks and suggest methods for their resolution in a distributed memory environment.

II. RELATED WORK

A trajectory is a set of points traversed by an object over time. Linear interpolation is used when processing queries that fall in between the points (i.e., one assumes that the points are connected by line segments). Most works on moving object databases propose sequential query processing algorithms that utilize an R-tree index [2] or variations of it (e.g., TB-trees [3], STR-trees [3], 3DR-trees [4], and SETI [5]). The R-tree indexes spatiotemporal data using hyperrectangular minimum bounding boxes (MBBs), where each line segment belonging to a trajectory is stored in one MBB at a leaf node. The non-leaf nodes contain the dimensions of the MBB that contains all MBBs in its sub-tree. Given a query MBB, an index search returns all leaf node MBBs that overlap the query MBB.

Except in [6], distance threshold queries have received little attention. The most related queries are k Nearest Neighbors (kNN) queries [7], [8], [9], [10]. In some sense a distance threshold query is a kNN query with an unknown value of k, since there is no a-priori limit to the number of query matches. Therefore, kNN query processing algorithms cannot be applied to process distance threshold queries.

III. SEQUENTIAL IMPLEMENTATION

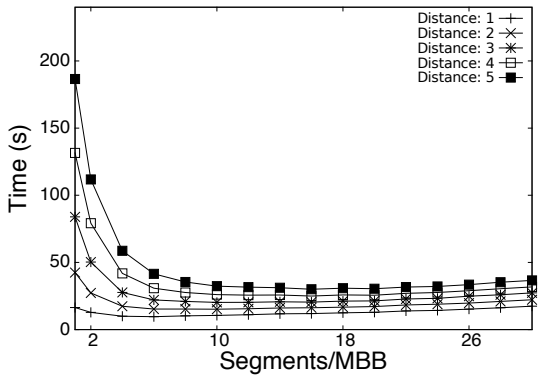
In this section we evaluate a sequential implementation of the TRAJDISTSEARCH distance threshold query processing algorithm that we proposed in [11]. Given a query trajectory line segment over a temporal extent, a query MBB is computed based on the query threshold distance. TRAJDISTSEARCH then searches a trajectory index for MBBs that overlap the query MBB. TRAJDISTSEARCH is implemented in C++, re-using an R-tree index implementation based on that initially developed by A. Guttman [2] with source code available at [12].

 ISBN: 978-1-61208-334-6

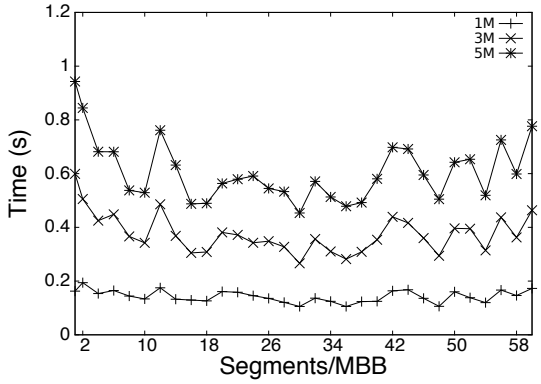
We run TRAJDISTSEARCH on one core of a dedicated 3.46 Ghz Intel Xeon W3690 processor with 12 MB L3 cache and sufficient memory to store the entire index. We average query response time over 3 trials. The variation among the trials is negligible so that error bars in our results are not visible. We ignore the overhead of loading the R-tree from disk into memory, which can be done once before all query processing. We measure the response time of TRAJDISTSEARCH for the following datasets and queries, which are available at [13]:

- S1: A 4-D (3 spatial + 1 temporal) dataset, *Galaxy*, containing star trajectories (for the application described in Section I), with 1,000,000 trajectory segments corresponding to 2,500 trajectories of 400 timesteps each. The query consists of 100 trajectories for 100% of their temporal extent, with a variable query distance d.
- S2: Three 4-D synthetic datasets, *Random*, with trajectories generated via random walks, with ∼1,000,000 (1M), ∼3,000,000 (3M) and ∼5,000,000 (5M) line segments corresponding to 2500, 7500 and 12500 trajectories, respectively. The query consists of 100 trajectories for 100% of their temporal extent, with a fixed query distance, $d = 15$.

One indexing approach is to assign each trajectory segment to its own MBB, minimizing index overlap, but maximizing the number of entries in the index. By assigning multiple trajectory segments to an MBB, index traversal time is decreased as the index contains fewer entries. However, a larger number of candidate segments is returned, many of which may not overlap the query MBB, leading to higher segment processing times. To explore the effect of varying the index resolution, for each trajectory we place its first (temporally) r segments in an MBB, its next r segments in another MBB, and so on. $r = 1$ corresponds to using a single MBB per trajectory segment. Figure 1 plots response time vs. r for S1 and S2. A small r value can lead to high response times. In Figure 1 (a) with a value of $r = 12$, the response time with $d = 5$ is 31.6 s in comparison to 186.5 s for $r = 1$, or a factor of 5.9 faster. Grouping multiple line segments into MBBs in this manner ensures that line segments of a trajectory are temporally contiguous. We also attempted to split trajectories so as to minimize MBB volumes using the *MergeSplit* algorithm [14], but saw no performance improvement (see full details in [11]).



(a) *Galaxy*



(b) *Random* datasets

Figure 1. Response time vs. the r for (a) S1 for the *Galaxy* dataset for various query distances; and (b) S2 for the *Random*-1M, 3M, and 5M datasets and a query distance of 15.

IV. SHARED-MEMORY PARALLEL IMPLEMENTATION

A. Multi-core OpenMP

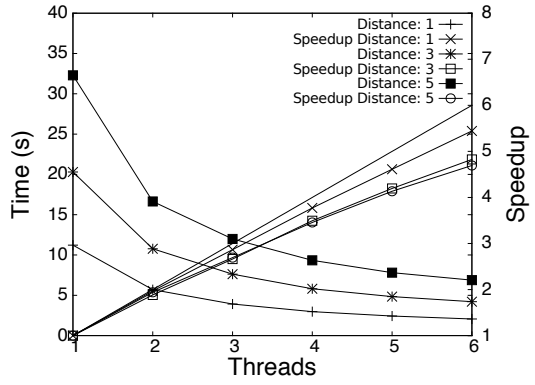
TRAJDISTSEARCH can be easily parallelized using OpenMP in a shared-memory setting because iterations of its outer loop are independent. Figure 2 shows the response time on the 6-core platform described in the previous section vs. the number of threads for S1 and S2 with $r = 12$ and $r = 10$, respectively (i.e., the "best" r values for the sequential implementation). We see high parallel efficiency (78%-90%), with parallel speedup between 4.69 and 5.44 with 6 threads for query distances ranging from $d = 1$ to $d = 5$.

B. GPGPU with OpenCL

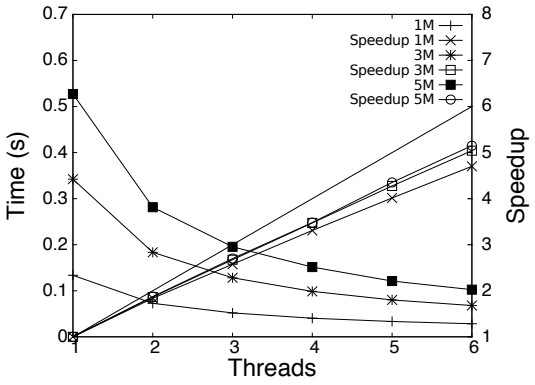
In TRAJDISTSEARCH, the R-tree is used to reduce the number of line segments that must be processed. Unfortunately, the index-tree traversal is memory-bound with non-deterministic execution paths due to branches, making efficient execution on a GPU challenging. As a result, we completely forego the use of the index-tree entirely so as to exploit the massive parallelism of the GPU. In the GPU version of TRAJDISTSEARCH, all line segments of each trajectory in the dataset are stored in the GPU's global memory once and for all before all query processing. These line segments are sorted temporally based on their start times (line segments of a trajectory may not be stored contiguously). On the host, for a specified number of bins, B, we bin these line segments, where each bin is defined by a range of start times and consists of the indices of the first and last segments in the bin.

1) Constant Sized Query Batches: We develop a GPU kernel that processes a set Q of N query line segments, initially stored on the host. The host first sorts the line segments in Q by their start times, and determines the relevant contiguous bins that contain entry line segments that may overlap temporally with at least one query line segment. Let E denote the entry line segment index range corresponding to this set of bins. Larger values of B (i.e., smaller bins) mean a more expensive computation for E but a more precise value for E (i.e., a smaller range). Our GPU kernel takes as input E and Q, where each GPU thread is responsible for one line segment in E, and computes whether any of the segments in Q are within distance d of that line segment. This brute-force search returns relevant time intervals annotated with the IDs of the trajectories that are within the query distance. This kernel can be invoked multiple times to overlap communication and computation.

 ISBN: 978-1-61208-334-6

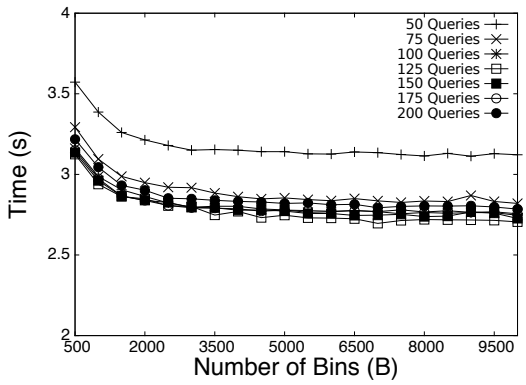


(a) *Galaxy*

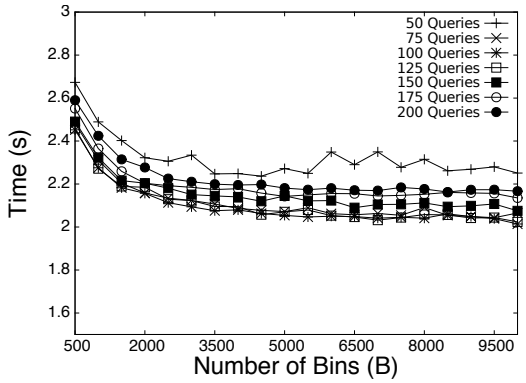


(b) *Random* datasets

Figure 2. Response time vs. number of threads (a) S1 for the *Galaxy* dataset for various query distances and $r = 12$; and (b) S2 for the *Random*-1M, 3M, and 5M datasets, with a query distance of 15 and $r = 10$.



(a)



(b)

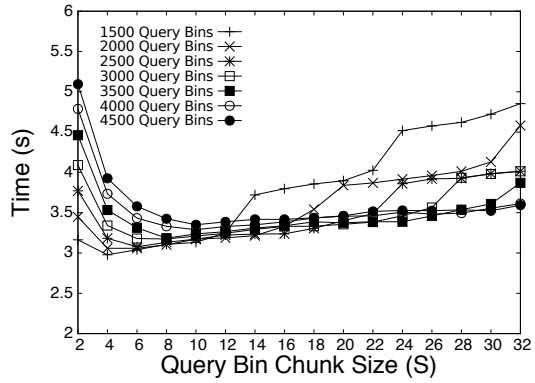
Figure 3. Response time vs. B for S1 with various N values and $d = 5$; (a) 1 work queue and kernel instance; (b) 3 work queues and kernel instances.

We implemented the above kernel in OpenCL and executed it on the platform described in Section III equipped with an Nvidia Tesla C2075 GPU device. Figure 3 (a) plots response time vs. B for dataset/query S1 and for a query distance $d = 5$, and for various values of N. For the results in this figure we only consider one workqueue with a single kernel (1 CPU thread), so that there is no overlap of computation and communication. We find that a value of $N = 125$ leads to the best performance, independently of the number of bins. We also see that too small a number of bins leads to high response time because the index range E is unnecessarily large. The response time plateaus around $B = 5000$. Figure 3 (b) shows similar results but with 3 workqueues each running an instance of the kernel, thus allowing overlap of computation and communication (using more workqueues leads to no further improvements in our experiments). We see the same trends in terms of the number of bins B. Using $N = 100$ or $N = 125$ leads to the lowest response time overall. The performance gain due to overlap is significant. For instance, with $B = 5000$ and $N = 125$, the response time in Figure 3(a) is 2.75s while that in Figure 3(b) is 2.07s, or 24.7% faster.

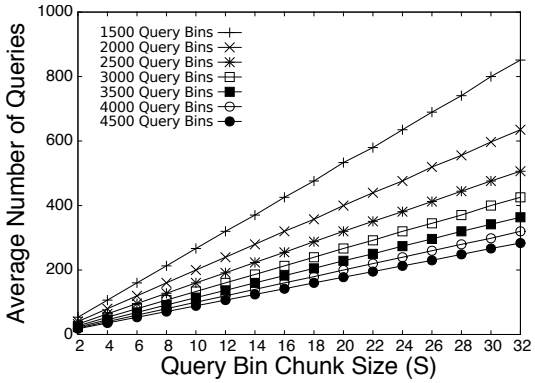
In comparison to the initial sequential implementation, still for dataset/query S1, using $r = 1$ (Figure 1 (a)) for $d = 5$, the GPU implementation using a single kernel (Figure 3 (a)) gives a speedup of over 67. Considering the best value of $r = 12$ for S1, and the multithreaded CPU implementation with 6 threads, we obtain a speedup using the GPU of over 2.5 with a single kernel, and a speedup of over 3.3 when using 3 workqueues.

2) Variable Sized Query Batches: One drawback of constant sized query batches is that, due to temporal properties of the dataset, the batch may temporally overlap with a large entry index range E but each individual query may overlap only a small subset of E, thus leading to wasteful computations. We propose here an approach that uses variable sized batches to reduce the number of these wasteful calculations. We group queries according to their temporal properties by sorting them temporally, as in the previous approach, but then binning them in the same manner as the entries described in Section IV-B. More precisely, each query line segment is assigned to one of C query bins. Each of the C bins is mapped onto the indices of the B entry bins for which the C bins overlap temporally. We construct query batches as contiguous sets of S query bins (batches contain different numbers of queries), which are sent to the GPU for each kernel invocation.

Figure 4 (a) plots response time vs. S for dataset/query S1 with $d = 5$. To compare with Figure 3 (a), we have plotted the average number of queries per kernel execution in Figure 4 (b). We observe that this approach performs slightly worse than the constant sized query batch approach (Figure 3 (a)). We attribute this to two factors: (i) the additional overhead required to construct the C query bins, and (ii) the fact that this particular dataset has roughly the same number of active trajectory segments at any given time. Elaborating on (ii), if the dataset contained short punctuated time periods where there are many relevant trajectories, and other periods with few relevant trajectories, then the variable sized query



(a)



(b)

Figure 4. (a) Response time vs. S for S1 with various C values, $d = 5$, $B = 7500$, with 1 work queue and kernel instance; (b) the average number of queries per kernel execution vs. S for the results in the upper panel.

batch approach would likely outperform the constant sized query batch approach, since on average each kernel instance would be given a smaller E range. One such application would be the trajectories of vehicles, where the number of active trajectories at a given time, unlike in the case of stars in the galaxy, is influenced by human activities, such as rush hour, daytime, nighttime, etc.

V. Discussion and Conclusions

This work in progress studies in-memory distance threshold queries for moving object trajectory databases. For a sequential implementation, a natural and effective approach is to use an index tree and to group multiple trajectory line segments into MBBs. We have shown that the resulting algorithm can be parallelized efficiently on a multi-core platform. A future work direction is to consider distributed memory environments with multiple multi-core nodes. The global index could be partitioned across the nodes, with however a high risk of load imbalance that would lead to deeper tree traversals for some of the nodes. In this context, it would be interesting to study the impact of index resolution on load balancing, possibly leading to a solution that uses different index resolutions on different nodes.

We have shown that GPGPU can lead to efficient distance threshold query processing provided queries are processed in batches of appropriate size and the use of the R-tree index is abandoned. For the queries/datasets used in our experimental evaluation we have found that the GPU can provide a speedup of over 3.3 when compared to a multi-threaded R-tree based implementation that uses 6 cores. An important result is that our brute-force GPGPU approach outperforms the traditional search-and-refine strategy that uses the popular R-tree index. Future work in this direction may investigate the use of a GPU implementation of the R-tree [15], and possibly using hybrid approaches for splitting up the execution of the query processing algorithm between the host and the GPU device.

Acknowledgments

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

References

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

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

[3] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Proc. for Moving Object Trajectories," in Proc. of the 26th Intl. Conf. on Very Large Data Bases, 2000, pp. 395–406.

[4] Y. Theodoridis, M. Vazirgiannis, and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," in Proc. of the Intl. Conf. on Multimedia Computing and Systems, 1996, pp. 441–448.

[5] 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, 2003, pp. 164–175.

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

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

[8] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Algorithms for Nearest Neighbor Search on Moving Object Trajectories," Geoinformatica, vol. 11, no. 2, 2007, pp. 159–193.

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

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

[11] M. Gowanlock and H. Casanova, "In-Memory Distance Threshold Queries on Moving Object Trajectories," in Proc. of the Sixth Intl. Conf. on Advances in Databases, Knowledge, and Data Applications, 2014.

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

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

[14] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos, "Efficient indexing of spatiotemporal objects," in Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology, 2002, pp. 251–268.

[15] L. Luo, M. D. F. Wong, and L. Leong, "Parallel implementation of R-trees on the GPU," in ASP-DAC, 2012, pp. 353–358.

 ISBN: 978-1-61208-334-6