Load Balancing for Processing Spatio-Temporal Queries in Multi-Core Settings

Anan Yaagoub Goce Trajcevski[*] Peter Scheuermann[*]

Nikos Hardavellas

Dept. of EECS
Northwestern University
Evanston, Il
anany,goce,peters,nikos @
eecs.northwestern.edu

ABSTRACT

We address the problem of efficiently parallelizing the processing of spatio-temporal range queries in multicore settings. Although the data set can be partitioned and assigned to individual cores for processing a collection of range queries, one cannot achieve an "ideal" assignment for all the cores' load. Hence, the cores should collaborate in a dynamic manner: ones that have completed their (sub)tasks should take part of the load from the cores that are still processing some of the data. We provide algorithms and synchronization data structures that achieve such collaborative behavior and we investigate their impact in different initial load-partitioning strategies. Our experiments demonstrate that about 40% speed-up can be gained when compared to static load-partitioning and that the proposed approach scales well.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithm

Keywords

Moving Objects Databases, Multicore Processing

1. INTRODUCTION

[*]Research supported by NSF-CNS: 0910952

Applying efficient parallel and/or distributed processing techniques have been among the earliest quests of every computing task [3]. Recent advances in the design of multicore processors have added a particular perspective of devising efficient algorithms in such settings [11,12]. When it comes to spatio-temporal data management, the distributed and parallel processing arise naturally as "context-attributes" in certain settings, e.g., data delivery and delegation of responsibilities among mobile clients [7,8,10], as well as query processing in sensor networks [2,5].

In our recent work [14] we presented techniques for incorporating the semantics of the underlying data when processing spatio-temporal range queries to better utilize parallelization in multicore environments. Our objective in [14] was to explore the advantages of splitting the load among the cores based on some awareness about spatial and temporal dimensions of the data and the query, when compared to readily-available generic compiling tools for generating parallel code [1].

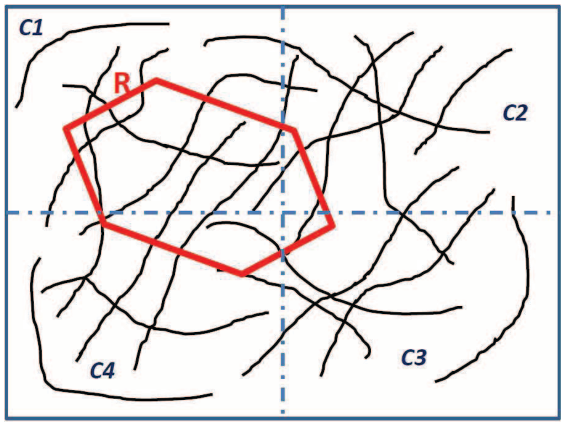

Figure 1: Static allocation of load per core

In this work, we explore a complementary dimension – namely, the impact of the (possible) *collaborative* parallel processing of spatio-temporal queries in multicore settings.

To better illustrate the motivation for our approach, consider the query:

$\mathbf{Q}_R$: *Retrieve all the moving objects that were inside the region R sometime between t_b and t_e.*

Figure 1 shows a collection of trajectories moving throughout a given region of interest, where the range query is illustrated with the hexagon R. The depicted scenario shows a setting in which the load (in terms of portions of the trajectories) has been distributed among four cores. In this particular case, one can readily see that the cores C_2 and C_3 are likely to finish their portions of the task of processing $\mathbf{Q}_R$ earlier than the cores C_1 and C_4. Intuitively, waiting for the (slower of the) C_1 and C_4 to complete their portions is what determines the overall processing time. Hence, the question that we are addressing in this paper is: could C_2 and C_3, upon finishing with their load, help C_1 and C_4, and what are the benefits in terms of the overall processing times?

Towards that, the main contribution of this work is a "load-aware" methodology for processing spatio-temporal range queries in multicore settings. After presenting the background in Section 2, we formalize the algorithmic aspects in a shared data environment and present simple communication and collaboration mechanism among the cores in Section 3. To quantify the benefits of the proposed methodology, we implemented our algorithms in two, four, and eight cores environment and evaluated them against naïve method in which the load is split among the available cores in a static manner, using the Brinkhoff simulator [4] to generate sets of up to 10,000 trajectories. Our experimental observation (Section 4) indicate that speed-ups of up to 50% can be achieved when the cores collaborate in a load-aware manner. Section 5 compares our work with the related literature, concludes the paper and outlines directions for future work.

2. PRELIMINARIES

We now present the basic concepts and notation used throughout the rest of this paper. In the MOD-literature, the motion of the objects is represented by a *trajectory* [9]:

Definition 1. *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.*

According to Definition 1, a trajectory is function from *Time* domain into the 2D Euclidian space (i.e., $f(t) \rightarrow \mathbf{R}^2$), and we consider *past* motion, which is, the entire motion of each objects is known and stored in the MOD. However, the said data type may also correspond to *future motion plan* of a given object, obtained either via some trip-planning tool (e.g., MapQuest or Google Maps), or dictated by some business fleet-planning rules [6].

We consider a multiple-reader multiple-writer (MRMW) shared memory context [11], where each core C_i can access different portions of the MOD-data, and operates on the one assigned to it. When C_i completes the processing of the range query with respect to its own data, it initiates a collaboration with another core, say C_j that has not completed the query processing yet. Assuming a unique trajectory identifier Tr_i for each trajectory in the given MOD, the initial allocation of a collection of trajectories per core is done based on the trajectory IDs (cf. [14]). Namely, assuming n cores and K trajectories in the MOD sorted by the unique ID, each core C_i is assigned a sequence of K/n consecutive trajectories.

3. PARTITION AWARENESS IN LOAD BALANCING

We now proceed with presenting the details of the proposed techniques for collaborative processing of range queries. We assume some underlying spatio-temporal indexing mechanism (cf. [9]) used to bring the trajectories relevant for processing a particular range query from the disk, without false negatives, and we focus on *pruning* and *refinement*.

The cores have shared access to all the MOD trajectories that have passed the filtering stage, however, each trajectory is initially assigned a single core. In other words, each core C_i is assigned a load $\mathbf{L}_{C_i} = \{Tr_{i,1}, Tr_{i,2}, \ldots, Tr_{i,(K/n)}\}$. We note that

For a given range-query $\mathbf{Q}_R$, let R denote the 2D spatial region of interest, and let $MBB(\mathbf{Q}_R)$ denote its minimal bounding box. Each core C_i will firstly check whether a particular $Tr_{i,j} \in \mathbf{L}_{C_i}$ intersects $MBB(\mathbf{Q}_R)$. If not, then $Tr_{i,j}$ can be safely pruned from any further consideration; otherwise, $Tr_{i,j}$ becomes a candidate for the refinement test.

The refinement, essentially checks whether any of the the segments of $Tr_{i,j}$ between t_b and t_e (the boundaries of the time-interval of interest for $\mathbf{Q}_R$) intersects R. If so, $Tr_{i,j}$ is added to the *Answer_Buffer* – a shared structure among the cores.

The key elements of our approach are the two memory components: *Finished* and *Collaboration*. Their respective purposes are explained in the sequel.

The *Finished* structure, a separate instance of which is assigned to every core, consists of two boolean variables, *Signaled* and *Accessed*. The *Collaboration* is an array of pairs (*Core_ID*, *CurrentLoad*) indicating what is the current load of a core that could potentially become of collaborator of another core. Initially, for each core C_i, *Collaboration.CurrenLoad* $= \mathbf{L}_{C_i}$. The behavior of the individual cores can be specified as follows:

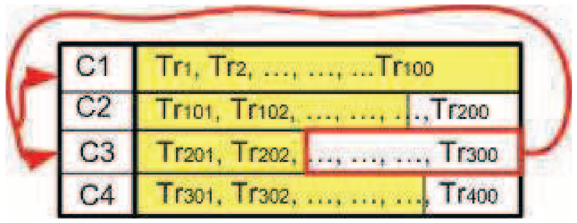
C1
Tr1, Tr2, ..., ..., ...Tr100
C2
Tr101, Tr102, ..., ..., ...Tr200
C3
Tr201, Tr202, ..., ..., ..., Tr300
C4
Tr301, Tr302, ..., ..., ..., Tr400

Figure 2: Collaborative load distribution

(1) When a given core, say C_i completes the processing of all the $TR_{i,j} \in \mathbf{L}_{C_i}$, it will set C_i.*Finished.Signaled* $=$ *true* and C_i.*Finished.Accessed* $=$ *false*. Following this, a *notification*

Algorithm 1 Range Query Processing – Collaborative Core C_i

```
Input:
$\mathbf{L}_{C_i}$: initial load of MOD trajectories
Output:
The trajectories that satisfy $\mathbf{Q}_R$
 1: while $Collaboration[i].CurrentLoad \neq \emptyset$ do
 2:   while $\mathbf{L}_{C_i} \neq \emptyset$ do
 3:     Select $Tr_{i,k} \in \mathbf{L}_{C_i}$
 4:     $\mathbf{L}_{C_i} \leftarrow \mathbf{L}_{C_i} \setminus \{Tr_{i,k}\}$
 5:     if $Tr_{i,k} \cap MBB(R) \neq \emptyset$ then
 6:       if $Tr_{i,k} \cap R \neq \emptyset$ then
 7:         $\mathbf{Answer} = \mathbf{Answer} \cup Tr_{i,k}$
 8:       end if
 9:     end if
10:     if Signal received from $C_j$ $(j \neq i)$ $\wedge$
        $C_j.Finished.Accessed = false$ then
11:       Bring $Collaboration[i].CurrentLoad$ up to date
12:       Wait $for\ notification\ from C_j$
13:       if selected as collaborator of $C_j$ then
14:         Split $Collaboration[i].CurrentLoad$ with $C_j$
15:         $\mathbf{L}_{C_i} \leftarrow$ new $Collaboration[i].CurrentLoad$
16:       end if
17:     end if
18:   end while
19:   $C_i.Finished.Signaled \leftarrow true$
20:   $C_i.Finished.Accessed \leftarrow false$
21:   $Notify\ all C_j (j \neq i)$
22:   Select $C_j$ with a largest cardinality
      $|Collaboration[j].CurrentLoad|$
23:   Split $Collaboration[j].CurrentLoad$
24:   $\mathbf{L}_{C_i} \leftarrow$ new $Collaboration[i].CurrentLoad$
25: end while
```

is sent to all the cores $C_j \neq C_i$ that have not finished the processing of their respective loads $\mathbf{L}_{C_i}$.
(2) Whenever a core C_j receives a notification that $C_i.Finished.Signaled = true$, it will check whether it is still $C_i.Finished.Accessed = false$.
If so, it will set its corresponding entry $Collaboration[j].CurrentLoad$ to $\mathbf{L'}_{C_j}$ indicating what is its currently-left load of trajectories to still process with respect to $\mathbf{Q}_R$
(3) As soon as every $C_j \neq C_i$ has finished updating its entry in the corresponding *Collaboration* array, C_i will:

(3.1) Select the C_j such that $(\forall (j, k \neq i))$ $\mathbf{L'}_{C_j} > \mathbf{L'}_{C_k}$.

(3.2) Split the IDs of the trajectories from $\mathbf{L'}_{C_j}$ into two subsets of equal cardinality $\mathbf{L}^i_{C_j}$ and $\mathbf{L}^j_{C_j}$ and, subsequently, assign
$\mathbf{L}^i_{C_j}$ to $Collaboration[i].CurrentLoad$, and
$\mathbf{L}^j_{C_j}$ to $Collaboration[j].CurrentLoad$

(3.3) Set $C_i.Finished.Signaled = false$ and $C_i.Finished.Accessed = true$, and notify the rest of the cores that a particular core for collaboration has been selected.

The behavior is illustrated in Figure 2 which shows a situation after C_1 has completed its initially assigned load of trajectories. Upon signaling its availability and communicating with the rest of the cores, C_1 realizes that C_3 is the core which has most of its initial load still left to process. The leftover load of C_3 is then split between C_1 and C_3.

The intersection test in line 5. takes a constant time, since the MBB(R) is a rectangle. If the region R is bounded by a convex polygon with s_R sides, then checking the intersection of a trajectory segment with R can be completed in $O(\log s_R)$ time-complexity, whereas it could take up to $O(s_R)$ for a concave[1] polygon (cf. line 6.). Consequently, the upper bound on the time-complexity of executing Algorithm 1 is $O((K/n)s_K(\log s_R))$ for a convex polygon R, and $O((K/n)s_K s_R)$ for a concave one. Assuming a perfectly balanced load distribution, in the sense of trajectories per-core that qualify to be part of the answer, this is also the upper bound of the time-complexity of the overall query processing. However, in practice this is rarely the case and without any collaboration among the nodes, the query processing time will be upper-bound by the time it takes for the core with largest contribution to the answer-set to complete its task. The detailed (probabilistic) complexity analysis of such cases depends on the distribution of the trajectories and the location of the query region – and we leave it as a subject of a future work. As our experiments will demonstrate, collaboration among the cores can significantly improve the overall query processing time.

Figure 3: Trajectory map used in the experiments.

4. EXPERIMENTAL EVALUATIONS

We now proceed with presenting the observations from the experimental evaluation of our proposed techniques. Our experiments were conducted on an Intel Core i7 CPU machine with 8GB memory, with 8 dual-core processors at 2.20GHz running 64-bit windows operating system. We used a task-based parallelization model among the individual core-queues in a work-stealing manner. The trajectories data-sets were generated using the Brinkkof data generator model, with 50 moving objects at the beginning and 10 external objects generating movement of 5 objects per timestamp for the beginning objects and 2 external objects per timestamp. Objects were mapped to move in the city of Oldenburg over a 5000 seconds time interval with a medium speed parameter. A total of 10,000+ trajectories were generated with each trajectory made up of more than 120 segments, and Figure 3 shows a snapshot of the data used in the experiments.

[1]We do not consider non-simple polygons, i.e., ones with holes, in this work.

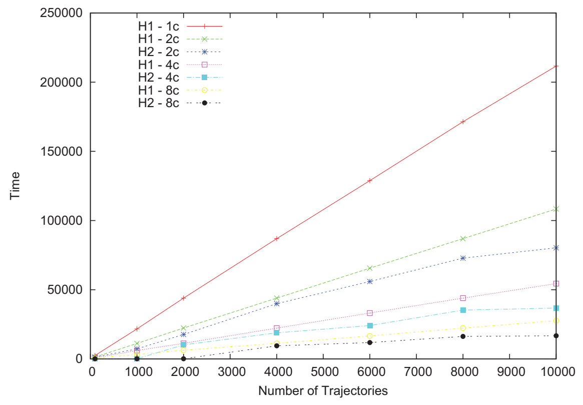

Figure 4: Single Range Query - Average Times

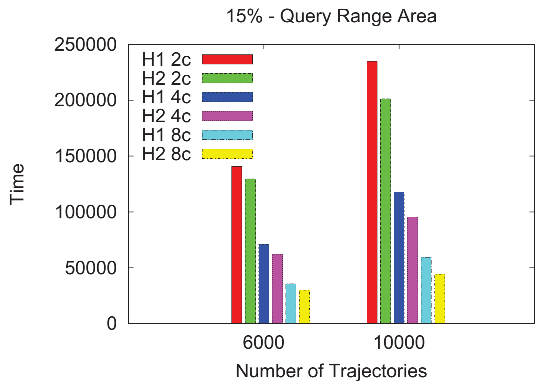

Figure 5: Medium-size Range Query (15%).

As for the query parameters, we used regular square, hexagon and octagon, with varying sizes (in terms of the percentage of the total area of interest). We ran 12 different simulations for each type of a polygon and each size, and we report the averaged results. The temporal interval of interest for the queries varied between .1h and 1.2h and, once again, we report the average values of the results. Results were generated over 1, 2, 4, and 8 cores. The final file data size exceeded 40MB[2]

We compared two heuristics (labelled "H1" and "H2" in all the graphs), where:

• H1 denotes the naïve heuristic, which statically assigns an equal number of trajectories from the dataset per available core and waits for the last core to complete its task, without any collaboration among them.

• H2 denotes the implementation of the heuristic which we presented in Section 3 and implements Algorithm 1.

Each of the heuristics was run in two, four and eight cores setting and, as indicated above, we averaged the results of all the simulation runs.

The first set of experimental observations is reported in Figure 4. The abscissa shows the number of trajectories, and the y-axis shows the time in miliseconds. Note that, as a "baseline", we also added the case where H1 executes in a single-core setting. As expected, the more cores available, the shorter the overall processing time for both heuristics. Another (expected) observation is that the overall processing time exhibits a linear dependency on the size of the data set. However, in each case, our proposed methodology (H2) shows improvements over the naïve approach, increasing proportionally with the number of available cores – 40% for 8 cores.

Our next set of experiments measured the impact of the size of the query region. Figure 5 shows the benefits of H2 over H1 for two values of the input: 6,000 and 10,000 – each bar corresponding to a particular number of cores used in the respective heuristics (as indicated in the legend). The time on y-axis is in milliseconds again, and the trend of improvements (cf. 4) – is maintained.

Figure 6 presents the exact same observations, except the *size* if the query region used was small – 2% of the total area of interest, whereas the query region covered 15% of the total area of interest in Figure 5. Comparing the two figures, we observe that the benefits, in terms of the processing time gains, are larger when the size of the query region is smaller. One main reason for this is the fact that a lot more trajectories can be eliminated during the pruning stage when the query region is smaller.

The last observation that we report pertains to the case of having multiple range queries and considering their conjunction. In other words, we are interested in:

$\mathbf{MQ}_{R_i}$: "Retrieve all the trajectories which intersect R_1 sometimes between t_{b1} and t_{e1}, AND intersect R_2 sometimes between t_{b2} and t_{e2}, AND ..., AND intersect R_M sometimes between t_{bM} and t_{eM}.

Specifically, in Figure 7 we show the averaged results of comparing our proposed methodology (H2) against the naïve one (H1) for the case six range queries were posed – all evaluated in two, four and eight cores settings. As can be observed, the trends of improvements in the query processing times are similar as for the single-query case. One observation that, in a sense, may seem somewhat counter-intuitive, is that the total processing time for the case of six range queries is smaller than the six-fold increase for a single range query. However, the explanation for it is that we can prune any trajectory that fails to satisfy a single region from one of the six range queries.

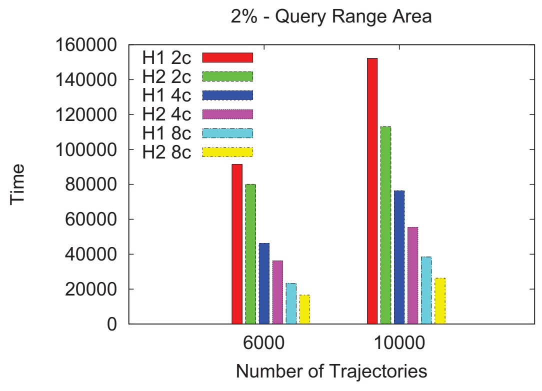

Figure 6: Small-size Range Query (2%).

[2]The dataset and the source code of the implementation is publicly available from:
http://www.sharpedgetech.com/CollaborativeMultiCore

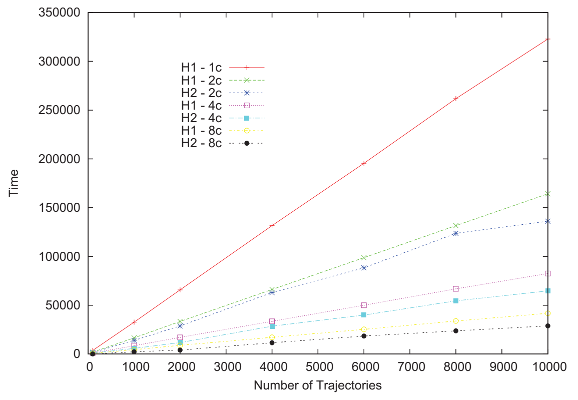

Figure 7: Multiple Range Queries (6)

5. RELATED WORK AND CONCLUSIONS

Several recent works that had the objective of efficient processing of spatio-temporal queries, have specifically addressed the aspects of distributed and parallel execution of the underlying algorithms. For example, in [7], part of the responsibility for monitoring location-based queries has been distributed among the participating mobile entities. Complementary to this, in [10] the authors capitalized on the inherent parallelism of a shared-nothing computing environment for storing and indexing the spatiotemporal data.

Some flavors of the problems addressed in [13] are similar in spirit to ours – considering the spatial and temporal dimensions of the data and queries for the purpose of load-shedding. However, the settings of the work are different – namely, the trajectory data is obtained via streams of (location,time) update and the objective is to enable efficient re-evaluation of a collection of pending spatio-temporal queries. Our goal was to investigate the load-balancing among multiple cores when processing range queries in the settings in which the trajectory data represents a complete(d) motion of the objects.

Wireless sensor networks are an environment in which the efficient processing of spatio-temporal queries [2, 5] relies on distributed and/or parallel algorithms, aiming at minimization of communication overhead. In our settings we did not have stringent energy constraints – which is something to consider in the future.

In our recent work [14] we did consider the impact of incorporating the (spatial and temporal) semantics of the data and the semantics of the queries when distributing the load among multiple cores. Complementary to [14], in this work we focused on the load-balancing among the cores.

We addressed the problem of efficient processing of spatio-temporal range queries for trajectories in multicore settings. More specifically, we focused on the collaboration among the cores for the purpose of improving the overall response time for generating the answer to a given range query, and developed a heuristic to cater to this desideratum. We presented experimental observations which provided quantitative data about the benefits of the proposed approach – demonstrating that it yields over 40% improvement of the processing time. In addition, our experiments demonstrated that the proposed approach scales well with the number of cores.

There are several extensions of the current results. Firstly, we plan to adapt the existing methodology to different kind of spatio-temporal queries – e.g., (reverse) nearest-neighbor, – and consider arbitrary boolean combinations of multiple queries. Secondly, we would like to have a more thorough investigation on the interplay of different context-attributes. Specifically, motivated by the observations in our experiments regarding the size of the query region, we plan to study the (impact of the) correlation between the distribution of the density of the trajectories and the relative positioning of a given query. how they may affect the load-balancing approach(es). In certain sense, this will spur improvements of the heuristics presented in [14] in a manner that could further benefit the load-balancing desideratum.

Lastly, we plan to analyze the data access and sharing patterns and devise partitioning heuristics that allow for the privatization of the majority of the data to computing cores. This enhancement would allow data to be stored in core-private cache slices and minimize cache coherence transactions and data transfers on the on-chip interconnect, greatly improving performance and minimizing chip energy consumption.

6. REFERENCES

[1] A quick, easy and reliable way to improve threaded performance, 2011. Cilk Plus (Intel Corp.).

[2] M. Bestehorn, K. Böhm, P. Bradley, and E. Buchmann. Deriving spatio-temporal query results in sensor networks. In *SSDBM*, pages 6–23, 2010.

[3] J. Blazewicz, K. Ecker, and B. P. (editors). Handbook on parallel and distributed processing, 2000. Springer.

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

[5] A. Coman, M. A. Nascimento, and J. Sander. A framework for spatio-temporal query processing over wireless sensor networks. In *DMSN*, pages 104–110, 2004.

[6] H. Ding, G. Trajcevski, and P. Scheuermann. Towards efficient maintenance of continuous queries for trajcectories. *GeoInformatica*, 12(3), 2008.

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

[8] B. Gedik and L. Liu. Quality-aware distributed data delivery for continuous query services. In *SIGMOD Conference*, 2006.

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

[10] M. Hadjieleftheriou, V. Kriakov, Y. Tao, G. Kollios, A. Delis, and V. J. Tsotras. Spatio-temporal data services in a shared-nothing environment. In *SSDBM*, pages 131–134, 2004.

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

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

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

[14] G. Trajcevski, A. Yaagoub, and P. Scheuermann. Processing (multiple) spatio-temporal range queries in multicore settings. In *ADBIS*, pages 214–227, 2011.