Hardware Accelerated Cloud Data Store for Data Mining over Big Spatio-Temporal Data

Nuno Henrique Nina Ribeiro Elvas Fangueiro

Instituto Superior Técnico, 1049-001 Lisboa, Portugal
n.fangueiro@gmail.com

Abstract. The recent proliferation of devices that are capable of sending information about their location over time, e.g., GPS-equipped smartphones, has turned big spatio-temporal data processing into a mainstream and highly relevant class of applications. Recent literature in the area of big data has focused on how to exploit recent hardware trends/mechanism to accelerate big data processing. In this thesis, I will focus on how to exploit Transactional Memory (TM) to accelerate applications that target big spatio-temporal data. Transactional Memory has emerged as a promising abstraction for parallel programming, which aims at enhancing performance and simplify programming of concurrent applications. More in detail, I plan to study how to integrate TM as a synchronization alternative to conventional locking for indexing data structures with spatio-temporal data.

Keywords: Spatio-Temporal Data, Concurrency Mechanisms, Transactional Memory, Performance, Indexing Structures

Table of Contents

1 Introduction

Over the past decade the amount of data generated in a global scale has increased exponentially and it appears that this trend is not going to change in the near foreseeable future. "The world contains an unimaginably vast amount of digital information which is getting ever vaster ever more rapidly" [1]. The "big data" term is used to describe these large amounts of data. Big data applications range a broad number of domains, including disease prevention, crime combat and prediction of business trends.

Big data may be split in three important components: Volume, Velocity and Variety. The volume component is pretty much self explanatory. Velocity refers to the amount of data being produced or the speed at which data is being processed in regards to its demand. Velocity may differ due to a wide range of reasons. In social networks for example, an event may become viral and so the demand for it may exponentially grow. Variety refers to the different types of information inside big data, whether it is video files, images, XML and so on. Finally big data can bring issues as security, privacy and storage space of information since this data is continuously multiplied and widely shared around the world in a hardly predictable way.

In this document, and in my thesis, I will focus on a specific sub-domain of big-data, namely the development of concurrent indexes for big spatio-temporal data applications. The relevance of spatio-temporal data applications, and the volume and velocity of such a type of data has dramatically increased over the last few years thanks to the proliferation of GPS equipped devices (e.g., smartphones). This has enabled a number of challenging data-intensive applications in the spatio-temporal domain, including traffic control [2] and geo-localized social networks [3].

The problem of developing indexes for spatio-temporal queries is well-known and several approaches have been proposed in literature. Existing solutions can be coarsely grouped into three classes: i) indexes that use specialized data-structures tailed for the multi-dimensional nature of spatio-temporal data, such as Quad-Trees, R-Trees, K-Trees, Grids [4–7], ii) solutions that first linearize spatio-temporal data into a one-dimensional space and then apply generic/non-specialized indexing solutions, such as the B+-tree [8], and iii) solutions which combine both techniques used in previous classes, using linearzation techniques to speed up the mapping of spatio-temporal information to multi-dimensional data structures [9, 10].

The problem that I plan to address in my thesis is to study efficient ways to enable concurrent access to spatio-temporal indexes, in order to take full advantage of modern multi and many core architectures.

This thesis is framed in the context of a more general trend, which has focused on how to exploit recent hardware advances to accelerate big data processing. In particular, I plan to study how to exploit hardware implementations of Transactional Memory (TM) to allow concurrent access to spatio-temporal index in a multi-threaded environment.

Transactional Memory has emerged as a promising abstraction for parallel programming,which aims at enhancing performance and simplifying programming of concurrent applications when deployed on modern parallel systems. TM represents an alternative to the traditional approach for regulating concurrency in a multi-threaded program, i.e., locking. However, the use of (fine-grained) locking is known to be quite complex, even for experience programmers [11, 12], and to lack one important property that is fundamental in modern software engineering approaches: composability [13].

In contrast, TM is a much more straightforward approach to building concurrent software, since all code that has to execute atomically has simply to be wrapped within a transaction. The underlying TM implementation transparently ensures atomicity, making programmers life much easier. Further, locks use a pessimistic approach, which ensures correctness by restricting parallelism. Conversely, TM allows to fully untap the parallelism offered by modern multicore architectures by adopting an optimistic approach that allows atomic code blocks to be executed in a speculative fashion, aborting execution only in case conflicts are actually detected.

The TM abstraction can in practice be implemented using different approaches, including software (STM), hardware (HTM) and a combination thereof, which is also called Hybrid TM. STM, being implemented in software, are very flexible but can suffer of large instrumentation overheads. HTMs, in contrast, rely on the processors' cache to store all metadata regarding executing transactions, and on the cache coherence protocol to detect conflicts among them. As such, HTM introduces very limited overhead. However, since HTM depends on the cache, which have limited capacity, it suffers from severe performance pathologies when dealing with workloads that encompass long-running transactions. Finally, Hybrid TMs aim to use concurrently STMs (for long-running transactions) and HTM (for the ones that fit in cache), hence trying to achieve the best of the two worlds.

More in detail my thesis seeks to answer two main questions:

1. whether any of existing TM implementations (Hardware, Software or Hybrid) can outperform state of the art lock-based concurrent indexes for spatio-temporal data.
2. whether the design of existing algorithms for spatio-temporal indexes can be revised to allow them to take maximum advantage of existing TM implementations.

The remainder of this document is structures as follows. In the related work section, I will overview existing solutions in the big-data domain, focusing in particular on indexing structures for parallel systems. Next I will briefly review existing literature on Transactional Memory systems. Section 3 will introduce more in detail the goals of my dissertation and the architecture of the proposed solution. Section 4 presents the evaluation plan and Section 5 presents the planned schedule of my work over the next semester. Finally, Section 6 concludes this document.

2 Related Work

2.1 Big-Data Systems

Spatio-temporal data is often composed by large data sets, mostly due to the needs of the applications that use it. With this being said, it is important to understand big data systems and how they handle the process of large datasets, before proceeding to review the state of the art systems for processing of spatio-temporal data.

MapReduce

MapReduce [14] is one of the pioneer systems for managing the processing of big data and the variants it involves. MapReduce provides computational operations over big data, which cannot be processed by a single machine to produce a result in a reasonable amount of time. Therefore, MapReduce run-time system is able to take care of the background tasks needed to work on large clusters of machines, as partitioning the input data, inter-machine communication, and handling machine failures. This facilitates the life of programmers that are new to the areas of distributed systems, thus being one of the reasons for the success of MapReduce.

Figure 1 illustrates the high-level architecture of MapReduce. As you can see, it uses an acyclic way of computing jobs. Starting by partitioning input files among the workers, and after two operations (Map and Reduce), output files are stored in disk with the result of these operations. Map and Reduce operations

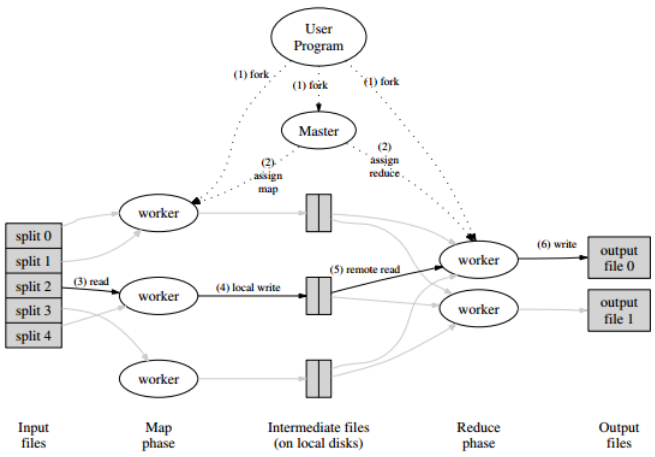

Fig. 1. Execution of MapReduce [14]

are user defined. The map function processes a key/value pair to generate a set of immediate key/value pairs. The reduce function merges all intermediate values associated with the same intermediate key. This functional model can be expressed in many real world tasks, and a large variety of problems are easily expressible as a MapReduce computation.

In relation with my work, MapReduce is a solution that focuses on the distributed area instead of the parallel, being typically deployed on clusters of machines. This makes MapReduce a good example of a big data system where TM could be used to accelerate the local synchronization among the workers of a node of the system.

Spark

Spark [15] is also a system designed for cluster computing over big data as is MapReduce. The main difference between them is that, MapReduce is built around an acyclic data flow model, that does not easily fit all types of applications. One of the key issues is that, in an iterative MapReduce job, output data needs to be written to disk and loaded back to memory to be fed as input at the beginning of the next iteration.

Spark, in contrast, focuses on those applications that reuse a working set of data across multiple parallel operations (cyclic data flow model). Ideally, a dataset to be queried would be partitioned and loaded into memory across the cluster machines, and then repeatedly queried.

In order for Spark to achieve this, it uses a distributed memory abstraction called Resilient Distributed Datasets (RDDs). "RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators" [15]. RDDs can loose some of their partitions and still be able to rebuild them. This is the main feature that makes RDDs fault tolerant, this only happens because RDDs are formally, a read only collection of records. RDDs can only be be recreated, upon failure, by recomputing them from other RDDs or by loading a checkpoint from stable storage. To create another RDD from a previous RDD it is realized a transformation. Some examples of transformations that are provided by Spark's interface are map, filter, and join. This transformations are coarse-grained, reaching to a large number of objects, thus making possible the recovery of lost partitions.

So as much as MapReduce, Spark uses a set of operators to make transformations to its data and deals with distributed systems challenges. However the data is stored in main-memory instead of disk, meaning iterative jobs are much faster. Both systems are general engines for large scale data processing and not focused on indexing, however they are good examples of systems that manage big data.

2.1.1 Spatio-Temporal Data focused Systems: Distributed Systems

This section focuses on systems that manage spatio-temporal big data with clusters of machines. These clusters are usually spread around the regions where the data resides. Thus, hotspots are a crucial challenge that these systems have to handle. These happen when a large number of objects/information gathers in a single area, forcing a reduced amount of servers to handle a large magnitude of requests.

In this section I will review two papers, both describing a system focused on distributed indexing of spatio-temporal big data. Their differences can make us better understand the needs/computational challenges spatio-temporal data arises.

A generalized solution of a big spatio-temporal indexing over a cluster of machines is MD-HBase [9]. MD-HBase uses Distributed Key-Value (DKV) stores [16], which are not spatio-temporal optimized and are used in many big data systems. DKV stores have proven to scale to millions of updates while being fault-tolerant and highly available. However, it is a key requirement in Location Based Systems (LBS), to support multi-attribute accesses, such as range queries over space and time. In the absence of any filtering mechanism for secondary attribute accesses, such queries resort to a full scan of the entire data. A full scan is wasteful, especially when the selectivity of the queries is high. In order to provide multi-attribute accesses MD-HBase uses a spatio-temporal index layer composed by two structures, the K-d-tree [6] and the Quad-tree [17], partitioned by the cluster.

Transformation of multi-dimensional location data into a one-dimensional space data, MD-HBase uses linearization techniques, such as Z-ordering [18]. A range partitioned Key-value store (HBase) is then used as the storage back-end. We can see an example of the linearization technique in Figure 2. Also

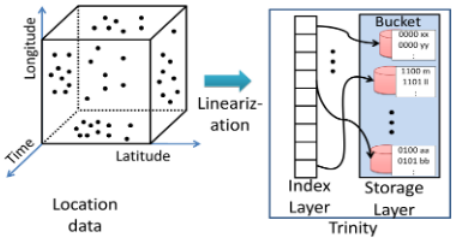

Fig. 2. Example of communication flow in MD-HBase [9].

showing the relation between the index layer and the storage layer. The index layer allows efficient real-time processing of multi-dimensional range and nearest neighbour queries. These comprise the basic data analysis primitives for location base applications. MD-HBase, then combines, the good scalability of DKV stores

with the effective indexation and query processing of the indexing structures, as the K-d and Quad-tree. MD-HBase is an example of a system adapted for spatio-temporal data indexing.

In contrast, Pyro [10] is a state-of-the art solution, for big spatio-temporal data indexing, over big clusters of machines. Pyro is able to achieve significant performance enhancements when compared to the state-of-the-art solutions. Experiments using NYC taxi dataset show that Pyro reduces the response time by 60X on 1km×1km rectangle geometries. Pyro further achieves 10X throughput improvement on 100m×100m rectangle geometries [10].

Pyro focuses on challenges which spatio-temporal big data has attached. Some of which are, hotspots, large queries over enormous regions, and translating multi-dimension data to a single dimension. Pyro brings a new concept which are *geometry queries*, these are queries for moving objects within a high resolution geometric area, forcing the back-end to handle orders of magnitude more requests. As so, general purpose distributed data stores, such as the DKV stores MB-HBase uses, are not enough to provide real time responses needed for typical object moving applications. Another issue, DKV stores cannot handle, are dynamic workloads (which may create hotspots).

Pyro is consisted of PyroDB (database) and PyroDFS (distributed file system), respectively HBase and HDFS. Pyro is implemented by adding 891lines of code into Hadoop-2.4.1, and another 7344 lines of code into HBase-0.99 [10]. In order to translate multi-dimension data to a single dimension, PyroDB internally implements Moore encoding [18], translating *geometry queries* into range scans (smaller areas to scan). Moore-Encoding helps to reduce the number of range scans by 40% when compared to ZOrder-Encoding [10]. Using an index structure as the Quad-tree, Pyro is able to divide space by tiles and intersect the data from the Moore Encoding, in order to make range scans over those tiles. PyroDFS uses a block grouping algorithm in order to preserve data locality on the occurrence of hotspots, this happens when PyroDB has to split region servers to balance the workload.

2.1.2 Spatio-Temporal Data focused Systems: Parallel Systems

So far, I have reviewed spatio-temporal systems which focus on the processing of large amounts of data over large clusters of machines. Next, I will focus on *concurrent* indexing data structures for spatio-temporal data on parallel systems, which will represent the area of work of my thesis. These solutions aim at taking maximum advantage from modern multi and many core architectures, supporting *concurrent* execution of updates and query operations. As such, they represent important building blocks for several of the distributed platforms for spatio-temporal data reviewed in the previous section.

Firstly, I will describe three main indexing structures for spatio-temporal data, namely Quad-trees, the R-trees and Grids. Secondly, I will review recent solutions aimed at extending these indexing data structures to support *concurrent* operations.

As I will clarify in Section 3, these state of the art indexes will be target of my thesis work, which will aim at enhancing their scalability thanks to the use of Transactional Memory.

In-Memory Indexes for Spatio-Temporal Data

Recently, the proliferation of devices that are capable of sending information about their location overtime (e.g., GPS-equipped smartphones), induces to spatio-temporal index structures, the scalability challenge of managing and querying numerous spatial objects. In order to meet the scalability requirements, there is a growing trend to maintain data fully in-memory to ensure real-time responses (e.g., [7]). In this regard, main-memory data base management systems (MMDBMS) seem to be a viable solution as the price of memory continues to drop [19]. Moreover, main-memory data structures are expected to be the ideal playground for HTM, which I intend to study in my thesis.

I must remind you that, in the literature, there exist a number of variants of these base algorithms. Such variants may exhibit different performances for what concerns update/query efficiency. Yet, in this report, I am describing the most common implementations.

Indexes APIs The following indexes (i.e., Grid and R-tree), support the same set of operations, such as the Update (Object_id, old_x, old_y, x, y), the RangeQuery (Rectangle q) and, the kNNQuery (Point q, num_neighbors k).

Insert and Delete operations are also supported by both indexes, however they are implicit in the Update operation, thus I am not going to explicitly refer to these. In the following, I will describe how these operations are implemented for each index.

R-Tree

An R-tree-based indexing and query processing has remained a focus of research in spatial databases for more than two decades now. The R-tree is known for its the support for a large number of different query algorithms, and its ability to index spatial-extended objects. However the main issue with the R-Tree is its poor performance updates [4].

Figure 3 depicts the R-Tree's structure. The R-Tree is a hierarchy structure of minimum bounding rectangles (MBRs). An MBR is the smallest rectangle that encloses a group of objects. There are two types of nodes in this tree. Internal nodes which have other nodes as child's and leaf nodes who are a the bottom of the tree and contain moving objects in them. These nodes are organized according to their MBR. Parent nodes MBR contains all child nodes MBR. Finally, the R-Tree is defined with two parameters that are needed to adjust the tree in order to achieve a good tree balance and thus a good performance. These are node size (ns), expressed in cache lines and minimum children (mc)

expressed as a fraction of the full node. In any node of the tree its entries must occupy at least *mc* percentage of *ns* otherwise the node is considered underfull. If a leaf node is too full with objects, it is considered a node overflow, and it splits.

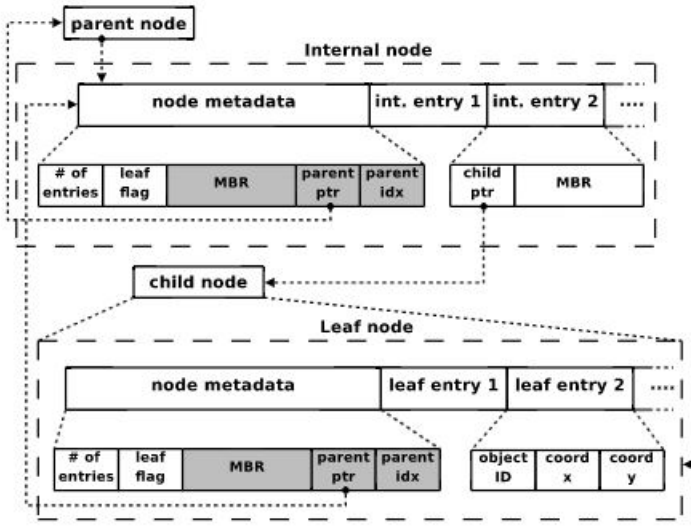

Fig. 3. Structure of a conventional R-tree [4].

Queries: The *range query* in the R-Tree is performed as a depth-first traversal from the root down to the leaves accessing the nodes with MBRs overlapping the range of the query. When the leaf nodes are finally reached, objects that satisfy the range query are outputted.

The *kNN query* in the R-tree is processed as a best-first traversal. As in the grid, kNN queries on the R-Tree also use a priority queue, in this case the priority queue refers to the nearest MBRs to the query point. Once the leaf nodes are reached, the k nearest objects are outputted.

Updates: Here, I explain the update operation in the R-Tree and explain why it has a poor performance. The updates are performed by two separate top-down operations, deletion and insertion. Figure 3 illustrates the tree structure.

The deletion operation descends the tree from the root searching the old position of the object. This is done by recursively accessing the nodes that contain the position of the object in their MBR. MBRs may overlap, so more than one path can be visited while searching for the object. Once the required leaf node is located, the appropriated entry is deleted. Finally, ancestors (all nodes above the current) MBRs, may become not minimum (the minimum value of MBRs differs with implementations), thus requiring traversing the tree back to the root

adjusting the MBRs. Furthermore, the nodes may become underfull (as earlier explained), requiring an expensive reinsertion of their entries.

The second part of the update operation uses the insertion algorithm. Insertion begins by traversing the tree from the root to the leaf node as well. At each node a heuristic function is called to choose the most suitable path to further descent the tree. When a suitable leaf node is located, the object is inserted there. Once again, by inserting a new object, ancestors MBRs may become invalid and have to be adjusted traversing the tree to the top. Also when inserting a new object on the leaf node, we might be causing a node overflow (earlier explained), which originates a node split on that leaf node. By doing this we are adding another entry to the parent node, which himself may exceed node capacity, and split. The split may propagate up the tree.

Summarizing, a single update results in at least 3 tree transverses, thus this being the main reason why updates in the R-Tree are very poor, performance wise.

Grid

Figure 4 illustrates the Grid's structure. The secondary Index (grey part of the image) will not be accounted for in this review, since this is only used for optimizations and is not a core element of the standard structure.

A fixed Grid is just a space-partitioning index where a defined area is divided into cells. Objects within a particular cell grid belong to that cell. Grid cells are stored as a two-dimensional array. Each grid cell within the array stores a pointer to the linked list of buckets that contain the object data. Objects are incrementally stored in buckets following no specific order. Thus the grid is defined by three parameters: grid_area, grid_cell_size (gcs), and bucket_size (bs) [4].

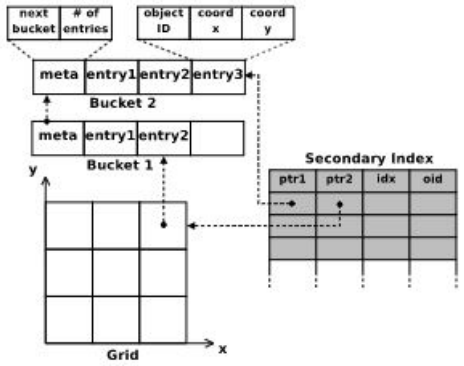

Fig. 4. Structure of a conventional Grid (the grey part is used for optimizations) [4].

Queries: The *range query* is defined by a rectangle with two corner points defining its limits. Cells fully covered, in the given rectangle query, have all their objects included in the result list. In contrast, cells that are not completely covered must have all their objects scanned, in order to check, if they are inside the query range and only then inserted into the result list.

The *kNN query* is defined by a query point, q, and the number of nearest neighbours required, k. The fundamental procedure for performing kNN queries on Grids is to divide cells into different groups, depending on the minimum distance to the query point, q. Cells with approximated minimum distances stay in the same group. Then, using a priority queue to store cells, these are traversed from the nearest to the further, until the k-nearest objects are determined.

Updates: Updates on the grid are pretty simple, if the old position and new position of an object are in the same grid cell, the update is local, then it is only necessary to scan the buckets to find the specific object and update its values. If the new position of the object however is in a different cell, the old object must be removed and a new object must be created with the new values in the new cell. This is managed by insertion and deletion algorithm that ensure fast processing of these updates.

Quad-Tree

The term quad-tree is used to describe a hierarchical data structure whose common property is the principle of recursively decomposition of space. In general quad-trees are classified in regards to three major characteristics. These are as follows, i) the type of data that they are used to represent, ii) the principle guiding the decomposition process, and iii) the resolution (variable or not) [17].

The quad-tree I will review is concerned with the representation of region data. Commonly termed *region quad-trees*, usually this type of quad-trees use successive subdivision of a image array (binary array filled with 0's and 1's) into four equal-sized quadrants. Figure 5 illustrates the mapping of region (a) into the binary array (b). All bits in the binary array contained by the region are at 1 the rest are at 0. In order to map the binary array (b) to the quad-tree (d), continual division of the image array is made into four equally-sized tree quadrants (each a block in (c)), until these are filled with only 0's or 1's (single pixels if necessary). When a quadrant/block is filled with equal bits, all 0's or 1's it means that it contains a part of the image array that is either fully inside the region or fully outside. In order to distinguish whether a quadrant/leaf is inside/outside of the region two colours are used. The black colour means they are inside and the white colour means the contrary. To notice, in tree (d) one can see that the tree is divided by (NW, NE, SW, SE). This typically happens in quad-trees that map a certain region and illustrates the partitions position.

With this simple example we can evaluate how the quad-tree is able to create a spacial index from an image array. Indexed positions (in leaf nodes) retain information of rather they are inside or outside the region. An evolution to the of the *region quad-tree* is the Multi-version Linear Quadtree [17]. This quad-tree

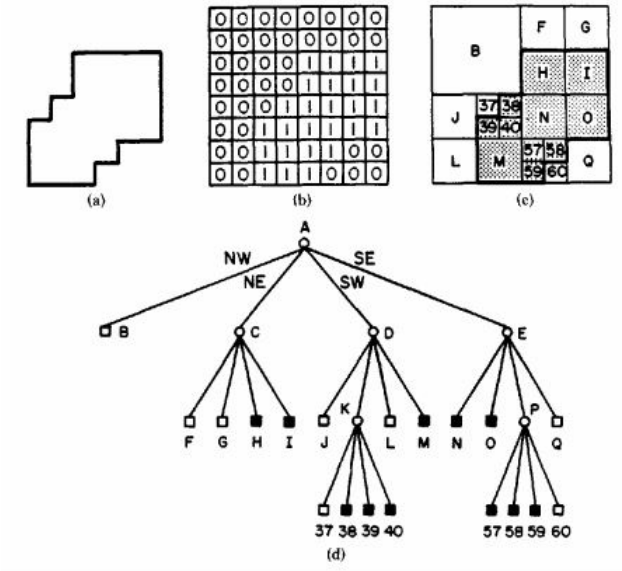

Fig. 5. A region, its binary array, its maxima] blocks, and the corresponding quad-tree. (a) Region. (b) Binary array. (c) Block decomposition of the region in (a). Blocks in the region are shaded. (d) Quadtree representation of the blocks in (c). [17].

is used as a spatio-temporal index, whereas images are stored over time. If a sequence of N images has to be stored in a Linear Quadtree, each image labeled with a unique timestamp Ti (for i=1, 2, ..., N), then updates will overwrite old versions and only the last inserted image will be retained [17]. This quad-tree also supports the main operation required for all the well-known spatial queries for quadtree-based spatial databases (e.g., range queries, and nearest neighbour queries).

Analysis of advantages and disadvantages the indexes

For this analysis I use as reference the book [20].

Grids use a *flat partition* of space, which is simpler since it is easier to reason about and simple to implement. Memory usage is constant since adding objects does not require creating new partitions. Finally it can be faster to update, since updates in trees (or hierarchical structures) require spacial partition and this may mean adjusting several layers of the hierarchy, as previously mentioned.

This makes Grids object independent, objects can be added incrementally and can be moved quickly because they will not change the partition. However the partition may become unbalanced, since it is a fixed partition it will not adapt to the number of objects it stores, which may cause suboptimal performance. This happens because a single grid cell may contain too many objects to consider at a time. In order to solve this problem, and to provide load balancing, adaptive

grids may be used as [21]. These grids use another indexing structure as the quad-tree to partition the cells in regards to the objects in the grid.

Trees in contrast use a *hierarchical partition* of space. Trees handle empty space more efficiently than Grids, since cells in Grids are predefined and there can be many empty cells. However this does not happen with trees, since empty space may be represented by one single node. They also handle densely populated areas more efficiently. If you have many objects in one point, *hierarchical partitions* adapt and subdivide into smaller partitions. Making it possible to continue to have only a few objects to consider at a time. However, insertion and deletion of objects may require adjusting several levels of the hierarchical partition, making updates a drawback in trees.

To finalize, quad-trees make the best of both worlds when compared with grids and typical multi-dimensional trees. Quad-trees are partition object independent and hierarchy object dependent. In fact partition do subdivide when objects are added in the tree, however the division is always known before hand (4 smaller leafs). Quad-trees start with the entire space as a partition. As objects are added, the quad-tree starts partitioning space slicing always into four smaller partitions (or leaf nodes), thus making the boundaries of those partitions always half of the previous.

Objects can be added incrementally, by identifying the right partition that should be maintaining it and adding it there. If this addition saturates the capacity (typically a fixed threshold) of the partition, it gets subdivided. The other objects in that partition get pushed down to new smaller partitions. Removing objects is quite simple as well. You remove the object from the partition, if the parent partition's total count is smaller than the minimum threshold you can collapse the subdivisions. Finally, partitions are balanced since this follows an hierarchical division of the objects, maintaining always only a few objects on each leaf/square.

Concurrent indexes for spatio-temporal data

As we have seen so far there are conflicting needs between update-efficient vs query-efficient index structures. On top of this, adding concurrency to index structures raises the problem of consistency. In order to ensure consistency updates and queries have to be synchronized. Although their processing may be parallel, their execution must be appropriately synchronized in order to ensure correct system behaviour, i.e., roughly speaking, executions equivalent to serial ones. The techniques used to ensure synchronization have to be carefully designed to favour parallelism and take maximum profit of the full power of the multi-core processors.

An interesting solution, in this sense, is the one presented by Darius Sidlauskas et al [7], which introduces new semantics that ensure a high degree of parallelism and ensure up-to-date query results. *Freshness semantics* is the main semantic that this paper proposes and it is implemented on one of the indexes structures that I intend to accelerate via HTM: the PGRID(concurrent Grid with freshness semantics).

In order to obtain better parallelism *freshness semantics* lower the degree of consistency of the system, however maintaining it still acceptable for target applications. For this purpose I make a quick review of the levels of consistency [22], ordering them from higher to lower consistency.

1. degree 3 - full serializability.
2. Snapshot Isolation (SI) - provides a consistent view of the system to a transaction as of its start. During the time of the transaction, no writes done by concurrent transactions affect its snapshot. No phantoms here.
3. Freshness semantics - does not allow i4 phantoms [7].
4. degree 2 - allows all four types of phantoms.

Freshness semantics enable updates and queries to be performed concurrently and ensure consistency situated between degree 2 and SI. It is less restrictive than SI since it allows some phantoms, but it is more restrictive than degree 2 of consistency since it does not allow i4 phantoms. All other type of phantoms can be tolerated easily by target applications. However i4 phantoms happen when an object, yet to be scanned, updates its position to an already scanned area. Thus, the object is still in the query range but it seems to have disappeared. This is not acceptable for any target application.

With *freshness semantics*, studies conducted on modern processors show that PGrid (and other proposals in the paper) scales near-linearly with the number of hardware threads. Thus, is able to benefit from increasing on-chip parallelism. In order for P-Grid to implement *freshness semantics* it uses fine-grained locks as the concurrency mechanism. However, as previously mentioned, locks have pessimistic concurrency control, thus lowering the potential parallelism of the system. In my solution I will replace or adapt the concurrency mechanism with all variants of TMs implementations, and try to achieve better performance.

2.2 Transactional Memory

Transactional Memory (TM) is a concurrency mechanism that many researchers believe to be the path to follow to untap the parallelism of modern multi-core systems, while drastically simplifying parallel programming. TM has several design characteristics that achieve greater parallelism than locks and for that reason it can be better suited for new multi-threaded processors.

TM is a mechanism that offers the possibility for programmers to define the transactional area of code. TM will then take under account all concurrency issues on that given area of code and provide parallel transactions.

Transactions have been successfully implemented in database systems since the early 90's [23]. Transactions provide atomicity, a transaction either fully completes its read/write operations and commits or it has no effect (aborts). Transactions buffer their writes and store their reads in memory in order to detect conflicts and abort the transactions that compromise some target consistency criterion (tipycally opacity [24]).

In the TM programming environment, programmers simply define as transactions those program regions that access shared variables. TM runtime systems

achieve high concurrent performance by optimistically executing the transactions in parallel [25].

Conflicts happen when two transactions address the same memory location. Conflict detection may be at transaction start time (eager), or at the end of the transaction (lazy).

The most important advantages that transactions bring in comparison with locks are their simplicity of implementation and their optimistic conflict detection.

There are various possible implementations of TM, in Hardware (HTM), in Software (STM), or combinations thereof, also called Hybrid TM (HyTM). In the following, each of these variants will be overviewed.

2.3 Hardware Transactional Memory

Hardware Transactional Memory (HTM) is a concurrency mechanism which provides the use of atomic operations (transactions) at cache level. This occurs via ad-hoc extensions of the processor instruction set (e.g., TSX in Haswell [26]).

In commercial HTM implementations, HTM uses the processors' cache to store the metadata generated by transactional read/writes, and the cache coherence mechanism to detect conflicts. Performance is increased since memory operations are made in cache and there is no need for software instrumentation, which greatly reduces overheads. The consequence of this design is the low amount of memory (cache) available to store the metadata (read/writes) of transactions. Workloads, including long transactions, may exceed hardware memory capacity, resulting in transactional aborts and inducing a big overhead.

Due to these (and other) limitations, HTM transactions are never guaranteed to complete (best-effort HTMs). A fall back plan (resorting to use a global lock) is required to maintain at least serial performance in case transactions fail repeatedly (and potentially deterministically) in hardware.

Recently, IBM and Intel have introduced HTM implementations for High-Performance Computing (HPC) and commodity processors respectively. "This represents a significant milestone for TM, mainly due to the predictable widespread availability of Intel Haswell processors, which bring HTM support to millions of systems ranging from high-end servers to common laptops" [27].

In the rest of this section the three topics below will be introduced:

1. Overview of existing HTM implementations
2. Hardware Transactional Memory Interfaces

Firstly I will make an overview of four commercial processors that have HTM implementations comparing their architectural structure. Secondly I will present different HTM interfaces that programmers use in order to implement HTM.

Overview of existing HTM implementations

Paper [25] investigates in depth the HTM implementations existent in four commercial processors from Blue Gene/Q [28], zEnterprise EX12 [29], Intel Core [30] and POWER8 [31]. From this research various design decisions of each processor are revealed. In table 1 we can observe such design choices. Next,

Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Conflict-detection guranularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Transactional-load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way, (shared by 16 cores)	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
Kinds of abort reasons	-	14	6	11

Table 1. HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8 [25]

I will compare two choices that have the most impact in the HTM environment.

The first choice is *Conflict-detection granularity*. The *Conflict-detection granularity* in zEC12, Intel Core, and POWER8 it is their cache-line sizes. When working with HTMs, even if two transactions address different bytes of a cache-line, this results in a conflict. This is called a *false conflict*. As shown in Table 1 zEC12 has the bigger cache granularity, which provides better *data locality*. However, in the case of HTMs, it induces the bigger abort rate due to *false conflicts*.

The second choice is *Transactional Capacity*. The *Transactional Capacity* is the maximum amount of memory data that can be accessed in a transaction [25]. This resource is scarce in HTMs since transactions use the cache to store their metadata. This metadata is composed by accessed memory locations, for conflicts detection, and buffered transactional stores.

In Table 1 we can observe that the load capacity is usually bigger than the store capacity. This happens because transactional stores must store the actual data into buffers. In contrast, to ensure conflict-detection, it is only necessary to record the accessed memory locations. Finally, *capacity overflow* is triggered when a transaction tries to access a cache line that will exceed the cache's capacity.

HTM Interfaces

The latest Haswell processors made by Intel come with a new extension of the instruction set architecture (ISA) [26], which provides support for Hardware Transactional Memory, called Transactional Synchronization Extensions (TSX). TSX provides two software interfaces to handle HTM, named Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

Hardware Lock Elision can be seen as subset of RTM, meant to give the possibility for legacy (lock-based) programs to also use HTM. HLE was made to replace lock implementations with two provided prefixes _XAQUIRE and _XREALEASE. These prefixes are used with locks. When the software acquires the lock, the hardware has the ability to check if a thread executing the critical path is going to have conflicts with other threads (speculatively). Threads that will not generate conflicts may run in parallel with others. Thus the lock is elided and threads may run without requiring any communication with the lock. However a conflict might happen for various reasons, in that case threads get rolled back and acquire the lock. HTM is used to check for conflicts.

The second HTM interface is Restricted Transactional Memory (RTM). This interface does not require locks as HLE. Instead it provides three new prefixes XBEGIN, XEND and XABORT to handle transactions (more prefixes are available). The programmer can start, end, or abort a transaction in any part of the program. Another difference in RTM is that programmers must define a fall-back path for an aborted transaction. RTM brings more flexibility to the programmer as he can choose what to do when a transaction aborts and explicitly start or end transactions without requiring locks.

In summary HLE is used for compatibility with legacy programs that use locks. In contrast, RTM explicitly enables the programmer to define the transactional critical areas, thus bringing more flexibility. However it requires programmers to define a fall-back path.

2.4 Software Transactional Memory

Software Transactional Memory has been focus of intense research along the past decade [12, 27]. Software is responsible to store the metadata (reads/writes) of transactions and detect conflicts. Most implementations have structures that identify which memory regions are being accessed by each transaction (i.e., ownership records). Nevertheless NOrec [32], which I will present in the following, abolishes such structures.

Since STMs are not cache dependent, as HTMs, they are better equipped to handle long transactions. However, STMs induce a bigger overhead when compared to HTMs, due to the need of software instrumentation.

Throughout the years a number of STM designs have been explored:

1. lock-based vs lock-free - whether an STM uses locks to handle concurrency.
2. encounter-time locking vs commit-time locking - when conflict detection is performed either at commit time (lazy) or encounter time (eager).
3. word-based vs object based - granularity at which memory is accessed, usually used to validate transactions data-sets.
4. Contention Management Scheme (CMS) - Contention manager module is responsible for ensuring that the system as a whole makes progress [33].
5. visible readers vs invisible readers - whether readers actions are visible to the rest of the system.
6. weakly atomic vs strongly atomic - whether non-transactional code may access the same data as of transactions.

In this section I will present two STMs, Swiss-TM [12] and NOrec [32]. I choose this two STMs because they are representative of different designs and are optimized for different workloads. Swiss-TM design choices focus on complex workloads and achieving good performance with large scale benchmarks, which require long transactions. NOrec however, is a much simpler approach to STM, and it is focused on applications with few concurrent writers and many concurrent readers, thus improving the performance of read only transactions.

Swiss-TM

Swiss-TM focuses on large applications as games or business applications in order to use the power of multi-core processors to the full extense. These are the design choices of Swiss-TM "Swiss-TM is lock- and word-based and uses (1) optimistic (commit-time) conflict detection for read/write conflicts and pessimistic (encounter-time) conflict detection for write/write conflicts, as well as (2) a new two-phase contention manager that ensures the progress of long transactions while inducing no overhead on short ones" [12]. Since Swiss-TM supports mixed workloads consisting of small and large transactions, as well as complex data structures, it uses these software mechanisms to achieve a greater performance.

Correctness, i.e., opacity [24], in Swiss-TM is achieved using a metadata structure where the current timestamp of each transaction is stored. The timestamps are ordered using a global counter. To validate a transaction, the timestamp of such transaction must never be greater than the timestamps of the resources it read. If such resources have a bigger timestamp, it means that another transaction concurrently changed them. Finally, a locking strategy is used to ensure that changes in memory are atomic.

Swiss-TM uses a mixed conflicting scheme. For read/write conflicts an optimistic (lazy) approach is used in order to allow more parallelism between transactions. Moreover big transactions may address memory locations being read by other transactions. If a eager/pessimistic approach was used all other transactions would abort, leading to an overall slowing of the system. In contrast for write/write conflicts it is used a pessimistic approach, in order to prevent transactions that are doomed to abort, from running and wasting resources.

The two-phase CMS enables that short read-write or read only transactions have no overhead, while favouring the progress of transactions that have made many updates (big transactions). For short read-write or read only transactions it is used a timid CMS, which aborts transactions at the first encountered conflict. For bigger transactions, it is used the Greedy [33] CMS, which induces more overhead, but ensures completion of long transactions, thus preventing starvation.

NOrec: Streamlining STM by Abolishing Ownership Records

In order to better understand NOrec it is best to learn about Transactional Mutex Locks (TML) [34], the system where it "evolved" from. This is a particular simple system, which uses a single global sequence lock to serialize writer transactions. A single global sequence lock ensures that every invisible reader in the system must be prepared to be invalidated (restart), once one of its peers becomes a writer.

The primary advantage of a sequence lock is that readers are invisible, and need not induce the coherency overhead associated with updating the lock. In contrast, the primary disadvantage is that doomed readers may be concurrent with writers, thus readers may read inconsistent data modified by writers. TML's built in memory management system handles these dangerous memory deallocations situations. Summarizing, TML is a very low overhead STM that is highly scalable in read-mostly workloads.

However, there are two main properties that limit TML's scalability, which will be improved and result in NOrec. Firstly, the eager acquisition of the lock by writers, in a single global sequence lock, means that only one writer may be active at a time. Secondly, invisible readers must be extremely conservative and assume that they may be invalidated by any writer, even if the writers accessed data did not conflict with the readers.

To solve the first problem, NOrec uses a lazy conflict-detection and a redo log for concurrent speculative writers. Modified values are locally stored in the redo lock, if the transaction is able to commit then the values are stored in the shared memory. Furthermore, writing transactions do not attempt to acquire the lock until their commit points, which ensures that the lock is held the minimum amount of time possible, thus increasing the likelihood of read-only transactions to commit.

To handle the second problem, NOrec uses Value-Based Validation (VBV) to allow transactions, both readers and writers, to detect if they actually where invalidated by a committing writer rather than making an conservative approach. VBV allows the abolishment of ownership records (thus the name NOrec), which is used in many other STM implementations to associate transactional metadata with each data location. In many STM the maintenance of this table is a hard task and induces a big overhead. With VBV, the actual address and value of the memory location are logged. Thus, validation only consist on rereading the adresses and verifying if there is a point (namely now) where the

transaction's reads could have occurred atomically (to check if the value present in the memory address is equal to the one in the log). The global sequence lock provides a "consistent snapshot" where these validations can occur.

Finally, these new mechanisms enable NOrec to achieve a greater scalability. Since there can be multiple writers and transactions (either readers or writers) are able to "survive" through a writers non-conflicting commit.

2.5 Hybrid Transactional Memory

Hybrid Transactional Memory (HybridTM) [35], is a mixture between both transactional paths, i.e, HTM and STM. Theoretically in order to take advantage of the strengths of both STMs and HTMs, HybridTMs seem to be a viable solution. In order to provide support for best-effort HTMs, the Hybrid solution may be used as the fall-back path (instead of the usual global lock approach) of the system. Thus, this is achieved by instrumenting a hardware transaction with accesses to STM metadata, allowing it to detect conflicts with software transactions. This approach enables logical transactions (HTM transactions) to switch over to STM, per-transaction basis. However, the synchronization required to support both TM implementations within the same system can cause major overheads. Thus, recent research [27] reveals that the combination of best-effort HTMs with STMs has a lower performance than the use of the separate transactional paths.

Among the various HybridTMs presented in the literature, hybrid NOrec [36] is probably one of the most popular solutions. Hybrid NOrec is a HybridTM which combines the NOrec STM (previously presented) with a best-effort HTM. Hybrid NOrec allows concurrency between both STM and HTM, however hardware transactions must respect the single writer NOrec protocol. Thus, hardware transactions cannot commit while there are running software writeback transactions and must signal their commit, so as to trigger validation by concurrent software transactions. Hybrid NOrec uses algorithms, as demonstrated in Figure

```
1  padded unsigned seqlock

5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   seqlock = seqlock + 2
```

(a) Our naive algorithm

```
1  padded unsigned seqlock
2  padded unsigned counter

5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   counter = counter + 1
```

(b) 2-Location

```
1  padded unsigned seqlock
2  padded unsigned counter[]
3  thread local unsigned id

5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   counter[id] = counter[id] + 1
```

(c) P-Counter

Fig. 6. Instrumentation of hardware transactions in hybrid NOrec algorithms [36].

6, to ensure consistency within STM/HTM concurrent transactions. Note that

these algorithms are only used for hardware transactions and are not applied to read-only transactions.

The first, most primitive algorithm, is simply to read the global sequence lock (used by HTM and STM) whenever a hardware transaction wants to start. If the lock is taken, the hardware transaction spins until the lock is released. Since the lock value is incremented when this happens, and since the read of the lock by the hardware transaction is transactional, the hardware transaction aborts and retries.

The problem with this first approach is that any hardware transaction will conflict with any software transaction, due to the early read of the sequence lock in HW POST BEGIN. The second, and more worrisome problem, is that any hardware transaction will conflict with any other hardware transaction due to the increment of the sequence lock when committing. These conflicts exist for each transaction's full duration, eliminating concurrency.

Concurrency between HW transactions is managed by the hardware, so there should not be any type of instrumentation at software level. To solve this second issue, algorithm 2-Location uses a new variable named counter. Counter is used by HW transactions to signal SW transactions of their commit, making the conflict between other HW transactions only at commit time and thus improving concurrency between HW transactions. The last algorithm is P-Count, it uses a different counter per thread to avoid even further conflicts with other HW threads, however both algorithms require software instrumentation to include verification to the counter variable, inducing more overheads.

We here have an example of a HybridTM that uses HTM as the main path to handle transactions. A second fall-back path with STM is used to handle transactions that the HTM path does not has capacity to support (e.g, long-transactions).

3 Solution Architecture

As seen in the related work section Transactional Memory has emerged as a promising abstraction for parallel programming, maximizing performance and simplifying programming of concurrent applications when deployed on modern parallel systems. TM represents an alternative to the traditional approach for regulating concurrency in a multi-thread program, i.e., locking.

I have two main goals with the work I will be developing in my thesis:

i) to investigate whether any of existing TM implementations (Hardware, Software or Hybrid) can outperform state of the art lock-based concurrent indexes for spatio-temporal data.

ii) to study whether whether the design of existing algorithms for spatio-temporal indexes can be revised to allow them to take maximum advantage of existing TM implementations.

The beginning of my solution will focus on the first goal, and will aim at porting and adapting two state of the art spatio-temporal indexes [4, 7], called uGrid and P-Grid, to use various TM implementations (Hardware, Software,

Hybrid). It should be noted that uGrid is not a concurrent data structure, i.e., it is designed for single threaded access, whereas PGrid is.

Thanks to the TM abstraction, it should be simple to turn uGrid into a thread-safe data structure, by simply wrapping the query and update operations of the index with atomic transactions. An analogous approach will be followed for p-Grid: its lock-based synchronization scheme will replaced with a TM based one.

With both data structures, I plan then to carry out an extensive experimental study aimed at evaluating the efficiency of alternative TM implementations, and in particular HTM, STM and HyTM. It is interesting to note that uGrid, which is single-threaded, will allow us to evaluate to what extent TM can be used to efficiently synchronize data structures that were not designed to operate in presence of concurrency.

Conversely, p-Grid is a state of the art concurrent data structure that uses a highly optimized and complex fine-grained locking scheme. Hence, it represents a very competitive solution, and an ideal benchmark to test the effectiveness of existing TM-based solutions.

As for the second goal of my thesis, I plan to build on the insights gathered during the first stage of my thesis, and capitalize on the lessons learnt by benchmarking and profiling pGrid and uGrid with various workloads and TM implementations. My investigation will focus on digging on the main sources of inefficiency of these solutions, when deployed in TM environments and to identify solutions to them. In particular, I plan to focus on sources of inefficiencies in HTM environments, which are definitely the ones to possess more potential, but also are the more susceptible to performance pathologies due to hardware restrictions.

4 Evaluation

For my evaluation I will be focusing on a number of metrics to ensure a good analysis of the system. The metrics I will be using are such as:

1. throughput - essential to determine the performance of the overall system.
2. response time - needed to evaluate the time it takes to the system respond to its queries.
3. energy consumption - a good metric to compare TM implementations and fine-grained locks.
4. abort rate - used to check if my TM implementations are having big overheads. The reason of the aborts (e.g., capacity or conflict) will also help me identify the errors in the program.
5. commit rate - can be compared with abort rate to verify whether transactions are manly committing or aborting.
6. lock rate - used to check how many transactions are using the fall-back path.

To use all these metrics I will be running different benchmarks in my solution. There are two different possible types of traces that benchmarks use, recorded

traces that use real data, made from analysis of events like traffic, providing a very realistic job stream and, synthetic traces that attempt to capture the behaviour of observed workloads and that have the advantage of isolating specific behaviours not clearly expressed in recorded traces.

One of the benchmarks I will use made by Thomas Brinkoff [37], has a special feature that usual spatio-temporal benchmarks do not consider. Moving objects like traffic usually have a specific path they must follow and interact with each other while following the path's rules. With this in consideration [37] uses a network to simulate the paths where moving objects can go threw. The network is used to simulate real world and it combines real data (the network) with user-defined specifications of the properties of a real data dataset.

Another benchmark I will use is MOTO (Moving Objects Trace generator). MOTO is a trace generator for the moving objects application domain, a spatio-temporal data benchmark. Its main strength is scalability for large networks (from Brinkoff) and many moving objects.

The final benchmark that I currently plan to use is of using is COST [38]. This benchmark is made specifically for the performance evaluation and comparison of spatio-temporal indexes. Thus COST is a great benchmark for my solution. COST also provides near-future uncertain positions of moving objects, granting more realism to benchmarks.

5 Work Plan

My plan of work will follow the schedule:

1. June 1 - June 14: adaptation of uGrid to use HTM.
2. June 15 - June 31: adaptation of uGrid to use STM.
3. July 1 - July 14: adaptation of uGrid to use HyTM.
4. July 15 - July 31: benchmarking and profiling of uGrid.
5. August 1 - August 14: adaptation of pGrid to use HTM.
6. August 15 - August 31: adaptation of pGrid to use STM.
7. September 1 - September 14: adaptation of pGrid to use HyTM.
8. September 15 - September 31: benchmarking and profiling of pGrid, comparison with fine-grained locking.
9. October 1 - November 14: optimizing/redesigning uGrid and pGrid to best perform on HTM.
10. November 15 - December 14: Thesis writing.

6 Conclusion

The relevance of spatio-temporal data applications and the volume and velocity of such type of data has dramatically increased, over the last few years, thanks to the proliferation of GPS equipped devices. The problems of developing indexes for spatio-temporal queries is well-known and several have been proposed in literature [7, 10]. In my thesis, I intend to study efficient ways to enable concurrent access to spatio-temporal data indexes, in order to take advantage of modern multi and many core architectures.

As reviewed in this report, TM has emerged as a promising abstraction for parallel programming. I aim to study how to accelerate concurrent indexes for spatio-temporal data using TM, focusing on two state of the art solutions for spatio-temporal data indexing [4, 7]. I also intend to evaluate different TM implementations (hardware, software and hybrid), and carry out an experimental study aimed at assessing TM competitivity in these workload scenarios, and whether it is possible to suggest enhancements to current data structures to best fit the restrictions of current HTMs. Finally, I have presented the evaluation methods and the work plan.

7 Bibliography

1. K. Cukier, *Data, data everywhere: A special report on managing information*. Economist Newspaper, 2010.
2. M. Selinger and L. Schmidt, "Adaptive traffic control systems in the united states," *HDR Engineering, Inc*, 2009.
3. A. Khanna, "Facebook's privacy incident response: a study of geolocation sharing on facebook messenger,"
4. D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys, "Trees or grids?: indexing moving objects in main memory," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 236–245, ACM, 2009.
5. R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: a comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 546–557, ACM, 2002.
6. D.-T. Lee and C. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23–29, 1977.
7. D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Processing of extreme moving-object update and query workloads in main memory," *The VLDB Journal*, vol. 23, no. 5, pp. 817–841, 2014.
8. C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient b+-tree based indexing of moving objects," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 768–779, VLDB Endowment, 2004.
9. S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1, pp. 7–16, IEEE, 2011.

10. S. Li, S. Hu, R. Ganti, M. Srivatsa, and T. Abdelzaher, "Pyro: a spatial-temporal big-data storage system," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 97–109, 2015.
11. D. Makreshanski, J. Levandoski, and R. Stutsman, "To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1298–1309, 2015.
12. A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *ACM sigplan notices*, vol. 44, pp. 155–165, ACM, 2009.
13. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 48–60, ACM, 2005.
14. J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
15. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, pp. 10–10, 2010.
16. V. Reddy, M. Aron, V. Gupta, and M. Thomas, "Distributed key-value store," Dec. 31 2015. US Patent 20,150,379,009.
17. H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.
18. M. Bader, *Space-filling curves: an introduction with applications in scientific computing*, vol. 9. Springer Science & Business Media, 2012.
19. S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance evaluation of main-memory r-tree variants," in *Advances in Spatial and Temporal Databases*, pp. 10–27, Springer, 2003.
20. R. Nystrom, *Game programming patterns*. Genever Benning, 2014.
21. H. S. Nagesh, S. Goil, and A. N. Choudhary, "Adaptive grids for clustering massive data sets.," in *SDM*, pp. 1–17, SIAM, 2001.
22. J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *IFIP Working Conference on Modelling in Data Base Management Systems*, pp. 365–394, 1976.
23. M. Herlihy and J. Moss, "Transactional memory: architectural support for lock-free data structures. sigarch comput. archit. news 21 (2), 289–300 (1993)."
24. R. Guerraoui and M. Kapalka, "Opacity: A correctness condition for transactional memory," tech. rep., 2007.
25. T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 144–157, ACM, 2015.
26. R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pp. 1–11, IEEE, 2013.
27. N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 3–14, ACM, 2014.
28. A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 127–136, ACM, 2012.

29. C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for ibm system z," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 25–36, IEEE, 2012.
30. P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," 2011.
31. I. P. I. Version, "2.07," 2013.
32. L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: streamlining stm by abolishing ownership records," in *ACM Sigplan Notices*, vol. 45, pp. 67–78, ACM, 2010.
33. R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pp. 258–264, ACM, 2005.
34. L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear, "Transactional mutex locks," in *Euro-Par 2010-Parallel Processing*, pp. 2–13, Springer, 2010.
35. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ACM Sigplan Notices*, vol. 41, pp. 336–346, ACM, 2006.
36. L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 39–52, 2011.
37. T. Brinkhoff, "Generating network-based moving objects," in *Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on*, pp. 253–255, IEEE, 2000.
38. C. S. Jensen, D. Tiešytė, and N. Tradišauskas, "The cost benchmark—comparison and evaluation of spatio-temporal indexes," in *Database Systems for Advanced Applications*, pp. 125–140, Springer, 2006.