Hindawi
Complexity
Volume 2021, Article ID 6645778, 11 pages
https://doi.org/10.1155/2021/6645778

Research Article

An Adaptive Parallel Method for Indexing Transportation Moving Objects

Kun-lun Chen,[1] Chuan-wen Li,[1] Guang Lu,[1] Jia-quan Li,[1] and Tong Zhang[2]

[1]*School of Computer Science & Engineering, Northeastern University, Shenyang 110819, China*
[2]*State Grid Dalian Electric Power Supply Company, Dalian, China*

Correspondence should be addressed to Chuan-wen Li; lichuanwen@mail.neu.edu.cn

Received 28 December 2020; Revised 26 January 2021; Accepted 3 February 2021; Published 22 February 2021

Academic Editor: Rui Wang

Transportation cyber-physical systems are constrained by spatiality and real-time because of their high level of heterogeneity. Therefore, applications like traffic control generally manage moving objects in a single-machine multithreaded manner, whereas suffering from frequent locking operations. To address this problem and improve the throughput of moving object databases, we propose a GPU-accelerated indexing method, based on a grid data structure, combined with quad-trees. We count object movements and decide whether a particular node should be split or be merged on the GPU. In this case, bottlenecked nodes can be translated to quad-tree without interfering with the CPU. Hence, waiting time of other threads caused by locking operations raised by object data updating can be reduced. The method is simple while more adaptive to scenarios where the distribution of moving objects is skewed. It also avoids shortcomings of existing methods with performance bottleneck on the hot area or spending plenty of calculation resources on structure balancing. Experiments suggest that our method shows higher throughput and lower response time than the existing indexing methods. The advantage is even more significant under the skewed distribution of moving objects.

1. Introduction

In modern economic infrastructure, a very important part is the transportation network. It connects cities, manufacturers, retailers, and nations by moving large volumes of freight and passengers through a complex network [1] (highway, railways, and so on). For better utilizing this network, the transportation system generally consists of the physical and cyber system (i.e., transportation cyber-physical systems, TCPSs) to provide innovative services. However, TCPS with mobile devices presents additional complexity because of the uncertainty of the object (vehicles and mobile devices). Many important applications such as traffic control and autonomous driving relay on the moving object satabase (MOD) [2], whose main goal is to provide users with a certain range of query results of mobile objects under the premise of meeting certain time accuracy and spatial accuracy. For example, in taxi applications, mobile users and taxis are mobile objects, and recommending a taxi within a certain distance for users is a typical mobile object query operation [3].

TCPS, in general, demonstrates a high level of heterogeneity, including sensor nodes, mobile devices, high-end workstations, and servers. The different components of TCPS probably have a nonuniform granularity of time and spatiality, TCPS is constrained by spatiality and real-time [4], and traditional single-threaded spatial data management methods are difficult to meet practical application requirements in terms of efficiency for massive mobile object data. Although considerable data processing platforms such as Hadoop and Spark have the advantage in dealing with massive data, their platform structures are mostly distributed and thus required for mutual communication between nodes. The time cost of these platforms cannot meet the real-time requirements of mobile spatial data management. Therefore, mobile object management is currently implemented in a single-machine multithreaded manner.

The facets of TCPS mentioned above require MOD to process large amounts of updates and queries in real-time, and thus spatial indexes are generally used to enhance its performance. There are currently two major categories of mobile object indexing structures: tree-based index and grid-based index. In the beginning, most of the indexes resided in disks since the index itself can be too large to fit in memory. Thus, tree-based indexes [5–7] are more popular structures because of sophisticated improvements that reduce disk I/O. Recently, the rapid development of computer hardware has made low-cost and high-capacity main memory capable of processing millions of moving objects. In fact, due to the high frequency of object position changes, the in-memory data structure for storing moving objects is much better than the on-disk one to improve IO efficiency [8]. Therefore, many works have been done to extend tree-based indexes to in-memory variants, such as [9–11]. However, the tree-based index update operation is specially constructed to reduce disk I/O, thus complicated and time-consuming.

On the other hand, compared with the tree-based index covering the space with its leaf nodes, the grid-based index divides space into uniform grids of uniform size. In the in-memory environment, these simple uniform grids are easy to update and maintain and thus more efficient than its tree-based competitor. Šidlauskas et al. [9] proposed u-Grid, an update efficient grid-based structure where a secondary index is employed to support the bottom-up updates. Xu et al. [12] proposed D-grid that takes advantage of the velocity information for further improving query performance. They also proposed the lazy deletion and garbage cleaning mechanism for accelerated update processing. Šidlauskas et al. [13] proposed PGrid, a parallel main memory indexing technique that supports heavy location-related query and updates operation by exploiting the parallelism of modern processors.

The grid-based index structure indeed is simple and easy to implement, but it is not suitable for uneven distribution, which is widespread in real-world applications [14]. For example, in the traffic monitoring applications, downtown compared with suburban (or in morning and evening peaks compared with other periods), the load on mobile objects' space and time is uneven. Although the tree-based index can make every leaf node contain approximately the same number of moving objects, the experimental results show that its performance is not as good as grid-based indexing [9].

In fact, the performance bottleneck of single-machine multithreaded mobile object management in main memory is no longer the I/O operation but the delay caused by the coordination among multiple threads, the most crucial influence of which is thread's locking operation on data structure on other threads. Reducing locking operations on other threads is the research direction of mobile object management. We observe that the objects need to be locked only when entering and leaving the leaf nodes. Thus, the division of leaf nodes in the tree index should not be determined by the total number of moving objects but the number of objects entering and leaving the leaf nodes in a unit time. However, counting the incoming and outgoing objects involves atomic operations, which will frequently lock the counters. We also need to decide whether to split the leaf nodes or merge the adjacent leaf nodes according to the changing count results that require continuous calculations, which will seriously affect the efficiency of object data update. Therefore, the existing research does not adopt the index method of dividing the leaf nodes by the entry and exit object counts [15].

Therefore, this paper proposes an adaptive parallel GPU-accelerated transportation moving object indexing method, which uses a grid-integrated quad-tree structure and in which GPU process counts the number of incoming and outgoing objects of leaf nodes and whether the leaf nodes need to be split or merged. This method only occupies a small amount of CPU computing resources and achieves the goal of continuously optimizing the index structure without affecting data update and query efficiency. The experimental results show that the performance of this index method is better than that of the best existing method for moving object updating.

2. Index Structure

2.1. Problem Definition. Given space plane S, moving object set $O = \{o_1, \ldots, o_n\}$, any of them $o_i = \{o_i^{id}, o_i^x, o_i^y, o_i^t\}$, o_i^{id} is the unique identifier of o_i, (o_i^x, o_i^y) is its location, and o_i^t is the last update time. Query operation set $Q = \{q_1, \ldots, q_e\}$, in which any query request $q_j = \{x_{\min}, y_{\min}, x_{\max}, y_{\max}, t_q\}$. The first four items define a rectangular query box and t_q is the time when the query started. The purpose of moving the object database is to return the mobile object located in the query box to the user when the query q_j arrives in Q.

This query method is called Range Query. Since other types of queries such as kNN queries can be converted into a series of range queries, this article only needs to discuss the support of the index structure for such queries.

2.1.1. Index Structure Design. The main task of moving the object database is to constantly update the position of the moving object and return the result according to the query requirement. Therefore, the mobile object index structure needs to meet the two basic conditions:

(1) Find object by object identification o_i^{id}

(2) Find and update moving objects based on object position (o_i^x, o_i^y)

An auxiliary index based on the hash table can be used to support condition (1).

Definition 1. Auxiliary index: all spatial objectsusing hash table $\mathscr{H}$ according to o_i^{id} d save key-value pairs like (o_i^{id} *p_bkt*, *idx*) in $\mathscr{H}$. *p_bkt* is the location of the memory space of the bucket in which o_i is located in the hybrid index (see Definition 2). *idx* represents its relative position in the bucket.

Figure 1 shows an example of an auxiliary index. The nature of the hash table shows that using an auxiliary index

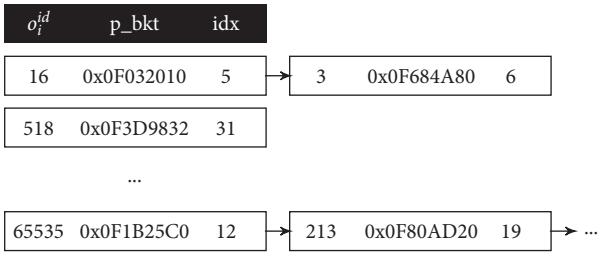

Figure 1: An example of the auxiliary index.

can find the memory location of o_i in memory in a constant time based on o_i^{id}.

Both grid-based and tree-based indexes have advantages and disadvantages in satisfying condition (2). The grid-based indexing method directly calculates the grid to which o_i belongs according to (o_i^x, o_i^y). However, when the moving object space is not uniformly distributed, the number of moving objects is excessive in the grid of hotspot region, and updating the location information of the object will cause the grid in the hotspot region to be locked frequently, which reduces the parallel performance of the hotspot region grid.

Tree-based indexing reduces the congestion of objects in hotspots. However, searching o_i within (o_i^x, o_i^y) requires multiple queries from tree root nodes to a series of intermediate nodes up to leaf nodes, which is inefficient. The more significant effect on overall efficiency is that tree-like indexes need constantly adjusting the structure to fit the distribution of moving objects. Computation of whether leaf nodes needs to be adjusted, and adjust operations themselves will consume a large number of computing resources.

This paper proposes a hybrid indexing method combining grid and quad-tree to avoid the disadvantages of the above two methods.

Definition 2. Hybrid index P divides the space plane S into $G_{\mathrm{num}} = 2^\rho \times 2^\rho$ grids, each of which can be converted between grid nodes and quad-trees depending on the conditions.

Hybrid index balances the load of each cell by transforming the grid of the hotspot region into a quad-tree [16]. ρ is the gridding parameter, using the selection condition given by the document [17]:

$$\rho = \frac{1}{2}\left[\log N - \log C_L\right], \tag{1}$$

where N represents the total number of moving objects in space and C_L represents the capacity of the leaf node.

During execution, the number of moving objects in each grid is dynamically determined, and the grid that satisfies the split requirement is converted into a quad-tree, and the quad-tree that meets the merge requirement is converted back to the grid. Within each quad-tree, its leaf nodes are also split or merged according to conditions.

Figure 2 shows an example of a hybrid index where the uppermost layer in the right half is a plane S that is divided into $G_{\mathrm{num}} = 2^\rho \times 2^\rho$ grids, where the black grid represents the hot grid that is converted to a quad-tree, and the lower

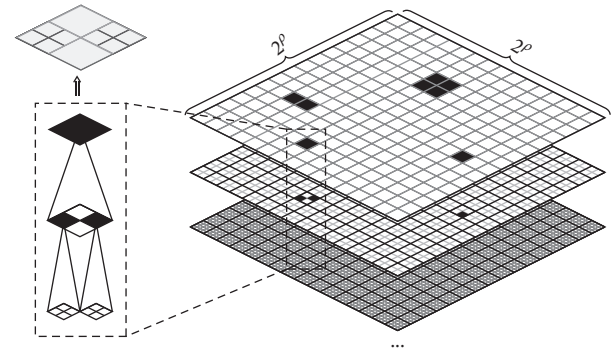

Figure 2: An example of the hybrid index.

layers represent the corresponding spatial regions of the quad-tree nodes at various levels, where the black nodes represent further subdivided regions. The left part of the figure shows one of the zoomed-in quad-trees. The top of the figure shows the quad-tree division to the grid. After converting to a quad-tree, the region is divided more finely.

Each grid node c_i contains only one p_{bucket} pointer to a bucket list $\mathscr{L}_i$. This list contains a series of buckets that hold a fixed number of all moving objects belonging to the node. Each quad-tree node ξ_i is represented as $\xi_i = \{p_{\mathrm{bucket}}, p_{c1}, p_{c2}, p_{c3}, p_{c4}\}$, where p_{c1}, p_{c2}, p_{c3}, and p_{c4} are pointers to a child node in the four quadrants with the current node as a parent.

2.1.2. Node Split Conditions. Since the splitting and merging operations on the grid are essentially the same as those on the quad-tree leaf nodes, we use the term nodes for both conditions.

When an object moves inside a node c, we only need to lock the object itself for updating its position without locking the node. When the moving object enters and exits the node c, the information of this object must be appended or deleted in a bucket corresponding to c and c must be locked simultaneously to prevent data collisions caused by different threads concurrently updating the c. Therefore, the update operation of all other threads to node c needs to be suspended. The greater the number n_c of moving objects into and out of c per unit of time, the more threads waiting at the same time. The total time required for the update operation φ_c is the sum of the actual execution time of the operation $n_c\tau$ (where τ is the execution time of each update operation) and the waiting time ψ_c. Therefore, there is a positive correlation between total update operation time and the number of moving objects in and out of the nodes.

By splitting the nodes, we can reduce the number of simultaneous waiting threads within the node. However, after splitting a node c into nodes, the movement of objects within a node c may become a movement among multiply nodes, increasing the total number of moving objects in and out of the nodes, i.e.,

$$n_c \le n_{c1} + n_{c2} + n_{c3} + n_{c4}. \tag{2}$$

Therefore, to reduce φ, it requires a moderate split of the node, only when

$$\varphi_c > \varphi_{c1} + \varphi_{c2} + \varphi_{c3} + \varphi_{c4}. \tag{3}$$

Splitting the nodes helps to improve update performance. Conversely, when the four split nodes c_1, c_2, c_3, and c_4 satisfy the condition $\varphi_{c1} + \varphi_{c2} + \varphi_{c3} + \varphi_{c4} > \varphi_c$, they need to be merged into one node.

This section will quantify the relationship between φ and n. We will first discuss the effect of each additional update on the total waiting time for moving objects.

Lemma 1. *Set the execution time of the update operation required for each node to enter and exit as τ and the number of moving objects in or out of the node per unit time as n_c ($n_c > 1$). Mark the expected value of the total waiting time required for updating n_c moving objects entry and exit information as $\psi(n_c)$, and then update the expected value $\psi(n_c + 1)$ of the waiting time for $n_c + 1$ moving objects as follows:*

$$\psi(n_c + 1) = \psi(n_c) + \frac{3}{2} n_c \tau^2 + 4(n_c - 1)\tau^3. \tag{4}$$

Proof. Assume that node c has n_c objects into and out of the node within a unit time. Then, the $n_c + 1$ th object cannot move in or out of its cell until the previous $n_c + 1^{\text{th}}$ objects finish updating, which is denoted by the following:

$$\psi(n_c + 1) = \psi(n_c) + \phi(n_c).(n_c \geq 1), \tag{5}$$

where $\phi(n_c)$ is the correlation function between waiting time and the number of moving objects. There are two types of update waiting time among the n_c objects. The overlapping update waiting time means that an object's update start time point appears within the period when another object is performing an update operation, whereas the nonoverlapping update waiting time means that the update start time point of an object occurs before the update start point of another object. The interval between two update start time points is greater than τ. This ensures that two objects are updated without waiting for each other. The significance of the correlation function $\phi(n_c)$ is the relationship between overlapping and nonoverlapping of object updates.

When the update waiting time of n_c objects is not overlapping, we mark the probability that the $n + 1$ object falls within the object execution period as p_1', and $p_1^{"}$ is the probability that $n + 1$ object falls before the time τ before an object updates time in time.

If $p_1' = n_c$, the waiting time at this time is the integral of τ on $[0, \tau]$, i.e.,

$$w_1' = \int_0^\tau \tau \mathrm{d}\tau. \tag{6}$$

If $p_1'' = n_c \times \tau$, the waiting time at this time is the integral of the constant 1 on $[0, \tau]$, i.e.,

$$w_1'' = \int_0^\tau 1 \, d\tau. \tag{7}$$

Finally, the waiting time between n_c objects can be expressed as $w_1 = p_1' \times w_1' + p_1'' \times w_1''$.

When the update waiting time is overlapping among n_c objects, the overlapping time will need to be postponed. The waiting time for the overlapping part is set as w_2, and the execution time of the object is τ. Each object can be overlapped for a period of time τ, so the period that can be overlapped is $o_t = 2\tau$.

The postponed period is denoted as w_2', the integral of the overlapped period o_t on $[0, \tau]$:

$$w_2' = \int_0^\tau 2\tau \, d\tau. \tag{8}$$

There are only overlapping periods among n_c objects, and the postponed period among n_c overlapping objects is

$$w_2' = (n_c - 1) \int_0^\tau 2\tau \mathrm{d}\tau. \tag{9}$$

The concept of overlap between two objects shows that each time the object can be overlapped is denoted as $o_t = 2\tau$. The overlapping probability between two objects is

$$p_2' = 2 \times o_t = 4\tau. \tag{10}$$

The overlapping waiting time w_2 between n_c objects can be expressed as

$$\begin{aligned} w_2 &= p_2' \times w_2', \\ \phi(n_c) &= w_1 + w_2. \end{aligned} \tag{11}$$

Substituting the above formula into it, we get

$$\begin{aligned} \phi(n_c) &= p_1' \times w_1' + p_1'' \times w_1'' + p_2' \times w_2' \\ &= n_c \int_0^\tau \tau \mathrm{d}\tau + n_c \tau \int_0^\tau 1 \mathrm{d}\tau + 4\tau(n_c - 1) \int_0^\tau 2\tau \mathrm{d}\tau \\ &= \frac{3}{2} n_c \tau^2 + 4(n_c - 1)\tau^3. \end{aligned} \tag{12}$$

Finally, $\psi(n_c + 1) = \psi(n_c) + (3/2)n_c\tau^2 + 4(n_c - 1)\tau^3$.

The closed form of the waiting time $\psi(n)$ can be derived from Lemma 1. □

Lemma 2. *Set the number of moving objects into and out of a node per unit time as n and the update execution time of each object as τ. Then, the total waiting time $\psi(n)$ for n update operations is as follows:*

$$\psi(n) = \frac{3}{4} n(n-1)\tau^2 + 2(\mathrm{n} - 1)(\mathrm{n} - 2)\tau^3. \tag{13}$$

Proof. According to Lemma 1, the update waiting time of a moving object is related to the number of moving objects and the update time point of the object. Formula (4) can be recursively expanded as

$$\begin{aligned}\psi(n) &= \frac{3}{2}(n-1)\tau^2 + 4(\mathrm{n}-2)\tau^3 + \psi(n-1),\\ \psi(n-1) &= \frac{3}{2}(n-1)\tau^2 + 4(\mathrm{n}-3)\tau^3 + \psi(n-2),\\ \psi(2) &= \frac{3}{2}\tau^2 + \psi(1),\\ \psi(1) &= 0.\end{aligned} \tag{14}$$

Substituting $\psi(1), \ldots, \psi(\mathrm{n})$ into the formula, we get

$$\psi(\mathrm{n}) = \sum_{n=2}^{n-1}\frac{3}{2}n\tau^2 + \sum_{n=2}^{n}4(n-2)\times\tau^3. \tag{15}$$

So, Lemma 2 was proved.

Lemma 2 describes the relationship between the object update waiting time and the number of object updates, and the total update time φ is the sum of the update waiting time ψ and the actual execution time $n\tau$. □

Theorem 1. *Set the number of moving objects to and from the node per unit time as n. The update execution time of each object is τ. Then, the total update time $\varphi(n)$ including node waiting for other thread per unit of time can be expressed as follows:*

$$\varphi(n) = n\tau + \frac{3}{4}(n-1)\tau^2 + 2(n-1)(n-2)\tau^3. \tag{16}$$

Proof. Substitute (5) into $\varphi(n) = n\tau + \psi(n)$. Then, Theorem 1 can be proved. □

3. Data Structures and Algorithms

This section uses the C++-like pseudocode to describe our index structure. The index structure is divided into two parts: the auxiliary index and the hybrid index, in which the auxiliary index $\mathscr{H}$ adopts a hash table; i.e., the form is

$$\mathrm{unordered}_{\mathrm{map}}\langle \mathrm{int}, \mathrm{pair}\langle \mathrm{Bucket}*, \mathrm{int}\rangle\rangle. \tag{17}$$

We use this structure to store (o_i^{id}, p_bkt, idx) key-value pairs. The hybrid index takes the form of a 2-D array combined with a quad-tree. Next, we will focus on the hybrid index data structure and its associated algorithms.

The hybrid index data structure includes grid index and QuadTree index, which balances the update load of each cell according to the conditions given by Theorem 1. It merges cells containing a small number of objects to reduce the cost of querying quad-trees. It also split cells that contain larger numbers of objects to reduce update waiting time.

From Section 2.1.2, we know that continuously comparing the relationship between the total execution time φ_c of the node c and the total execution time of its four child nodes $\varphi_{c1} + \varphi_{c2} + \varphi_{c3} + \varphi_{c4}$ is crucial to deciding whether to split or merge a node. Since these comparison operations consume massive computing resources, the indexing method performs these operations on the GPU. Although counting moving objects in or out of nodes does not require much calculation, each time increased, the counter needs to be locked, which seriously impacts the overall parallel performance. Therefore, the counting operation needs to be performed on the GPU.

This chapter introduces the indexing algorithm from two aspects: CPU and GPU. All algorithms are based on a unified data structure stored in the main memory. The data required for GPU operation are copied from main memory to memory only when used. After the calculation result is obtained, the data are copied back to the main memory.

4. Data Structure

The grid index in a hybrid index is implemented using a two-dimensional array:

$$\mathrm{array}\langle \mathrm{array}\langle \mathrm{unique}_{\mathrm{ptr}}\langle \mathrm{Node}\rangle, \mathrm{width}\rangle, \mathrm{height}\rangle. \tag{18}$$

where width and height represent the number of grid columns and rows, respectively. The Node class is the parent of Cell and quad QuadTree. Cell represents a grid that contains only one pointer of type $\mathrm{unique}_{\mathrm{ptr}}\langle \mathrm{Bucket}\rangle$, and QuadTree represents a quad-tree:

```
Struct QuadTree{
unique_ptr⟨Bucket⟩p_bucket;                    / * Bucket Chain Header * /
  array⟨unique_ptr⟨QuadTree⟩, 4⟩children;      / * Child nodes * /
  QuadTree * parent;                           / * Parent nodes * /
  int left, right, floor, ceiling,             / * Coverage * /
}.
```
(19)

Cell or Objects in QuadTree are stored in a linked list bucket. Buckets are used to store moving objects on leaf nodes. The size of the buckets is fixed. The number of moving objects determines the number of buckets of each leaf node. If the bucket is full when the object is inserted, a new bucket is created; if the bucket becomes empty when the object is deleted, the bucket is deleted. The bucket structure is as follows:

```
struct Bucket{
   Site[Max × _size]sites;          / * Array of objects * /
   int current,                     / * The number of objects in the bucket * /
   unique_ptr⟨Bucket⟩bucket,        / * Next bucket * /
}.
```
(20)

The Site class holds moving object data, including id, x, y and updates time t_u. It can be inferred from the contents of the Site class that the data of one object needs at least 128 bits of memory space (four int values on a 32 bit machine). An access violation may occur between different threads when reading and writing Site classes in parallel without protection. The traditional way to avoid an access violation is to lock it while reading and writing Site. Since Site is the most frequently used class in the index, in order to avoid the impact of locking on performance, we merge the four data in the Site class into an object of type _m128i, use _mm_load_si128, and _mm_set_epi32 operation in the Intel MMX instruction set [18] to read and write the content and use _mm_extract_epi32 to extract the corresponding data. In this way, the index can correctly read and write Site data without locking.

The index also maintains two ⟨Node *⟩ lists: split_candicates list and merge_candidates list. Both save nodes need splitting and need merging separately. We calculate the nodes that need splitting or merging on the GPU.

4.1. CPU Algorithm. As the object continues to update, the structure of the quad-tree will change. The object's insertion algorithm inserts it into the appropriate node based on its coordinates. The deletion algorithm finds the bucket based on the object id and deletes it. The division or merging of cells is a crucial operation for balancing quad-trees. Only cells that meet the conditions for splitting and merging can be divided and merged.

4.1.1. Spatial Object Insertion Algorithm. During the movement of the object, as the position changes, the object will continue to move between nodes. The main purpose of Algorithm 1 is to insert an object into the leaf node. The idea is to find the leaf node to which the object should be inserted based on the position of the object (o_i^x, o_i^y) and determine the bucket state of the node. If the bucket is not full, insert the bucket directly. If the bucket is full, create a new bucket n_bucket and insert the new bucket into the bucket list of the current node. Then, insert the object o_i into the bucket and increase the number of objects stored in the bucket by one.

4.1.2. Spatial Object Deletion Algorithm. When the location of a moving object is updated, objects that no longer belong to the current leaf node range need inserting into its new belonging leaf node and deleted from the current node.

As shown in Algorithm 2, it finds the bucket where the object is located according to the object that uniquely identifies (o_i^{id}). The object is then removed from the bucket, and the number of objects stored in the bucket is decremented. After deleting the object o_i, if the bucket is empty, the bucket will also be deleted.

4.1.3. Cell Partitioning Algorithm. According to the node splitting condition in Section 2.1.2, when the leaf node satisfies the condition $\varphi_c > \varphi_{c1} + \varphi_{c2} + \varphi_{c3} + \varphi_{c4}$, the Quad-Tree index structure should divide the leaf node s_node into four equal-sized subgrids, where x_middle = (left + right/2) and y_middle = (floor + ceiling/2) (lines 1-2). After the division is completed, all children's parent nodes are set as the current leaf nodes, and all the objects in the divided node s_node are moved to the bucket of the corresponding child. If the parent of the partitioned node belongs to the split_candidates list before the node is unpartitioned, it will be removed from the list split_candidates and appended the current node to split_candidates list as shown in Algorithm 3.

4.1.4. Cell Merge Algorithm. According to the node split merge condition in Section 2.1.2, when the leaf node satisfies the condition $\varphi_{c1} + \varphi_{c2} + \varphi_{c3} + \varphi_{c4} > \varphi_c$, QuadTree index structure merges four grids into one. The purpose is to adjust the quad-tree structure and balance the update load to improve the efficiency of the update query.

As shown in Algorithm 4, when a node satisfies the merge condition, buckets of all children of the node are linked and assigned to the bucket of the current node. Adjust the bucket chain of the current node and delete the empty bucket. If the current node belongs to the merge_candidates list, the current node m_node is deleted from merge_candidates. After merging, if the parent node of the current node belongs to the merge_candidates list, the parent node m_node.parent of the current node is added to merge_candidates.

4.2. GPU Algorithm. GPU's logic computation ability is weaker than that of the CPU, so it is not suitable for making complex logical decisions and is more suitable for a large amount of parallel data calculation. Different from CPU, GPU adopts a SIMD (single-instruction, multiple-data) operating mode. The same instruction can be executed on multiple sets of data at the same time so as to improve the efficiency of parallel execution [19]. Therefore, the work that is handed to the GPU in the index is mainly composed of two parts that involve large data amount calculations: (i) counting the number of moving objects entering and leaving the node and (ii) deciding whether the nodes need to be split or merged. Both operations involve

```
Input: move the object o_i = {o_i^id, o_i^x, o_i^y, o_i^t}
Output: no output; the leaf node is updated after the operation is completed
(1)  cur_leaf = get_leaf (o_i^x, o_i^y) /* find the inserted leaf node by location (o_i^x, o_i^y) */
(2)  if (is_fill(cur_leaf.p_bucket))
(3)     n_bucket = new Bucket();
(4)     insert_bucket (n_bucket);
(5)  Insert_object(o_i); /* insert the object o_i into the bucket of the current node */
```

Algorithm 1: Object o insert leaf node add_to_leaf.

```
Input: move the object o_i = {o_i^id, o_i^x, o_i^y, o_i^t}
Output: no output; the leaf node is updated after the operation is completed
(1)  bucket = get_bucket (o_i^id);
(2)  delete_from_bucket (o_i);
(3)  if (is_empty(bucket))
(4)     delete_bucket(bucket); /* delete empty bucket */
```

Algorithm 2: Deleting object from leaf node O remove_from_leaf.

```
Input: quad-tree leaf node
s_node = {p_bucket, children, parent, left, right, floor, ceiling}
Output: no output; quad-tree structure changes after the operation
(1)  x_middle = (left + right)/2;
(2)  y_middle = (floor + ceiling)/2;
(3)  s_node.children[0] = new QuadTree (x_middle, right, y_middle, ceiling; /* in a similar way to initialize s_node.children [1–3] */
(4)  for each (child in s_node.children)
        child.parent = s_node;
           s_node.p_bucket. o_i ⟶ child.p_bucket;
(5) if (s_node.parent in split_candidates)
        delete_from_split_candidates (s_node.parent);
(6)  insert_split_candidates (s_node);
```

Algorithm 3: Cell division split.

```
Input: quad-tree node: m_node = {p_bucket, children, parent, left, right, floor, ceiling}
Output: no output; quad-tree structure changes after the operation
(1)  if(for each m_node, children is_leaf()) /*all children with the node m_node are leaf nodes*/
(2)     m_node.p_bucket ← child.p_bucket;
(3)     delete _null_bucket (m_node.p_bucket);
(4)     if (m_node in merge_candidates)
(5)        delete _from_merge_candidates (m_node)
(6)     if (for each m_node.parent.child is_leaf())
(7)        insert _merg_candidates (m_node.parent);
```

Algorithm 4: Cell merge merge.

two ⟨Node ∗⟩ type lists split_candidates and merge_candidates, which are maintained in the index.

4.2.1. Counting Algorithm. We take counting objects entering the node as an example to describe the counting algorithm in this section. If the operation runs on the CPU, several threads are usually opened according to the parallel capability of the CPU, and the following processes are executed, respectively, Algorithm 5 is as follows.

update_info_list stores a series of object movement data received by the system, including the object id, original location, and new location. Each CPU thread handles part of the update_info_list. After the node where the object is moved is calculated by get_location, and the third row increments the shifted-in node counter. Note that the counter

```
Input: partial object movement information linked list update_info_list
Output: update each node counter value
(1)   for each (item in some part of update_info_list)
(2)     node_in = get_location (item.new_pos);
(3)     lock_and_increase (node_in);
```

Algorithm 5: CPU-based counting algorithm CPU_Count.

must be locked when incrementing to ensure correct reading and writing of counters in parallel.

Compared with the CPU, GPU generally has thousands of stream processors [13], each of which maintains a fixed number of counters for several nodes. Since each counter can only be accessed by a fixed stream processor, it does not need to be locked when incrementing. Algorithm 6 demonstrates that each stream processor is responsible for $m \times n$ node operation flow.

threadIdx is a built-in stream handler index value automatically set by the system. Count array is a local array of stream processors and saves the node counters it is responsible for. Line 6 determines whether the node belongs to the node that the stream processor is responsible for, and if it belongs to the stream processor, it increments the corresponding counter (line 7).

The indexing system not only calls GPU_count on all nodes in the lists split_candidates and merge_candidates to record the number of objects in or out of nodes but also calls GPU_count on four child nodes that each node may divide in the list split_candidates and parent nodes that may be merged by four neighboring nodes in the list merge_candidates.

4.2.2. Split Merge Judgment Algorithm. We take the split judging algorithm as an example. Because the indexing system has counted any node c in the split_candidates list and the four child nodes c_1, c_2, c_3, and c_4 in which it may be split into, we just need to assign nodes in the split_candidates list to each stream handler and then substitute equation (16) into (3) to make the decision.

5. Simulation Experiment and Result Analysis

This section compares the index structure, denoted as GAPI with PGrid. PGrid is the state-of-the-art parallel moving object index structure and is widely adopted in applications for indexing moving objects [13].

The experimental simulation environment is Win 10 system, Intel Xeon E5-2620 v3 six-core CPU × 2, NVIDIA Quadro K2200 GPU. The experimental code is implemented by *C++* and CUDA Toolkit 7.5, and the spatial object data are generated by the open-source mobile object generation tool MOTO (http://moto.sourceforge.net) based on the Brinkhoff [20] algorithm. We set the parameter of GAPI as $\rho = (1/2)[\log N - \log C_L]$ according to equation (1), where N represents the total number of moving objects in space and C_L represents the capacity of the leaf node. Experimental parameters are shown in Table 1.

Figure 3 shows the experimental results of the effect of the update/query ratio on throughput. With the increase in the update/inquiry ratio, the throughput of the two index structures increases. Among them, our method's throughput is higher than that of PGrid in most cases but is only lower when the update/query is small. This is because our method mainly optimizes the update operation. The GPU-assisted splitting and merging of the quad-tree reduce the thread waiting time in the parallel update. Therefore, the average update time is better than PGrid. Because the query on the quad-tree structure needs multiple queries from the root node to the leaf nodes, our index's query time is generally higher than that of the PGrid.

Figure 4 shows the experimental results of the effect of query area on throughput. From the figure, our method is better than PGrid in all cases. With the increase in the area of the query area, the overall throughput is declining. This is because the query operation occupies more and more computing resources.

Figure 5 shows the experimental results of the effect of the update interval on throughput. The longer the update interval, the greater the probability that the object moves out of the current node. When the renewal interval gradually increases and the PGrid throughput decreases drastically, the throughput of our method can still be stable. This is because our method can dynamically adjust the nodes according to the actual situation at running time. We can also see that the dropping speed of GAPI is getting slower as update intervals increase. The reason is that a larger update interval allows GAPI to have enough time to adjust its nodes dynamically, which can balance the effect of the larger moving out ratio and get a rather stable throughput. Figure 5 also shows that our throughput is better than PGrid in all cases.

Figure 6 shows the experimental results of the effect of the number of moving objects on throughput. As the number of moving objects increases, the throughput of both types of indexes decreases. This is because the greater the number of moving objects, the greater the probability of simultaneous updates of multiple threads in the same node and the more threads that need to wait. However, as the number of mobile objects increases, the throughput of our method decreases significantly slower than PGrid. This is because our method can dynamically split the nodes according to the actual running

```
Input: object movement information list update_info_list
Output: update each node counter value
(1)   for each (item in update_info_list)
(2)     node_in = get_location (item.new_pos)
(3)     if (node_in.x/m) == threadIdx.x and
(4)       node_out.y/n = = threadIdx.y
(5)       increase (count[node_in.x%threadIdx.x] [node_in.y%threadIdx.y]);
```

Algorithm 6: GPU-based counting algorithm GPU_count.

Table 1: Experiment parameters.

Parameter	Experimental value	Defaults
The total area (km^2)	200000×200000	—
Query area (km^2)	0.25, 1, 4, 16, 32	4
CPU threads	24	24
GPU stream processors	640	640
Update/query ratio ($\times 10^3$)	0.25, 0.5, 1, 2, 4, 8, 16	1
Number of spatial objects ($\times 10^6$)	5, 10, 20, 40	10
Update interval (second)	10, 20, 40, 80, 160	10

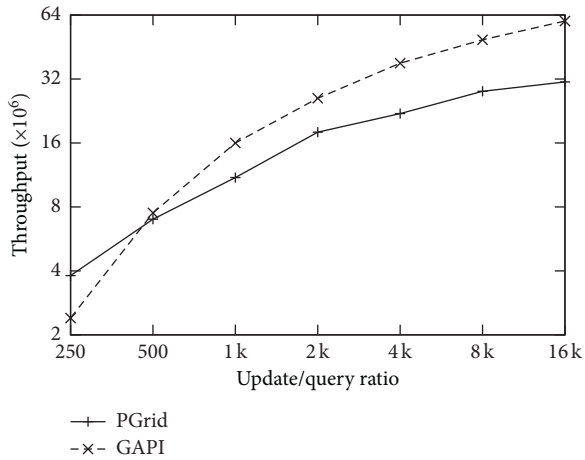

Figure 3: The effect of throughput on update/query ratio.

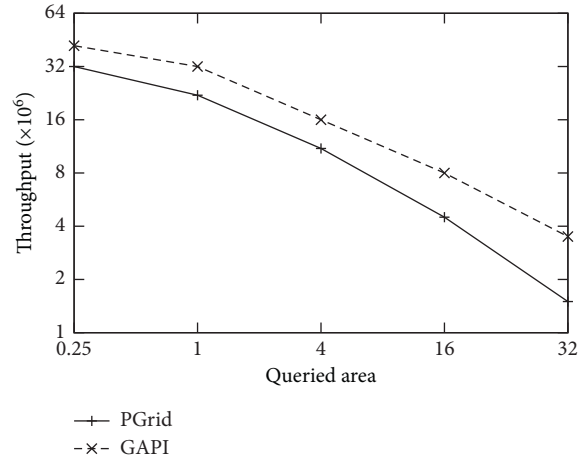

Figure 4: The effect of throughput on the size of the queried area.

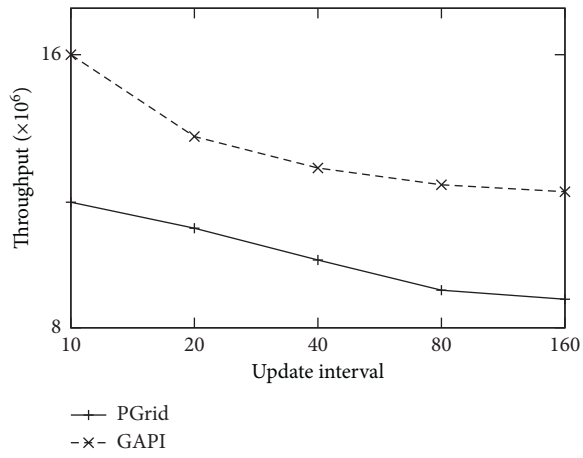

Figure 5: Effect of throughput on the interval between updating.

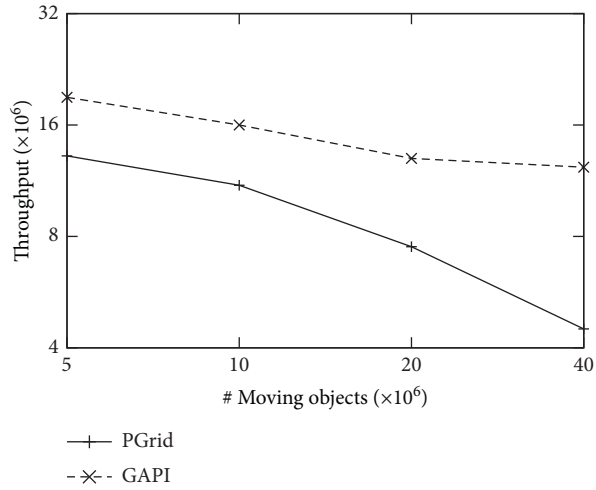

Figure 6: Effect of throughput on the number of moving objects.

conditions, which reduces the number of objects in the nodes, reduces the number of threads that need to wait, and improves the efficiency of parallel updating.

6. Conclusions

For satisfying the requirement of TCPS applications' spatiality and timeliness, based on the grid index, this paper proposes a hybrid indexing method that combines quad-tree and GPU acceleration to avoid the disadvantages of tree-based indexing and grid-based indexing. And, through experimental verification, the following conclusions have been drawn:

(1) Our index passes the calculation of the balanced index structure to the GPU for processing, making full use of the advantage that the GPU can quickly calculate a large amount of data, reducing the CPU's computational load as much as possible, thereby greatly improving the index optimization efficiency.

(2) The index structure divides the number of objects entering and leaving the leaf nodes per unit of time into criteria. Compared with the traditional dynamic index structure that uses the number of leaf nodes as the dividing criteria, the index structure has a better balance of hotspots to update the load. Capability performance in moving object updates is significantly better than existing methods.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they haveno conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Key R&D Program of China (2018YFB1003404) and the National Nature Science Foundation of China (61872071).

References

[1] D. P. F. Möller and H. Vakilzadian, "Cyber-physical systems in smart transportation," in *Proceedings of the 2016 IEEE International Conference on Electro Information Technology (EIT)*, Grand Forks, ND, USA, 2016.

[2] Y.-K. Huang, "Indexing and querying moving objects with uncertain speed and direction in spatiotemporal databases," *Journal of Geographical Systems*, vol. 16, no. 2, pp. 139–160, 2014.

[3] T. Nguyen, Z. He, R. Zhang, and P. Ward, "Boosting moving object indexing through velocity partitioning," in *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 860–871, 2012.

[4] G. Xiong, "Cyber-physical-social system in intelligent transportation," *IEEE/CAA Journal of Automatica Sinica.*vol. 2, no. 3, pp. 320–333, 2015.

[5] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," *SIGMOD Conference*, vol. 2, pp. 331–342, 2000.

[6] Y. N. Silva, X. Xiong, and W. G. Aref, "The rum-tree: supporting frequent updates in r-trees using memos," *The VLDB Journal*, vol. 18, no. 3, pp. 719–738, 2009.

[7] Y. Zhu, S. Wang, X. Zhou, and Y. Zhang, "Rum ± tree: a new multidimensional index supporting frequent updates," *Web-Age Information Management*, vol. 2, pp. 235–240, 2013.

[8] D. Idlauskas, S. Altenis, and C. S. Jensen, "Processing of extreme moving-object update and query workloads in main memory," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 23, no. 5, pp. 817–841, 2014.

[9] S. S. Sidlauskas, J. M. J. Christiansen, and D. Saulys, "Trees or grids?: indexing moving objects in main memory," *SIGSPATIAL*, vol. 2, pp. 236–245, 2009.

[10] X. Xu, L. Xiong, V. Sunderam, J. Liu, and J. Luo, "Speed partitioning for indexing moving objects," *Advances in Spatial and Temporal Databases*, vol. 2, pp. 216–234, 2015.

[11] R. B. Ray and A. K. Goel, "Supporting location-based services in a main-memory database," *MDM*, vol. 2, pp. 3–12, 2014.

[12] X. Xu, L. Xiong, and V. Sunderam, "D-grid: an in-memory dual space grid index for moving object databases," *17th IEEE International Conference on Mobile Data Management (MDM)*, vol. 1, pp. 252–261, 2016.

[13] D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," *SIGMOD*, vol. 2, pp. 37–48, 2012.

[14] Q. Che, C.-W. Li, Y. Zhang et al., "GAPI: GPU accelerated parallel method for indexing moving objects," *Journal of Frontiers of Computer Science and Technology*, vol. 11, no. 11, pp. 1713–1722, 2017.

[15] L.-V. Nguyen-Dinh, W. G. Aref, and M. Mokbel, "Spatio-temporal access methods: Part 2 (2003-2010)," *IEEE Data Engineering Bulletin*, vol. 33, no. 2, pp. 46–55, 2010.

[16] J. Tang, Z. Zhou, K. Ning et al., "A novel spatial indexing mechanism leveraging dynamic quad-tree regional division," 2013.

[17] S. Chen, B. C. Ooi, K.-L. Tan et al., "ST 2 B-tree: a self-tunable spatio-temporal b ± tree index for moving objects," 2008.

[18] A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, pp. 24–38, 1997.

[19] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, Newnes, London, UK, 2012.

[20] A. D. Sarma, S. Gollapudi, M. Najork et al., "A sketch-based distance oracle for web-scale graphs," 2010.