P^2EST: Parallelization Philosophies for Evaluating Spatio-Temporal Queries

Xiling Sun[*] Anan Yaagoub Goce Trajcevski[†]

Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Il 60208
x-sun,anany,goce@eecs.northwestern.edu

Peter Scheuermann[*] Hao Chen Abhinav Kachhwaha

Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Il 60208
peters,h-chen,abhinav@eecs.northwestern.edu

ABSTRACT

This work considers the impact of different contexts when attempting to exploit parallelization approaches for processing continuous spatio-temporal queries. More specifically, we are interested in various trade-off aspects that may arise due to differences of the computing environments like, for example, multicore vs. cloud. Algorithmic solutions for parallel processing of spatio-temporal queries cater to splitting the load among units - be it based on the data or the query (or both) - relying to a bigger or lesser degree on a certain set of features of a given environment. We postulate that incorporating the service-features should be coupled with the algorithms/heuristics for processing particular queries, in addition to the volume of the data. We present the current version of the implementation of our P^2EST system and analyze the execution of different heuristics for parallel processing of spatio-temporal range queries.

Categories and Subject Descriptors

H.2.8 [**Spatial Databases and GIS**]: Miscellaneous

General Terms

Algorithms, Performance

[*]Research supported in part by the National Science Foundation under grants CNS 0910952.

[†]Research supported in part by the National Science Foundation under grants CNS 0910952 and III 1213038.

Keywords

Spatio-temporal queries, Multi-core, Cloud

1. INTRODUCTION

The goals of Moving Objects Databases (MOD) are centered around two main themes: (1) efficient storage and retrieval of the spatio-temporal data representing the motion of a large number of moving objects; and (2) efficient processing of various queries of interest, such as, whereabouts-in-time, range, (k)Nearest-neighbor, similarity, skyline, etc... [9]. In addition to the properties that are consequence of the type of a particular query, many context dimensions have been used to improve the efficiency of the query processing have been by capitalized upon. Some examples are: historic trajectories vs. streaming moving objects data [10, 12, 20, 30]; "crisp" vs. uncertainty-aware settings [18, 19, 27, 26, 25]; road network vs. free motion [4, 21].

More specifically, some works have investigated the correlation of architectural/environmental aspects with the data properties, for the purpose of generating efficient algorithms for processing spation-temporal queries. For example, the load-shedding introduced in [20] introduces structures to drop moving objects from memory once they become insignificant. Complementary to this, [7] proposes distributed processing of spatio-temporal queries by delegating some of the updates-responsibilities to the (distributed but collaborative) mobile entities themselves.

In this particular work, we take a step towards incorporating the context of *processing platforms* when selecting approaches for efficient evaluation of spatio-temporal queries. Since many GIS application need effective and scalable techniques for processing such queries and parallelization is a good approach, we investigated two "ends of the spectrum":

1. On one hand, we consider efficient execution of algorithms in multicore settings [11] for the purpose of

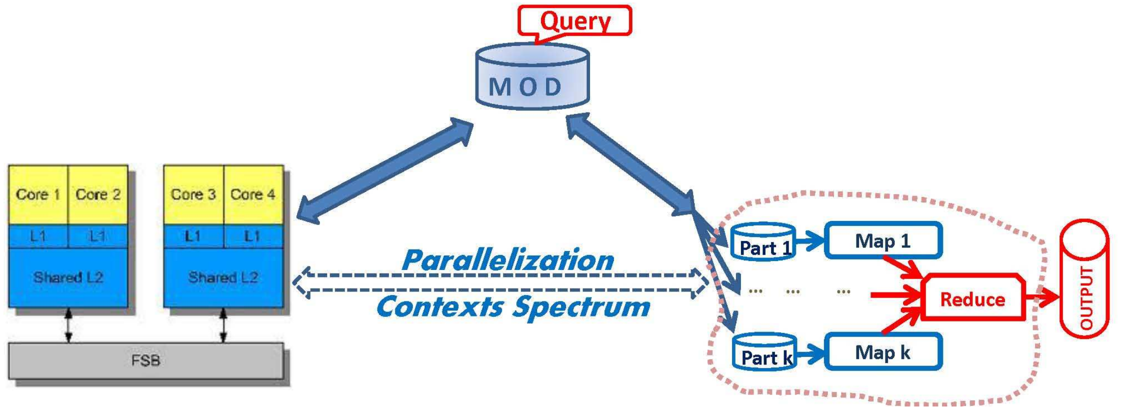


Figure 1: Different Platforms for Parallel Processing of Spatio-Temporal Queries

maximizing the benefit of locally-parallelizing various tasks involved in generating queries' answers.

2. On the other hand, we incorporate the style of parallel programming supported by capacity-on-demand distributed cloud environments [8]. Only recently has the cloud-based paradigm been applied to geo-spatial queries [3].

Cloud computing has become a popular technique to process large-scale dataset [13], in large due to its elasticity provision and economic benefits (i.e., avoiding a purchase of own servers for processing large data). [15]. Applications are delivered as services come from the traditional software delivery model called Software as a Service (SaaS) [1]. From a global perspective, part of our motivation comes from the arguments presented in [24] – except we took the complementary stand of focusing on spatio-temporal queries and the impact of a particular paradigm/platform.

Clearly, incorporating the semantics of the underlying data and/or queries will (almost) always yield considerable advantages for exploiting the benefits of a particular platform. However, the drive behind our work stems from the observation that "blind incorporations" need not yield same (or similar) benefits in different environments.

In [28] we proposed several partitioning and cooperative load-balancing strategies for processing a collection of spatio-temporal range queries in multicore environments. Our experimental observations indicated a certain degree of differences in terms of execution time among the proposed heuristics, thereby implying a ranking among them. What we asked ourselves were the following two basic questions:

1. *If we execute the same heuristic in a different parallelization environment,* (e.g., Amazon Web Services (AWS) cloud[1] *would the relative ranking be the same?*

2. *Is there any data size which would determine when one environment should be preferred to the other?*

[1]http://aws.amazon.com/

The intuition of our motivation is illustrated in Figure 1. We view the multicore systems and the cloud as two "ends of spectrum", and aim at providing a tool that would let the users seamlessly test their different implementations on different platforms.

In the rest of this paper, after collecting some preliminary background in Section 2, we discuss the parallelization heuristics in Section 3, along with the files' structures and the algorithms used in processing the spatio-temporal queries in AWS. Section 4 describes our current system prototype and presents some experimental observations. We conclude the paper and outline directions for future work in Section 5.

2. PRELIMINARIES

We now review the basic concepts and notation used throughout the rest of this paper.

Typically, in MOD settings, a *trajectory* Tr of a moving object, is a polyline in a 3D space (2D spatial + time), represented as a sequence of points $Tr = (x_1, y_1, t_1), \ldots, (x_n, y_n, t_n)$, where $\forall (i,j)(i < j \Rightarrow t_i < t_j)$. Between two consecutive points (x_i, y_i, t_i) and $(x_{i+1}, y_{i+1}, t_{i+1})$, the object is assumed to move along the straight line-segment $(\overline{(x_i, y_i)(x_{i+1}, y_{i+1})})$, and with a constant *expected speed* $v_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}/(t_{i+1} - t_i)$.
The *expected location* of the object at any time-point t $(\in (t_i, t_{i+1}))$ is the one obtained via linear interpolation between the endpoints, using the expected speed v_i. The projection of Tr_k in the Euclidian 2D space is called its *route*.

We assume multicore settings in a multiple-reader multiple-writer (MRMW) shared memory context [11]. Specifically, each core $\mathbf{C}_i$ can access the different portions of the MOD-data and, when applicable can write in a buffer(s) that can be read by the other cores for the purpose of determining whether a particular thread running on $\mathbf{C}_i$ should be terminated or not, without affecting the correctness of the answer to the query. In addition, we assume a cooperation among the cores in the sense that when a particular core $\mathbf{C}_i$ completes the processing of its assigned load, it can take part of

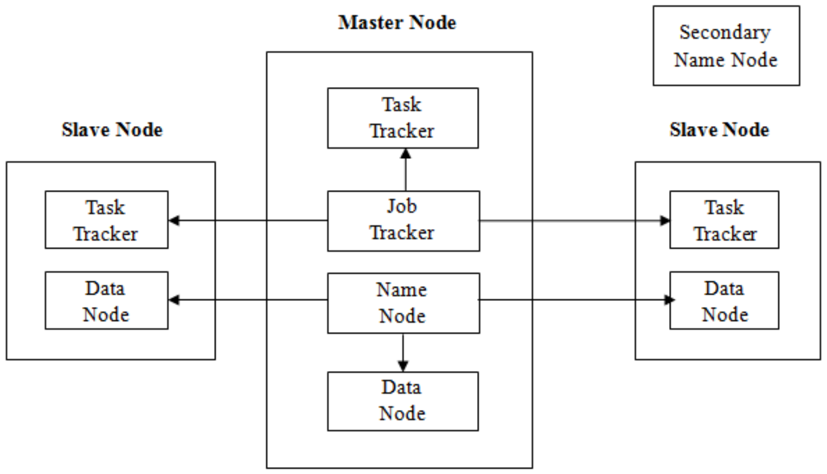


Figure 2: Hadoop Architecture

the current load of another core $\mathbf{C}_j$ (cf. [29]).

Utilizing the cloud for any computation or query processing typically assumes a collaboration among four main categories: (1) the front end platform (i.e. the client); (2) the back end platform (i.e. the server); (3) the delivered service (i.e. software or hardware); and (4) the Internet. The front end platform requests the service from the back end platform via the Internet. Popularized by Google over the last decade, MapReduce is a distributed programming framework used to relief distributed applications developers from the burdens of reliability and scalability [14]. Map function and Reduce function are the two fundamental computing units. Users only need to provide their Map function and Reduce function in order to distribute the processing of large amount of data. Map function processes the input, generates key-value pairs, and then emits the key-value pairs to the intermediate of MapReduce framework. Then MapReduce framework will gather all key-value pairs with the same key to the same Reduce function. Each Reduce function, in turn, processes these key-value pairs and emits the final output [16].

Along these lines, Hadoop is open-source software that supports reliable, scalable, data-intensive distributed computing for large-scale dataset among clusters[2]. Hadoop has three main parts: the Hadoop kernel, the distributed file system HDFS (Hadoop Distributed File System), and the implementation of the basic paradigm, named Hadoop MapReduce. The open source implementation, among others, contains the Google File System [22], MapReduce framework [14] and BigTable [6].

As shown in Figure 2, a typical Hadoop cluster contains one master node and several slave nodes. The master node has four parts: TaskTracker, JobTracker, NameNode, and DataNode. Each slave node has two parts: TaskTracker and DataNode. Each Hadoop cluster also has a SecondaryNameNode which mainly manages the log files of cluster[3]. Data in Hadoop clusters is split into smaller chunks which are distributed among clusters using data block protocol specified by HDFS. HDFS acquires fault-tolerant by replicating the data chunks across multiple machines [10]. Hadoop will get the data chunks from the nearest machine when running MapReduce applications.

There are several open source implementations of Hadoop relying on HDFS for storing the data (e.g., HBase, Hive, Pig) as well as commercial ones (e.g., HDInsight for Microsoft Windows Azure platform). In this work, we used Amazon Web Services (AWS) platform, running our MapReduce program by using Elastic MapReduce and Scalable Storage services.

3. PARTITIONING AND PARALLELIZATION STRATEGIES

We now describe the data structures and partitioning approaches used throughout this work. For presentation purposes, we separate the discussion along the ”environmental” context and we address the multicore settings first, followed by the cloud environment. For conciseness, we use the term ”unit” to denote a core in multicore setting, or a node in AWS cloud-setting, respectively.

We focus on range queries, for which the basic statement is:

Qr: *Retrieve all the moving objects which are inside the region R between* $[t_1, t_2]$.

and, as mentioned in Section 2, when processing such queries, we assume that in-between consecutive updates the location of a moving object is obtained via linear interpolation.

We note that the above example illustrates the, so called, *existential* variant of a range query. However, one can also consider other variants – e.g., retrieving of the objects who were inside R throughout the entire time-interval of interest

[2]http://hadoop.apache.org/
[3]http://hadoop.apache.org/docs/stable/hdfs_user_guide.html

(universal); or retrieving of the objects who were inside R for at least Θ ($0 < \Theta < 1$) fraction of $[t_1, t_2]$.

3.1 Heuristics and Multicore

In multicore settings, we consider a data stored in a MOD, in which each trajectory is represented as a user-defined type consisting of a unique object_ID (oID) and a sequence of *(location, time)* values – e.g.,

$$\{o_{25}, [(x_{25,1}, y_{25,1}, t_{25,1}), \ldots, (x_{25,k}, y_{25,k}, t_{25,k})]\}$$

We consider three heuristics (labelled $\mathbf{H}_1 - H_3$) which we describe in the sequel. We note that different collaborations and load balancing techniques among the units are possible based on a particular variant of the range query, and we presented such details for multicore settings in [28, 29].

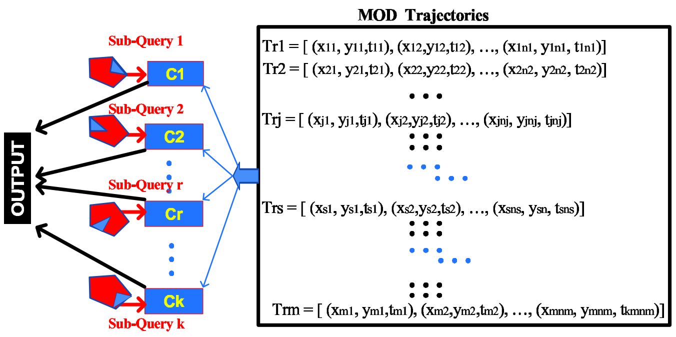

MOD Trajectories
Sub-Query 1
Sub-Query 2
Sub-Query r
Sub-Query k
C1
C2
Cr
Ck
OUTPUT
Tr1 = [(x11, y11,t11), (x12,y12,t12), …, (x1n1, y1n1, t1n1)]
Tr2 = [(x21, y21,t21), (x22,y22,t22), …, (x2n2, y2n2, t2n2)]
Trj = [(xj1, yj1,tj1), (xj2,yj2,tj2), …, (xjnj, yjnj, tjnj)]
Trs = [(xs1, ys1,ts1), (xs2,ys2,ts2), …, (xsns, ysn, tsns)]
Trm = [(xm1, ym1,tm1), (xm2,ym2,tm2), …, (xmnm, ymnm, tkmnm)]

Figure 3: H3: *R*-based Load Distribution Among Cores

- $\mathbf{H}_1$: The first heuristic partitions the MOD "vertically" – i.e., along the temporal dimensions. Essentially, this implies that each of the m units is in charge of checking a corresponding segment of all the trajectories reduced to a time-interval of duration $(t_2 - t_1)/m$. We re-iterate that different variants of the query may offer different (subtle) opportunities for speed-up. Namely, for the existential variant (cf. **Qr:** above), the moment a particular unit detects an intersection/containment, for a given trajectory, the rest of the units can safely ignore their parts of the corresponding trajectory.

- $\mathbf{H}_2$: Complementary to $\mathbf{H}_1$, the second heuristics partitions the MOD data "horizontally". Namely, each unit is assigned a subset of *oID*s encompassing the entire sequence of points for each individual trajectory. It is in charge of processing the range query for each complete trajectory in its subset. Similarly to $\mathbf{H}_1$, one can expect different speed-up possibilities based on the exact semantics of **Qr**. Specifically, for the existential case, a given core can stop considering the rest of the segments in a given trajectory Tr_i after the first intersection is detected. Since each core is in charge of entire trajectory, there is no need to communicate the detection of the satisfiability to the rest of the cores and have them stop processing any portions of Tr_i. It is subtle issues like this that make $\mathbf{H}_2$ exhibit better speed-up than $\mathbf{H}_1$ (cf. [28]).

- $\mathbf{H}_3$: The main difference between this heuristics and the previous two is that the partition is based on the query-space. Namely, for a given collection of regions R_1, …, R_n, each unit is assigned an *equal area sub-region(s)*, and is in charge of processing the range-query over the entire collection of trajectories in the MOD over the corresponding sub-region(s). As illustrated in Figure 3, the main intuition behind H3 is to partition the query-region into k non-overlapping regions – $R_1, R_2, \ldots, R_k$, having only a common boundary-edge as intersection between two consecutive sub-regions. The processing of the sub-query pertaining to R_j is assigned to the core $\mathbf{C}_j$. An individual core $\mathbf{C}_j$ operates over the entire collection of trajectories in the MOD, however, it only evaluates a query pertaining to a sub-region from the original query region R. Assuming that the number of cores in a multicore machine is typically a power-of-2 (e.g., [17], although other multicore architecture exist), we can recursively apply the algorithms for bisecting a given polygon into two (sub)polygons of equal areas [23], so that each core is assigned an equal-area sub-region [4] of R.

3.2 Cloud: Data Representation and Algorithms

Amazon's Elastic MapReduce assumes a default input format to be plain text files, where consecutive lines are separated by $\backslash n$ (newline) character[5]. Given that this is the most commonly used format, we used two basic representation methods:

- *Segment Per Line* (SPL): This format has 8 attributes in each line: *Moid, Tripid, Tstart, Tend, Xstart, Ystart, Xend*, and *Yend*. Essentially, a given trajectory is broken into individual "trip-segments" (i.e., segments between two consecutive time-stamps), and each segment is stored in the separate line.

  The meaning of each attribute is as follows:

  Moid is the ID of a moving object;

  Tripid denotes the ID of a particular trip (in case a given moving object may have a prolonged "stationary" time during the interval of interest).

  Tstart and *Tend* denote the start time and end time for the segment.

  Xstart, Ystart and *Xend, Yend* denote the spatial coordinates of the locations where the segment starts and ends, respectively.

- *Trajectory Per Line* (TPL): This format is closer to the MOD representation of the trajectory, in the sense that all segments are listed in the same line.

  The constituents of a line are: Moid, Tstart_1, Tend_1, Xstart_1, Ystart_1, Xend_1, Yend_1, ... ,Tstart_n, Tend_n, Xstart_n, Ystart_n, Xend_n, and Yend_n – with their intuitive meaning. We note that the TPL format does not have a *Tripid* column, since the trajectory of a given trip for a particular object is represented in its entirety before the next $\backslash n$ symbol.

[4]See [28] for details.

[5]Once can use the InputFormat to specify other types in Hadoop, and even create a subclass of the FileInputFormat class to handle custom data types – which we plan to accommodate in the future.

In either case, the entries (SPL or TPL format) are submitted to the Maper and the code for intersection is specified as part of the Reducer. We note that the source codes for all the heuristics (both for multicore and cloud implementations) and the datasets that we used are publicly available at:
http://www.sharpedgetech.com/PPEST.

As an example, we present the pseudo code of two variations of the MapReduce for $\mathbf{H}_1$ and $\mathbf{H}_2$ in Algorithm 1. and Algorithm 2, respectivelly.

Algorithm 1 MapReduce - H1 algorithm for TPL

```
 1: procedure MAP ($id$, $Tr_i$)
 2: $t_{mid,i_j} = (\lceil Tr_j.\text{End\_time} - Tr_j.\text{Start\_time}) / nReducers\rceil /2$
 3: $p$ = Time-Key-ID($t_{mid,i}$,$nReducers$, $MOD_{tstart}$,$MOD_{tend}$)
 4: EMIT($p$,$S_i$)
 5: end procedure
 6: procedure REDUCE ($p$, $[S_{1,i_1}, S_{2,i_2}, \ldots S_{k,i_k}]$)
 7: ResultSet $TR\_Intersected = \emptyset$
 8: for all $S \in [S_{1,i_1}, S_{2,i_2}, \ldots S_{k,i_k}]$ do
 9:     if $S_{j,i_j}.oid$ NOT IN $TR\_Intersected$ then
10:         if $S_{j,i_j}.oid$ INTERSECT $Q_r$ then
11:             $TR\_Intersected = TR\_Intersected \cup S_{j,i_j}.oid$
12:             EMIT ($Tr_{ij}.oid$)
13:         end if
14:     end if
15: end for
16: end procedure
```

The intersection test (and corresponding outputs) will be performed on time-segmented portions of the trajectory data in one of the Reducers (*nReduce*). To achieve this, we have the *Time-Key-ID*(...) function in the Mapper which takes is inputs a time value corresponding to the time-instant of the middle of the i-th time-portion of the h-th trajectory – t_{mid,i_j}. The number of segments is equal to the number of reducers, earliest and latest times of the MOD trajectories (MOD_{tstart} and MOD_{tend}), and the collection of the reducers, *nReduce*. The output of the function is the value of the reducer (variable p) that will be in charge of performing the intersection test for the particular subset (collection of consecutive segments) of the trajectory, the midpoint of which is t_{mid,i_j}.

However, there are some subtle observations due to the choice of representation. Namely, subsets from more than one trajectory may be assigned to a same reducer. This is why we need some "book-keeping" in terms of maintaining the *oid* of a trajectory. When an intersection is detected, the reducer knows which particular trajectory should be added to the answer set. Moreover, if a particular *oid* is already in the intersection set, the intersection test for any segment S_i for the same *oid* can be avoided.

We note that when SPL representation is used, the main difference is that the consecutive segments from one trajectory (based on the mid-point time value) are assigned by the Mapper to different reducers. Similarly to Algorithm 1, in SPL case we can avoid un-necessary intersection tests.

Algorithm 2 MapReduce - H2 algorithm for TPL

```
 1: procedure MAP ($id$, $T_r$)
 2: $t_{mid,i} = (Tr_i.\text{End\_time} - Tr_i.\text{Start\_time}) / 2$
 3: $p$ = Time-Key-ID($t_{mid,i}$,$nReducers$, $MOD_{tstart}$,$MOD_{tend}$)
 4: EMIT($p$,$T_r$)
 5: end procedure
 6: procedure REDUCE ($p$, $[T_{r1}, T_{r2}, \ldots T_{rk}]$)
 7: for all $T_{rj}$ in $[T_{r1}, T_{r2}, \ldots T_{rk}]$ do
 8:     for all $S_{rj}$ in $T_{rj}$ do
 9:         if $S_{rj}.oid$ INTERSECT $Q_r$ then
10:             EMIT ($S_{rj}.oid$)
11:             exit loop
12:         end if
13:     end for
14: end for
15: end procedure
```

Algorithm 2 describes the TPL based implementation of $\mathbf{H}_2$. As can be observed, the fundamental difference with Algorithm 1 is the manner of assignment of the data from an input file to a reducer. Specifically, the mid-point along the temporal dimension is chosen for an entire trajectory (line 2.) and is used to determine the Reducer that will subsequently perform the intersection tests (line 3.). A particular Reducer (cf. line 7) gets a collection of entire/complete trajectories to perform the intersection test. Note the subtle difference with Algorithm 1 – since the trajectories are present as whole, there is no need for the extra "book-keeping" to eliminate unnecessary calculations.

The corresponding algorithm for $\mathbf{H}_2$ when SPL is used is similar to Algorigthm 2.

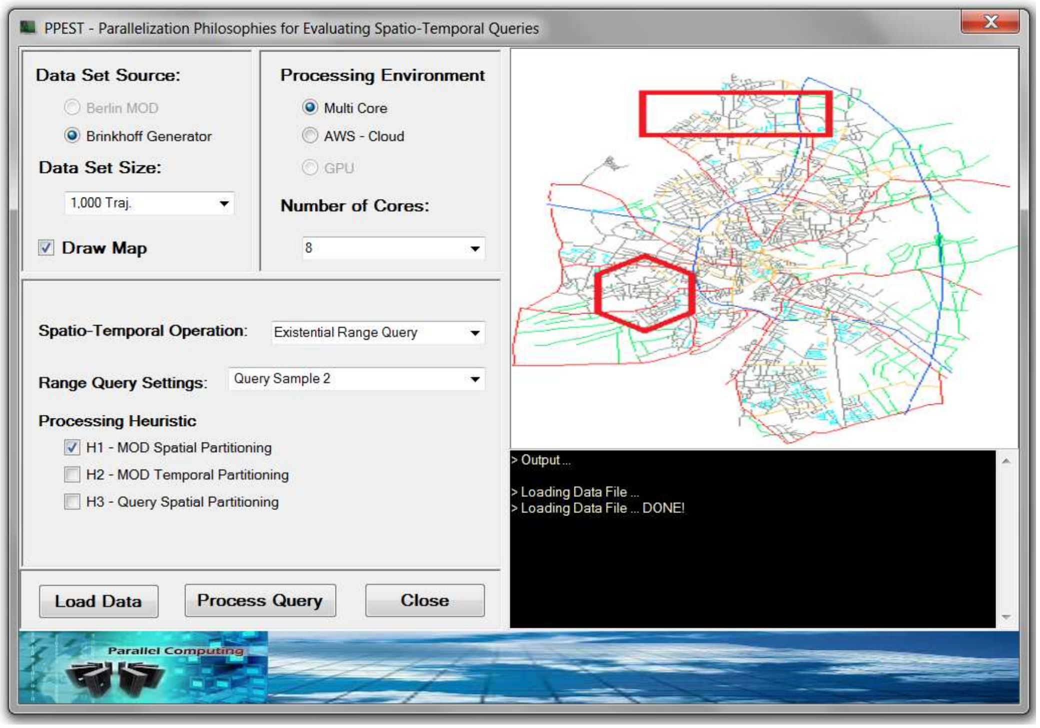
Lastly, we note that for $\mathbf{H}_3$, each Reducer is assigned a subset of the query range to perform the intersection test upon. The main difference is that for SPL, the Reducers need copies of multiple files, whereas for TPL copies of a single file are used.

4. EXPERIMENTAL OBSERVATIONS

Our algorithmic approaches have been implemented in the prototype version of the P^2EST system. Figure 4 shows the interface of the system which offers three basic categories of functionalities:

1. The users can select the data set. Pressently, we have trajectories' datasets from BerlinMOD [5], as well as several datasets generated using Brinkhoff simulator [2]. Optionally, the users may choose to visualize (top-right portion) the map used for trajectories generation and the regions of the range queries.

2. The second feature that the simulator offers is the choice of a processing environment – spanning from "local" multicore, through Amazon Web Services (both available at present). Optionally, users may select the number of cores or, in case of AWS – the number of nodes.


PPEST - Parallelization Philosophies for Evaluating Spatio-Temporal Queries

Data Set Source: Berlin MOD / [✓ Brinkhoff Generator]

Data Set Size: 1,000 Traj.

[✓ Draw Map]

Processing Environment: [✓ Multi Core] / AWS - Cloud / GPU

Number of Cores: 8

Spatio-Temporal Operation: Existential Range Query

Range Query Settings: Query Sample 2

Processing Heuristic:
[✓ H1 - MOD Spatial Partitioning]
H2 - MOD Temporal Partitioning
H3 - Query Spatial Partitioning

```
> Output...

> Loading Data File ...
> Loading Data File ... DONE!
```

Figure 4: P^2EST User Interface

Size	Heuristic	2-Core	4-Core	AC-10	AC-20
1 GB	H2	2.4%	3.3%	-29%	-46%
	H3	25%	33.3%	7%	12%
3 GB	H2	4.9%	1.9%	-39%	-40%
	H3	32%	38.5%	47%	45%
4.5 GB	H2	5.7%	4.8%	-37%	-61%
	H3	25%	41.3%	42%	36%
15 GB	H2	6.3%	4.3%	-30%	-16%
	H3	20%	37.9%	23%	17%

Table 1: Comparison of speed-ups between heuristics in multicore and cloud settings

3. The last main feature of the P^2EST involves options for specifying the parameters of spatio-temporal queries like, e.g. the polygon for the range query and the partitioning/parallelization heuristics (cf. Section 3).

In our experiments with multicore settings, typically, $\mathbf{H}_2$ would offer higher speed-up than $\mathbf{H}_1$, and $\mathbf{H}_3$ would offer the highest speed up. The benefits would increase both with the size of the data as well as the number of cores. However, when we tried to "replicate" the partitioning heuristics on the AWS-cloud, we had some surprising observations.

Table 1 shows the speed-up benefits when comparing $\mathbf{H}_2$ and $\mathbf{H}_3$ to heuristic $\mathbf{H}_1$ for the case of existential range query. We report observations for both multi-core (2 and 4 cores) as well as the AWS cloud with configurations of 10 and 20 nodes. The results are shown for datasets of trajectories with sizes of 1GB, 3GB and 4.5GB. Looking at the 3GB dataset, we see that for 4 cores setup, H_2 provide a speed up of 1.9% whereas H_3 executes 38.5% faster than $\mathbf{H}_1$for the same dataset. However, in AWS-cloud setting with 10 nodes, strangely, H_2 executes 37% slower than $textbf H_1$ (hence -37%), whereas H_3 yields a speed-up of 42% compared to $\mathbf{H}_1$.

The results reported in Table 1 above are based on averaging observations from several experimental settings. We ran up to 5 experiments for each Heuristic/Environment coupling, where each experiment varied in a number of aspects: – we used different number of polygons (representing the area of interest) with varying number of regions and shapes; – we used different MOD input data formats (i.e. time or object orders); – lastly, for the AWS-cloud implementation, some experiments used the one phase MapReduce approach while other use chained MapReduce (in our case - 2 Mapreduce). As mentioned, it was the discrepancies between the expected and observed speed-ups that gave the main motivation for pursuing the work towards developing a system that we will

Nodes	Size in GB	H2 - Segment per Line	H2 - Trajectory per Line
1	1	754.321 s	500.308 s
	3	1955.96 s	1279.791 s
	4.51	2943.877 s	2059.029 s
10	1	305.869 s	194.53 s
	3	375.065 s	295.638 s
	4.51	474.399 s	361.445 s
20	1	461.241 s	171.303 s
	3	308.367 s	248.293 s
	4.51	337.089 s	391.659 s

Table 2: Comparison of speed-ups between Data Structures - cloud w/Query Size = 4

to demonstrate.

There was another ambiguity that we observed in our experiments. Namely, for a particular query, different representations would yield different speed-ups on the cloud. Specifically, Table 2 shows the speed-up benefits when using H_2 for a query size of 4 between the two different data representations – SPL and TPL. As can be seen, for 20 nodes on AWS, the relative benefits of SPL and TPL vary for different sizes.

5. CONCLUSION AND FUTURE WORKS

We presented our comparative implementations for processing spatio-temporal range queries in multi-core vs. cloud environments. We brought some experimental observations which hint that there are some important "environmental context" which need to be considered when attempting a parallel processing of such queries. Specifically, mixing different representations and different heuristics, resulted in some counter-intuitive performances. To say the least, we observed inconsistencies among the relative benefits of particular heuristics and representation methods.

We are working towards augmenting the current functionalities of the P^2EST system in two MOD-related directions. Firstly, we would like to enable the users to experiment with different types of queries (e.g., kNN and skyline). Secondly, we investigate the option of incorporating moving objects with spatial extents (e.g., mobile shapes representing tails of hurricanes).

In addition, we are also extending the AWS implementation in the following two aspects: – combining the use of $\mathbf{H}_3$ in conjunction with $\mathbf{H}_1$ and/or $\mathbf{H}_2$; – accepting different representation of the motion plans (e.g., interpolation between two points using acceleration information), relying on the Hadoops' extensibility features.

Part of our long-term desiderata is to provide an extensible tool that will enable the users to incorporate (and test the impact of) various distributed environments (cf. [24]). Towards that, we already have some preliminary steps for incorporating Windows Azure platform and we would also like to extend P^2EST by incorporating GPU-based parallelization strategies. The later, however, will require additional research and implementation efforts that are different in a few contexts from the current instance of our work.

6. REFERENCES

[1] Fox A., Griffith R., Joseph A., Katz R., Konwinski A., Lee G., and Stoica I. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley. Rep. UCB/EECS, 28*, 2009.

[2] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2), 2002.

[3] Ariel Cary, Yaacov Yesha, Malek Adjouadi, and Naphtali Rishe. Leveraging cloud computing in geodatabase management. In *GrC*, pages 73–78, 2010.

[4] Ugur Demiryurek, Bei Pan, Farnoush Banaei Kashani, and Cyrus Shahabi. Towards modeling the traffic data on road networks. In *GIS-IWCTS*, 2009.

[5] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. Berlinmod: a benchmark for moving object databases. *VLDB J.*, 18(6):1335–1368, 2009.

[6] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra, T., Fikes A., and Gruber R. E. Bigtable: A distributed storage system for structured data. *OSDI*, page 1, 2006.

[7] Bugra Gedik and Ling Liu. Mobieyes: A distributed location monitoring service using moving location queries. *IEEE Transactions on Mobile Computing*, 5(10), 2006.

[8] Robert L. Grossman and Yunhong Gu. On the varieties of clouds for data intensive computing. *IEEE Data Eng. Bull.*, 32(1):44–50, 2009.

[9] Ralf H. Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

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

[11] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] Katja Hose and Akrivi Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J.*, 21(3), 2012.

[13] Foster I., Zhao Y., Raicu I., and Lu S. Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop*, pages 1–10, 2008.

[14] Dean J. and Ghemawat S. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[15] Rolia J., Zhu X., Arlitt M, and Andrzejak A.

Statistical service assurances for applications in utility grid environments. *MASCOTS*, pages 247–256, 2002.

[16] Chen K. and Zheng W. Cloud computing: System instances and current research. *Journal of Software 5*, 2009.

[17] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[18] Bart Kuijpers, Rafael Grimson, and Walied Othman. An analytic solution to the alibi query in the space-time prisms model for moving object data. *International Journal of Geographical Information Science*, 25(2):293–322, 2011.

[19] Xiang Lian and Lei Chen. Probabilistic ranked queries in uncertain databases. In *EDBT*, pages 511–522, 2008.

[20] Mohamed F. Mokbel and Walid G. Aref. SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. *VLDB Journal*, 17(5):971–995, 2008.

[21] Kyriakos Mouratidis, Man L. Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.

[22] Ghemawat S., Gobioff H., and Leung S. T. The google file system. *In ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[23] Thomas C. Shermer. A linear algorithm for bisecting a polygon. *Inf. Process. Lett.*, 41(3):135–140, 1992.

[24] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[25] Yufei Tao, Xiaokui Xiao, and Reynold Cheng. Range search on multidimensional uncertain data. *ACM Trans. Database Syst.*, 32(3):15, 2007.

[26] Goce Trajcevski, Alok Choudhary, Ouri Wolfson, Ye Li, and Gang Li. Uncertain range queries for necklaces. In *MDM*, 2010.

[27] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3), 2004.

[28] Goce Trajcevski, Anan Yaagoub, and Peter Scheuermann. Processing (multiple) spatio-temporal range queries in multicore settings. In *ADBIS*, pages 214–227, 2011.

[29] Anan Yaagoub, Goce Trajcevski, Peter Scheuermann, and Nikos Hardavellas. Load balancing for processing spatio-temporal queries in multi-core settings. In *MobiDE*, pages 53–57, 2012.

[30] Yu Zheng and Xiaofang Zhou. *Computing with Spatial Trajectories*. Springer, 2011.