G-ML-Octree: An Update-Efficient Index Structure for Simulating 3D Moving Objects across GPUs

Ze Deng, Lizhe Wang†, *Senior Member, IEEE,* Wei Han, Rajiv Ranjan, and Albert Zomaya, *Fellow, IEEE,*

Abstract—In real simulation applications, simulations often involve large volumes of three-dimensinal (3D) moving objects. With the rapid growth of the scale of simulation-problem domains, it has become a key requirement to efficiently manage massive 3D moving objects. Conventional indexing approaches for managing 3D moving objects during simulations generally suffer from excessive update costs. Aiming to this problem, this paper first proposes an update-efficient indexing structure by fusing a loose Octree and one update-memo structure, namely ML-Octree. ML-Octree significantly reduces the update costs of one simulation involving massive 3D moving objects. Towards providing a more efficient indexing approach, this paper has explored the feasibility of paralleling ML-Octree by employing Graphic Processing Unit (GPU). A load-balancing scheme is used to further improve the update performance of the GPU-aided ML-Octree. Finally, a distributed GPU-aided ML-Octree is proposed for large-scale simulations. The experimental results indicate that (1) ML-Octree can acquire the update-performance gain of an order of magnitude similar to that of Octree, (2) the GPU-aided ML-Octree can accelerate 5.07× faster than a parallel ML-Octree with 8 CPU threads on average, (3) the load-balance scheme can improve GPU-aided ML-Octree by 2.3× on average, and (4) the distributed GPU-aided ML-Octree can efficiently support large-scale simulations.

Index Terms—Simulation, Moving objects, Indexing structure, Update-Efficient

1 Introduction

Simulation is an essential technique in many areas of science and engineering since real-world or physical experiments are extremely costly and pure mathematical approaches or analytic models do not adequately characterize the features of problems in areas of science and engineering [1], [2], [3]. In real simulation applications, simulations often involve large volumes of 3D moving objects. For example, in large cosmological simulations [4], [5], [6], the universe is represented by N moving particles, and the gravitational N-body simulation method is used to simulate the particles' interactions to deeply understand problems related to the universe, such as dark matter, the halo abundance, etc. The 3D moving objects represented by multi-agents [7] also appear in the simulations for studying the collective behavior of large aggregations of animals through the local rules of interaction among the 3D moving objects. The representative examples are bird flocks [8] and fish schools [9], [10]. With the rapid growth of both the complexity and the scale of problem domains [11], [12], [13], it has become a key requirement to efficiently manage massive moving objects for accurate and fast large-scale simulations [14].

Indexing approaches based on tree structures are dominant for efficiently managing 3D moving objects in such simulations. These indexing approaches make use of tree structures, e.g., Octree [4], [15], [16] and KD-tree [17], to recursively divide up the simulation domain to sub-volumes and to index all moving objects in a time-step simulation. Thus, a time-step simulation with the tree-indexing structure contains two phases: the query phase and update phase [18]. In the query phase, each moving object can employ the indexing structure to accurately and quickly obtain the information of nearby objects and to compute their new states (e.g., velocity, location, etc). After all moving objects finish this query phase, the simulation goes to the update phase, where the tree-indexing structure is updated or rebuilt to again index all moving objects for the next time-step simulation.

Previous works on the uses of tree-indexing structures in simulations mainly focus on optimizing their query performances. These approaches exploit the parallelism of tree-indexing structures for processing queries with the techniques of high-performance computing (HPC) such as supercomputers [4], [5], General Purpose GPUs (GPGPUs) [17], [19] and Field-Programmable Gate Array (FPGA) [6]. Nevertheless, a

- *Z. Deng, L. Wang (Corresponding author, lizhe.Wang@gmail.com), and W.Han are with the School of Computer Science, China University of Geosciences, Wuhan, 430074, P.R.China.*
- *Z. Deng, L. Wang (Corresponding author, lizhe.Wang@gmail.com), and W.Han are also with Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China.*
- *R. Ranjan is with School of Computing, Newcastle University, U.K.*
- *A. Zomaya is with the School of Information Technologies, The University of Sydney, Sydney, Australia.*

new challenge arises with the volume of 3D moving objects becoming very large in current simulations. For example, [4] presents a cosmological N-body simulation of more than 69 billion particles on 256K (2^{18}) processors. That means each processor is responsible evenly for indexing 2^{18} particles at least. Thus, the maintenance cost associated with updating or rebuilding indexing structure in the update stage currently has a major impact on the simulation performance. Meanwhile, traditional indexing structures for simulations such as Quadtree and Octree exhibit the inferior performance for updating large-scale moving objects with geometries [14]. This is because the sizes of these minimum Quadtree (or Octree) enclosing cells depend on the positions of the centroids of the objects and are independent of the size of the objects. There exists a pressing need for a new indexing approach for simulations that supports the efficient frequent updates of large-scale 3D moving objects with geometries.

To address the new research challenge, we first propose a new indexing structure that aims to support the frequent updates of 3D moving objects with geometries. The new structure has been constructed upon a loose Octree [20] and an update-memo structure [21]. Inspired by loose Quadtree for indexing 2D moving objects in [14], we employed a loose Octree (i.e., L-Octree) to index 3D moving objects to accommodate more frequent updates than that which occurs with Octree as the loose Octree has less possible cells associated with the 3D moving objects. Thus, the L-Octree can process updates much more quickly than the Octree can. Furthermore, we propose a L-Octree with an update-memo structure called ML-Octree, like RUM-tree (a R-tree with an update memo) [21], for efficiently handling frequent updates when the changes between consecutive updates are large. The update memo can help L-Octree to eliminate the need to delete the old data item during an update operation. Thus, the cost of an update operation for ML-Octree is approximately that of an insert operation.

Meanwhile, to gain more efficient update performances of ML-Octree for simulations, we map ML-Octree to the architecture of one Kepler GPU, namely G-ML-Octree. G-ML-Octree is organized by multiple 1D arrays and one hash table in GPU memory. Then, we parallelize the insertion operations of a large number of moving objects on G-ML-Octree using a dynamic load-balancing scheme on GPUs [22] and the *dynamic parallelism* feature of Kepler GPUs [23]. Finally, we employ a two-level indexing scheme (i.e., one global spatial index and multiple local indexes) to design a distributed G-ML-Octree on a GPU cluster for large-scale simulations.

A number of experiments have been carried out to evaluate the update performance of ML-Octree for 3D moving objects in the N-body simulation. The Octree-based indexing approach was referenced for the purpose of comparison. Furthermore, G-ML-Octree has been examined against the Central Processing Unit (CPU)-based alternative. Finally, the distributed G-ML-Octree has been evaluated on a GPU cluster with four nodes. The experimental results indicate that (1) the ML-Octree indexing structure can significantly reduce maintenance costs compared with the Octree indexing structure for the N-body simulation, (2) the GPU-aided ML-Octree on a Kepler GPU [23] outperforms the ML-Octree in terms of update costs, (3) the load-balancing scheme further improves the update performance of G-ML-Octree, and (4) the distributed G-ML-Octree is proved to be one way for large-scale simulations of massive 3D moving objects.

The principle contributions of the paper are summarized as follows:

1) We have proposed an update-efficient indexing structure named ML-Octree that supports the simulation of 3D moving objects with geometries. ML-Octree employs the loose feature of loose Octree cells and one update-memo structure to accommodate frequent indexing updates during the simulation.
2) We have designed a parallel indexing structure of ML-Octree on one GPU for fast index updating.
3) We have improved the update performance of G-ML-Octree by combining one GPU load-balancing scheme [22] and the Kepler GPU's *dynamic parallelism* feature.
4) We have presented a distributed G-ML-Octree on a GPU cluster for large-scale simulations.

The remainder of this paper is organized as follows: Section 2 presents the related work of indexing moving objects. Section 3 proposes the update-efficient indexing structure (ML-Octree). Section 4 introduces the design of ML-Octree on one GPU (i.e., G-ML-Octree) and the load-balancing scheme for G-ML-Octree. Section 5 introduces a distributed G-ML-Octree on a GPU cluster. Section 6 presents the experiments and results for the performance evaluation of the proposed approaches. Section 7 concludes this paper with a summary.

2 Related Work

This section describes the most salient works along indexing moving objects.

The indexing structures for moving objects can be roughly classified into two categories, i.e., the cell-based and the tree-based categories. The cell-based structures employ a grid structure [24], [25], [26] to partition the indexing space into equal-sized cells for indexing moving objects. In [27], authors introduce a novel grid index called DGI (Distributed Grid Index). The server transmits DGI and the client examines the received index to process spatial queries. The proposed index structure and search algorithm support

efficient spatial queries in a wireless broadcast environment, shortening query search times. However, the selection of cell size is non-trivial. Too large of a size may cause too many moving objects residing in one cell to decrease the query performance of the indexing structure. On the contrary, too small of one can lead to the high costs of space and time for building and rebuilding their indexing structures [28].

The tree-based approaches index moving objects to retrieve predicative answers or to approximate query answers using tree-based indexing structures. Traditionally, tree-based approaches exploited the indexing structures of R-tree and its variants. For instance, R*-tree in [29] and TPR-tree in [30] are used to index positions of moving objects to support queries. However, the R-tree and its variants were designed mainly for static data. Thus, these indexing structures may suffer from the high maintenance costs associated with indexing moving objects since their indexing structures are updated frequently with the continuous changes of moving objects. Bottom-up update approaches have been made to alleviate the update-performance issue of the R-tree family. For example, Lazy Update R-tree (LUR-tree) [31] updates the structure of the index only when an object moves out of the corresponding minimum bounding rectangle (MBR). If the new position of an object is in the MBR, this changes only the position of the object in the leaf node. The Frequently Updated R-tree (FUR-tree) [32] extended the LUR-tree through a scheme in which a certain object can move to one of its siblings. However, the update performances of these bottom-up methods degrade quickly when consecutive changes are large. To deal with the drawback of bottom-up methods, Y. N. Silva et al. proposed a R-tree with an update memo (RUM-tree) to handle moving-object updates more efficiently. The update memo eliminates the need to delete the old data item from the index during an update operation. In [33], authors proposed a novel query indexing structure, referred to as the Query Region tree (QR-tree), which allows the server to cooperate with moving objects efficiently by leveraging the available computational resources of the moving objects to improve the overall system performance.

For the large-scale simulations of moving objects, R-tree indexing suffers from the heavy cost of rebalancing the R-tree structure [18]. Therefore, non-balanced tree-based structures such as Quadtree, Octree and KD-tree have been widely used in indexing moving objects in the large-scale simulations of moving objects. For example, for 2D simulations, the loose Quadtree structure is applied for indexing moving objects with extents in games [14], and Sim-tree is proposed for indexing vehicles for large-scale microscopic traffic simulations [18]. In the case of the simulations of 3D moving objects, the Octree structure [4], [15], [16] and the KD-tree [17] structure are used for indexing moving objects (moving particles) for the N-body simulation. In [34], a hybrid indexing structure of grid and loose Octree called as GLOctree is proposed to provide a general purpose spatial partitioning method for dynamic scenes. GLOctree holds a great query performance with the grid structure and an excellent update performance by loose Octree. Similar to GLOctree, we utilize loose Octree for efficient updates as well. Unlike GLOctree, we employ GPU to gain a good update performance. As we can see, the current most indexing structures for 3D moving-object simulations still employ the traditional Octree and KD-tree indexing structures. These indexing structures seldom consider the geometric feature of moving objects and exhibit inferior performance in updating large-scale moving objects with geometries [14].

Different from existing indexing methods for moving objects, this paper targets the emerging challenge of supporting efficient frequent updates of large-scale 3D moving objects with geometries during simulations. Our indexing method gains a great update performance through inheriting the merits of the loose Octree and the update-memo structure, and through exploiting the parallelism of GPU device and GPU cluster.

3 The Update-Efficient Indexing Approach

This section first formulates this problem. Then, the details of the proposed indexing structure ML-Octree are described. This section ends with a series of algorithms and one deletion scheme upon ML-Octree.

3.1 Problem Formulation

We assume that the simulation space is a 3D Euclidean space, and the geometric feature of the moving object is considered. Then, we formulate the problem.

Definition 1 (**A 3D moving object**) : A 3D moving object with geometries in a 3D Euclidean space is described as: $o = (id, MBB, r, S)$. In Definition 1, id is the identifier of the moving object. MBB = (p_{l+b}, p_{r+t}) is a minimum bounding box that represents the 3D moving object, where $p_{l+b} = (x_1, y_1, z_1)$ and $p_{r+t} = (x_2, y_2, z_2)$ represent the position coordinates of the bottom-left corner and the top-right corner of the MBB respectively, in a 3D Euclidean space. r is half the length of a side of the minimum bounding hypercube of MBB. S is a set of states that describes the moving object. The content of S varies with different kinds of moving objects.

Definition 2 (**Simulation moving objects**) : A set of N moving objects in one simulation with S timesteps. O is defined as a dynamic set over the simulation timesteps: $O = \{o_i^j \mid 1 \leq i \leq N \wedge 1 \leq j \leq S\}$. o_i^j represents the i-th moving object in the j-th timestep of the simulation.

Definition 3 (**The indexing structure for O**) : The indexing structure for indexing the dynamic set of moving objects O is defined as imbalanced tree-based indexing structure $T(O)$.

Here, the objective is to make the indexing structure $T(O)$ update efficient for large-scale simulations.

3.2 The Indexing Structure: ML-Octree

An update-efficient indexing structure based on a loose Octree [20] and an update-memo structure [21] is proposed for the simulation of 3D moving objects, namely ML-Octree. ML-Octree is constructed by the two phases of (1) loose Octree construction and (2) update-memo incorporation.

3.2.1 Loose Octree construction

In our highly dynamical simulation environment, the Octree structure suffers from the drawback that the structure is frequently updated, as the size of the its minimum-enclosing cell depends on the positions of the moving objects. Thus, these sizes of Octree cells require frequent change with the moving objects. The loose Octree [20] can overcome this issue by expanding the size of the space. It is spanned by each Octree cell c of width w by a cell expansion factor p ($p > 0$). Thus, the expanded cell is of width $(1 + p) \times w$, and a moving object $o = (id, MBB, r, S)$ is associated with its minimum-enclosing expanded Octree cell. Therefore, the procedure of inserting o into the loose Octree is one of finding the smallest Octree cell c that contains the centroid of $o.MBB$, and whose expanded cell also contains o. To determine the appropriate cell of width w for o, we use the following formula described in [14]:

$$\frac{1}{1+p} \leq \frac{w}{2 \times o \cdot r} < \frac{2}{p} \tag{1}$$

Let the root cell of the loose Octree have width 2^g so that all other cells have $w = 2^k$ ($k \leq g$). One function is used in [14]:

$$M(x) = 2^k, (2^{k-1} < x \leq 2^k) \tag{2}$$

The formula (2) is used to compute the number of levels in the loose Octree at which the smallest Octree cell c of the moving object o could possibly lie. It is upper bounded by UB. The value of UB is:

$$UB = \log_2^{M(2/p)} - \log_2^{M(1/(1+p))} \tag{3}$$

Thus, given p, moving object o can find the corresponding cell with formula (1) for the cost of traveling at most UB levels in the loose Octree. Fig. 1 provides an example for constructing a loose Octree to index a set of moving objects O (Ref O to Definition 1 in Section 3.1) including three moving objects o_1, o_2, and o_3 given a specific simulation timestep. As we can see in Fig. 1(a), the insertion of these three objects with the above-mentioned insertion procedure leads to the cell

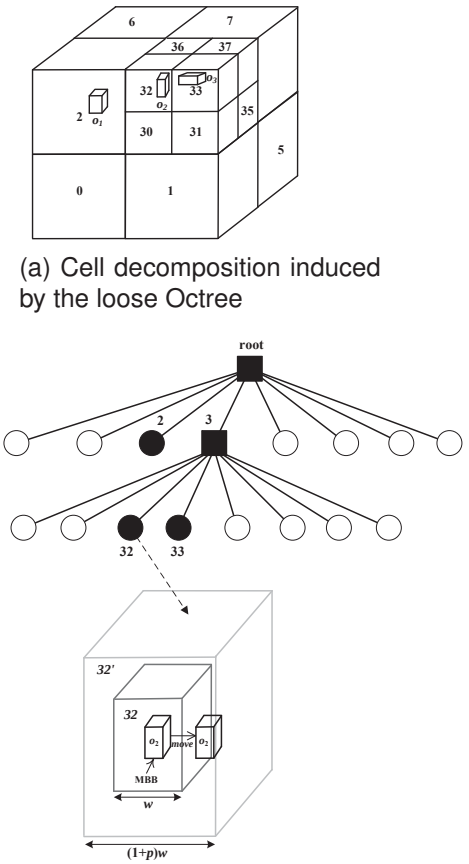


Fig. 1. An example for constructing loose Octree

decomposition of the loose Octree. Thus, the indexing space is first partitioned into eight octants from cell 0 to cell 7. Then, cell 3 further is decomposed into eight octants including the cells from cell 30 to cell 37. As a result, each moving object is assigned to one Octree cell. That means o_1 corresponds to cell 2, o_2 is in cell 32, and o_3 falls under cell 33. The corresponding tree representation is shown in Fig. 1(b). In Fig. 1(b), we assume o_2 moves away from cell 32 of width w. Unlike the Octree, o_2 should have been deleted and reinserted. The loose Octree can avoid this reinsertion since the centroid of MBB of o_2 is still enclosed by the expanded Octree cell 32' of width $(1 + p)w$.

3.2.2 Update-memo incorporation

To further improve the update performance of the loose Octree, we incorporate an update memo that has been used in R-tree for frequent updates [21] into the loose Octree called ML-Octree. The primary feature behind ML-Octree is that the old entry in the tree is allowed to co-exist with newer entries before it is removed later, rather than deleting it when updating moving objects. Like [21], garbage cleaners are also used to remove old data entries in ML-Octree lazily. Thus, the cost of an update operation is about the cost of an insert operation.

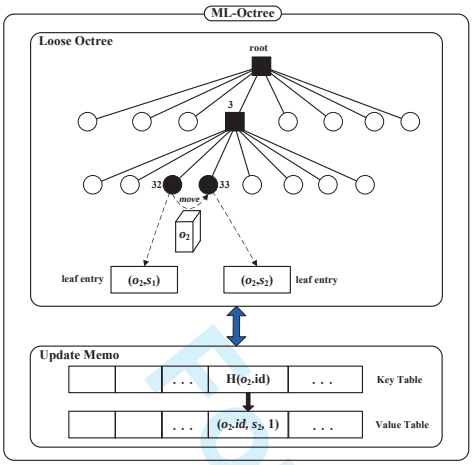


Fig. 2. Illustrating the structure of ML-Octree

In ML-Octree, each index entry in one leaf cell is assigned a stamp by a global stamp counter when the entry is inserted into the tree. The stamp places a temporal relationship among leaf entries, i.e., an entry with a smaller stamp is inserted before an entry with a larger stamp. Normally, index entry e in the loose Octree is represented with $(o, stamp)$, where o is the same as Definition 1 and $stamp$ is the assigned stamp number. To distinguish the latest entries from the obsolete entries, the update-memo structure is used as in [21]. Specially, the update-memo structure is a hash table. It contains multiple $< key, value >$ pairs. Each entry is formed with $< H(o.id), V >$, where $H(o.id)$ is the hash value of $o.id$ and where $V = (o.id, Slatest, Nold)$ represents one entry in the update memo, in which $o.id$ is the object identifier, $Slatest$ is the stamp of the latest entry of the object, and $Nold$ stands for the maximum number of obsolete entries.

Fig. 2 shows an example of a hybrid structure with the loose Octree and the update memo for indexing moving objects. We assume that moving object o_2 initially falls under cell 32 and then moves to cell 33. This motion causes two changes in ML-Octree. First, one new leaf entry (o_2, s_2) is inserted into cell 33, while the old leaf entry (o_2, s_1) remains in cell 32, holding the condition $s_2 > s_1$. Then, one key-value pair $< H(o_2.id), V = (o_2.id, s_2, 1) >$ is inserted into the update memo. Value $V = (o_2.id, s_2, 1)$ means that for moving object o_2, the stamp of the latest entry is s_2, and there exists one obsolete entry, i.e., $(o_2.id, s_1)$, in the leaf nodes of ML-Octree. This processing avoids the deletion of an old leaf entry $(o_2.id, s_1)$, and the latest entry of o_2 is recorded in the update memo. The deletion of $(o_2.id, s_1)$ is done later by the clean-on-touch way or garbage cleaners in the batch manner through the update memo.

3.3 The Algorithms upon ML-Octree

This section describes insert/update and search algorithms on ML-Octree. We present the deletion scheme on ML-Octree as well.

3.3.1 Insert and update

The update procedure is similar to the insert procedure except for the operations on the update memo. Thus, given a 3D moving object o (see Definition 1), and ML-Octree T with cell expansion factor p, we present both insert and update procedures using the pseudocode in algorithm 1. As we can see, initially, we query the update memo with $o.id$ (line 2) and record the current timestamp (line 3). If there is no record in memo, there are two cases. The first case is that the moving object has been an existing object in ML-Octree, so we first update the update memo (lines 4-8) and then update the index entry of o by an insert operation (lines 10-20). Otherwise, o is a new moving object. For this case, we generate a new index entry and then execute an insert operation (lines 10-20) as well. During the insert procedure, we use the search algorithm in [14] to locate the tree cell holding the index entry (lines 21-27).

3.3.2 Search

In our setting, the basic query type is the range query used to find out index entries within a given range. Let q be one range query and RS be one result set. The search algorithm with the range query is presented in algorithm 2. First, we search matched entries from the root node (line 2) and then search its children (lines 3-15). For each child, we match its expanded cell with the range query q. If there is overlap, the procedure is recursively repeated (lines 9-11).

3.3.3 Delete

Since no moving objects are deleted during the simulation procedure, no delete operations happen on the update memo. Thus, we focus only on how to delete obsoleted index entries in L-Octree and we employ the garbage-cleaning scheme in RUM-tree [21]. Concretely, multiple cleaning tokens that are logical objects for traveling tree nodes are used to clean the old index entries in the traveled nodes. Note that, unlike RUM-tree, where data are stored only in leaf nodes, data are distributed in all nodes in ML-Octree. Therefore, cleaning tokens are responsible for all tree nodes rather than leaf nodes. Fig. 3 provides an example of the deletion scheme, where Token 1 and Token 2 clean nine nodes and eight nodes, respectively.

4 The GPGPU-Aided ML-Octree

We first map ML-Octree to GPU's memory to form G-ML-Octree, and then, we present a fast update scheme for G-ML-Octree.

Algorithm 1: Insert and Update on ML-Octree

```
1  InsertAndUpdate_Procedure(T, o, p)
2  memo_e ← retrieve index entry with the hash value of o.id from the update memo
3  s = CurStampCount+1 /* the current stampcount increases by one */
4  if memo_e == NULL then
5  |  if o is an existing object then
6  |  |  insert an index entry (o.id, s, 1) into the memo
7  |  end
8  end
9  else
10 |  memo_e.Slatest = s, memo_e.Nold++
11 end
12 set one new index entry e = (o, s)
13 c = findMINCell(T, o, p) /* find the minimum enclosing expanded Octree cell for o */
14 if c == null then
15 |  create one cell satisfied with formula (1) through recursively partitioning the indexing space of T
16 end
17 e.s ← (stampcount = stampcount + 1) /* set the e's stamp through the global stamp counter */
18 if c is not full then insert e into the cell c
19 ;
20 else split the cell into eight subcells and insert e into one of them
21 ;
22 return
   /* find the minimum enclosing expanded Octree cell */
23  findMINCell(T, o, p)
24 rc = NULL;
25 for i = log_2^{M(1/(p+1))}; i < log_2^{M(2/p)} ; i++ do
26 |  w ← 2^{(i+1)} × M(o.r) /* computing the width of the smallest possible cell c containing o, M(.) see formula(2) */
27 |  if o falls in the c with w in T then rc ← c;
   |  break; ;
28 end
29 return rc
```

Algorithm 2: Search on ML-Octree

```
1  Search_Procedure(T.cell, q, RS) /* Input: T.cell is the cell of a ML-Octree, q is a range query; Output: RS is the query results */
2  GetResults(T.cell, q, RS) /* return all matched index entries in the T.root node */
3  CellSet = findCells(T.cell, q) /* find all subcells of T's cell overlaps q */
4  for each cell c ∈ CellSet do
5  |  if c's expended cell (i.e (1+p)w ) overlaps q then
6  |  |  GetResults(c, q, RS)
7  |  |  CellSet = findCells(c.cell, q)
8  |  |  for each subcell subC ∈ CellSet do
9  |  |  |  if subC.(1+p)w overlaps q then
   |  |  |  |  /* recursive search */
10 |  |  |  |  Search_Procedure(subC, q, RS)
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return RS
   /* retrieve all matched query results in one given cell */
16  GetResults(Cell, q, RS)
17 EntrySet ← get all entries in the Cell
18 for each index entry e ∈ EntrySet do
19 |  gain index entry memo_e with the hash value of e.o.id from the update memo
20 |  isLatest = (e.stamp == memo_e.Slatest)? TRUE:FALSE
21 |  if isLatest ∧ e.o.MBB overlaps q then
22 |  |  RS = RS ∪ {e}
23 |  end
24 end
```

4.1 Data Structures for G-ML-Octree

ML-Octree has two components: one loose Octree and one update memo. Thus, we achieve the structure map through constructing the loose Octree and the update memo on the GPU, respectively.

In [19], the Octree structure on one GPU has been represented by multiple arrays. Different from the structure in [19], in our setting, the indexed data are 3D geometries rather than point data, and the expanded cell factor needs to be stored. Therefore, we extend the data structure of Octree in [19] for loose Octree. Fig. 4(a) shows that L-Octree in the GPU memory consists of a cell array and a data array. The cell array holds all Octree cells. Each element in the cell array has eight sub-elements that represent eight children, and the grey-filled sub-elements mean they are leaf nodes. Meanwhile, each element in the data array is used to store 3D geometry data and the properties of each cell especially including the expanded cell factor.

The update-memo structure in ML-Octree is represented by one hash table in our design. Thus, we employ one parallel hash table structure on the GPU [35] for ML-Octree. Fig. 4(b) illustrates the hash table. As we can see, this hash table is one two-level structure. The index entries first distributed smaller buckets by a first-level hash function. Then, the index items in each bucket are stored in three sub-tables by

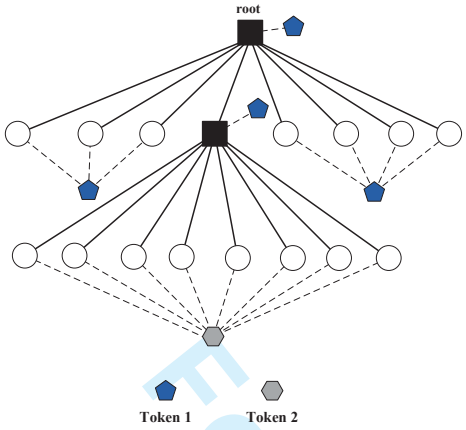
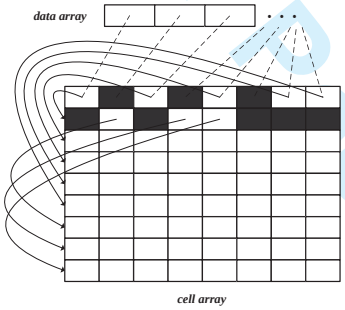
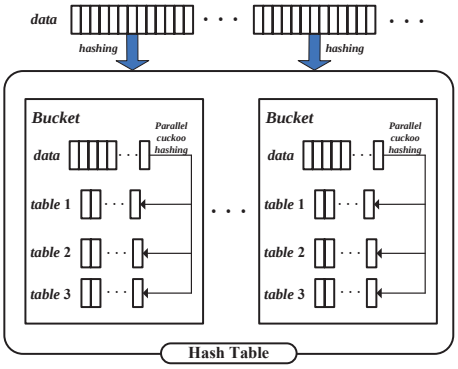


Fig. 3. Illustrating the deletion scheme of ML-Octree



(a) Data structure of L-Octree in GPU memory



(b) Data structure of update memo in GPU memory

Fig. 4. Illustration of the structure of ML-Octree on GPUs

a parallel cuckoo hashing algorithm [35].

4.2 Updating G-ML-Octree

After constructing G-ML-Octree on the GPU, we need to update the indexing structure efficiently during the update stage of the simulation procedure. However, the construction procedure of Octree is a dynamic one, since there is no information beforehand on how deep each branch will be, based on the description in [22].

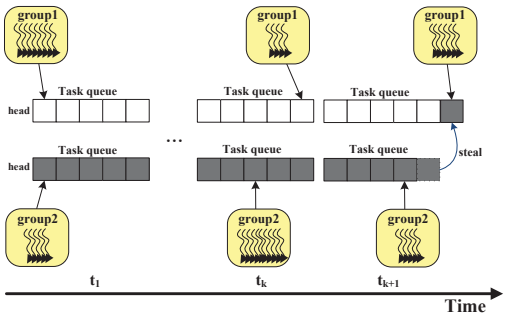


Fig. 5. Illustrating the load-balancing scheme of G-ML-Octree

This results in the load-unbalance problem of the GPU threads for constructing our G-ML-Octree.

To address this issue, we propose one two-level balance scheme. First, we use a task stealing scheme in [22]. Each GPU thread group is assigned a set of update tasks by one task queue and attempts to steal tasks from another GPU thread group once it has completed assigned tasks. The task assignment and stealing are executed by CPU.

However, the issue of load unbalance still exists when a GPU thread group executes sequentially the tasks from its task queue since different tasks have various workloads. One solution is to resize the GPU thread group for different tasks by host-calling kernels. Nevertheless, the Peripheral Component Interconnect (PCI) traffics generated by multiple kernel launches between the host and device reduce the update performance. We further employ the *dynamic parallelism* feature provided by the NVIDIA's GPU based on the Kepler architecture to solve this problem. *Dynamic parallelism* enables a CUDA kernel to create and synchronize new nested work, using the CUDA runtime API to launch other kernels and optionally synchronize on kernel completion, without CPU involvement. Thus, each thread group can adaptively resize itself for incoming tasks with various workloads by using *dynamic parallelism*. Fig. 5 provides an example of our scheme of load balance. First, two thread groups, i.e., group 1 and group 2, are assigned two task queues and begin to execute individual tasks at time point t_1. Then, group 1 and group 2 change their sizes by using dynamic parallelism to efficiently process different tasks as time escapes (e.g., at time point t_k). At time point t_{k+1}, group 1 steals one task through one CPU thread from the tail of the queue for group 2 since it has no tasks to execute.

5 A Distributed G-ML-Octree on a GPU Cluster

Since the number of indexed moving objects is limited by the size of GPU memory on one GPU device, one GPU device may not index all 3D-moving objects in large-scale simulations. Therefore, in this section, we

propose a distributed G-ML-Octree on a GPU cluster. We assume a GPU cluster contains n nodes and each node is equipped with one GPU device. We index N 3D-moving objects across n nodes. Concretely, the indexing is composed of (1) a global spatial index, and (2) n local G-ML-Octree indexes.

We employ one domain decomposition method for parallel reservoir simulation [36] to design the global spatial index. Concretely, the minimum global space containing all 3D moving objects first is partitioned into n disjoint domains D_1, D_2, ... , D_n (one for each node) and each node t owns D_t only. The partition scheme depends on the GPU memory size of each node. That means $V_{D_1} : V_{D_2} :, ..., : V_{D_n} = S_{node_1} : S_{node_2} :, ..., : S_{node_n}$. V_{D_i} is the volume of D_i and S_{node_i} represents the size of GPU memory of $node_i$. Meanwhile, each node $node_i$ holds a maximum capacity of holding moving objects MAX_{node_i}. In the procedure of simulation, one node may receive some incoming moving objects from other nodes. Thus, we assume $\sum_{i=1}^{n} MAX_{node_i} = \alpha \times N$, where α is a capacity expansion factor (≥ 1) to guarantee that each node has enough GPU storage space to store incoming moving objects and intermediate results. That means we need deploy enough nodes in a cluster for large-scale simulations. After then, each moving object o is distributed into one domain D' using the following formula:

$$D' = \arg\max_{D_i}[overlap(D_i,\ o)] \wedge (C_{D_i} \leq MAX_{node_i}) \tag{4}$$

Where, $overlap(D_i, o)$ is the overlap of spatial region between D_i and o, and C_{D_i} is the current number of moving objects in D_i. To avoid network transfer overhead, the global spatial index is replicated across nodes in the cluster. Noted that, the global spatial index only records the mapping relation between each cluster node and its corresponding space domain. Therefore, it only consumes a very small space overhead. Thus, we simply store the global spatial index in the host memory of each node. After distributing all moving objects, we generate n local indexes on n nodes by constructing n G-ML-Octrees. In the distributed case, the update procedure is still similar to the insert procedure, we also present both insert and update procedures using the pseudocode in algorithm 3 like algorithm 1. As we can see, we first need to mark the incoming moving object o as obsolete one on n_s if o has not belonged to n_s by global index determination (lines 3-5), and then insert o into the local index of the remote node n' (line 6). Otherwise, the moving object o still resides on n_s. Thus, we only insert o into the local index of n_s again. A search algorithm with the range query over a distributed G-ML-Octree is presented in algorithm 4, for a range query q on n_s, we first find out all nodes overlap q by using global index on n_s, then forward q to these nodes to execute search procedure in parallel, finally all results are returned to n_s and merged into a final result.

Algorithm 3: Insert and Update on distributed G-ML-Octree

```
1 InsertAndUpdate_Procedure(o, n_s)/* o is a moving object, n_s is one local node who initializes the operation in the cluster */
2 n' ← find the node holds o through n_s's global spatial index
3 if n' is not n_s then
4 |   Mark o as an invalid object in the local index of n_s
5 end
6 Insert o into the local index of n'
```

Algorithm 4: Search on distributed G-ML-Octree

```
1 Search_Global_Procedure(q, n_s, RS) /* Input: q is a range query, n_s is one node who initializes the query q in the cluster; Output: RS is the query results */
2 NodeSet = findNodes(n_s's global index, q)
    /* return all nodes whose region overlap q */
3 for each node n ∈ NodeSet in parallel do
  |   /* search local indexes */
4 |   Search_Local_Procedure(q, n's local index, RS') Return RS' to n_s
5 end
6 n_s merges all RS' to from RS
7 return RS
```

6 Performance Evaluation

We have evaluated the performances of the proposed indexing methods of ML-Octree and G-ML-Octree for one 3D N-body simulation on one computer equipped with a Kepler GPU (Titan Black), and the configurations are presented in Table 1. Meanwhile, we also evaluated the update performance of ML-Octree on one high-performance rack server (see in Table 2). Furthermore, the update performance of distributed G-ML-Octree has been measured on a GPU cluster with four nodes. These four nodes are connected via one 100Mbps's Ethernet. Table 3 gives major configurations of the four nodes.

6.1 Experimental Setup

To evaluate the proposed indexing methods' potential in serving large-scale simulations for 3D moving objects with extents, we take the N-body simulation as a

TABLE 1
Configurations of the Computer

Specifications of CPU platforms	**Computer**
OS	Linux Ubuntu 14.04
CPU	i7-4790 (3.6GHz, 4 cores)
Memory	32GB DDR3
Specifications of GPU platforms	**GTX Titan Black**
Architecture	Kepler GK110
Memory	6GB DDR5
Bandwidth	Bi-directional bandwidth of 16GB/s
CUDA	SDK 6.0

TABLE 2
Configurations of the Rack Server

Specifications of CPU platforms	**Server**
OS	Linux Ubuntu 14.04
CPU	Intel Xeon E5-2609 v4 (1.7GHz, 8 cores)
Memory	32GB DDR4

case study. Concretely, we modify an open-source N-body simulator REBOUND [37] to support 3D moving objects and our indexing structures. In our following experiments, one 3D moving object is a spheroid. The modified simulator can generate numerous sphere objects in the unit-volume space [0,1] $\times$ [0,1] $\times$ [0,1]. Each 3D moving object is represented by a set of properties, including position, velocity, acceleration, mass, and radius.

One time of simulation consists of multiple timesteps. Each timestep has two stages by using indexing structures: 1) the query stage, where each object employs our indexing structure to search neighboring objects to calculate new values for interaction force and acceleration, and 2) the update stage, where all object move to new positions and update indexing structures according to individual new states and collision rules in [37].

6.2 ML-Octree Evaluation

Here, we evaluate the update performance and space cost of ML-Octree for the N-body simulation. For comparison, we observe the update performance and space cost of Octree, L-Octree, and ML-Octree. Since REBOUND has employed an Octree to index moving objects, we implemented a L-Octree in [38] and replace the Octree in REBOUND. Furthermore, to realize ML-Octree, we obtained the source code about RUM-tree [21] from authors, then incorporate the update-memo structure of RUM-tree into L-Octree to form ML-Octree. All these three indexing structures have been constructed by using two double-precision data structures (i.e., tree and hashing table) on one computer equipped with CPU i7-4790 (3.6GHz) and 32GB memory (see in Table 1). For L-Octree and ML-Octree, the cell expansion factor p is set to 0.999, which is an optimal value according to the experimental results in [14]. The update performance is measured by the operation number incurred by indexing updates and the runtime speedup.

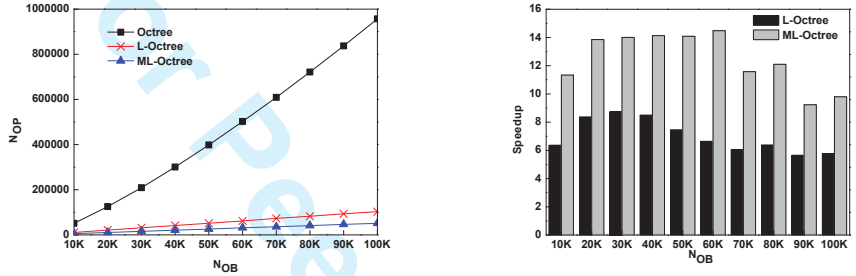
First, we fix the number of simulation timesteps as 20 and observe the the operation number incurred by indexing updates and the runtime speedup for various numbers of moving objects ranging from 10K to 100K. These operations include query, delete, and insert operations. The experimental results are presented in Fig. 6(a) show that L-Octree outperforms Octree about an order of magnitude in terms of operation number N_{OP}. Moreover, ML-Octree only uses half of the operations of L-Octree. Meanwhile, the results of runtime speedup in Fig. 6(b) show that L-Octree can gain a speedup of 6.98$\times$ compared with Octree, while ML-Octree obtains a speedup of 12.4$\times$ faster than Octree on average.

The reason for the great performance gains of L-Octree and ML-Octree is that L-Octree and ML-Octree can avoid numerous unnecessary updates through their expended cells. On the other hand, ML-Octree can further enhance the update performance using the update-memo structure to delete old index data lazily compared with L-Octree. In addition, we also perform another experiment where the number of moving objects is fixed to 100K and the number of timesteps ranges from 5 to 30. The experimental results shown in Fig. 7 are similar to those of the first one.

Moreover, we observe the space costs of ML-Octree, Octree, and L-Octree for various numbers of moving objects for one N-body simulation with 20 timesteps. The space costs of the three indexing structures have been measured in both single-precision and double-precision cases. Fig. 8(a) shows that ML-Octree and Octree have maximum and minimum space costs, respectively, among the three indexing structure. This reason is that L-Octree needs to maintain an additional loose factor than Octree, and ML-Octree needs to incorporate an update memo into L-Octree. Moreover, we can see that the size of ML-Octree increases with the increase of the timestep, while the sizes of Octree and L-Octree are unchanged. That is because that the update memo of ML-Octree is empty when the simulation starts and then its size gradually expands as the simulation timestep increases (see in 3.2.2 Update-memo incorporation). Meanwhile, ML-Octree adapts the lazy update scheme so that the number of obsolete index entries increases as the simulation timestep increases as well. In contrast to ML-Octree, Octree and L-Octree have no such an

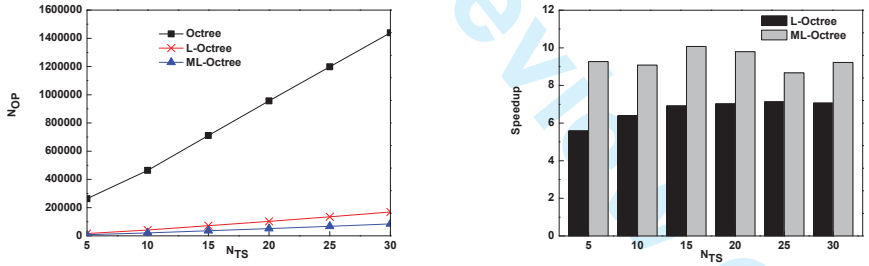
TABLE 3
Major hardware and software features of four nodes in the cluster

Features	node #1	node #2	node #3	node #4
		Hardware		
CPU	i7-4790@3.6GHz	i7-5820K@3.3GHz	i7-5820K@3.3GHz	i7-4790K@4.0GHz
Host memory	32GB	32GB	32GB	16GB
GPU	GTX Titan Black	GTX Titan X	GTX Titan Black	GTX 960
GPU memory	6GB	12GB	6GB	4GB
		Software		
OS	Linux Ubuntu 14.04	Linux Ubuntu 15.04	Linux Ubuntu 16.10	Linux Ubuntu 15.04
CUDA	6.0	8.0	8.0	8.0



(a) The number of operations (b) The runtime speedup

Fig. 6. Evaluating the update performance of ML-Octree with 20 timesteps



(a) The number of operations (b) The runtime speedup

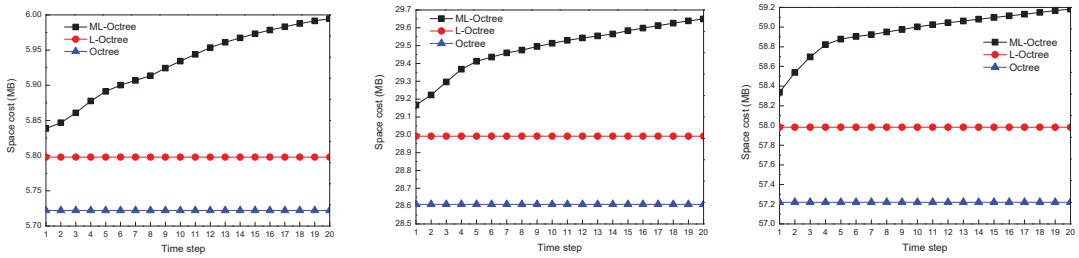
Fig. 7. Evaluating the update performance of ML-Octree with 100K moving objects

update scheme so that their sizes can keep unchange in the simulation. Fig. 8(b) and (c) indicate the similar results to the one in Fig. 8(a). Noted that, when the size of ML-Octree becomes very large, the garbage cleaning scheme (see in 3.3.3 Delete) can be triggered to alleviate the issue of space cost.

Furthermore, from Fig. 9 we also observe that the space cost of ML-Octree in the single-precision case can on average decrease by 46% compared with the space cost in the double-precision case. Meanwhile, we compare the speedup of ML-Octree relative to Octree for update time, in both single-precision and double-precision cases. According to the experimental results in Fig. 10, ML-Octree has almost the same update performance in the single-precision and double-precision cases.

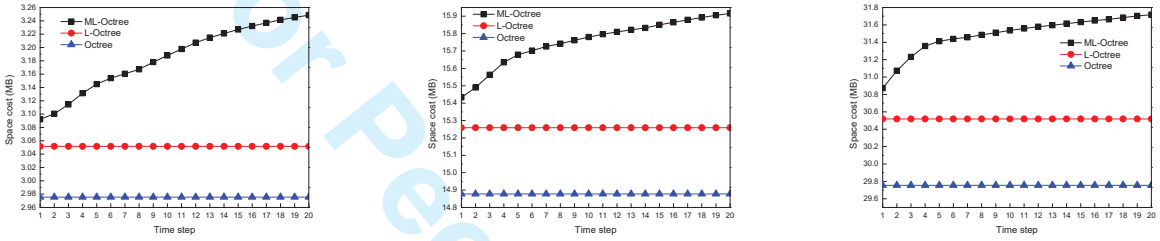
6.3 G-ML-Octree Evaluation

The goal of this experiment is to investigate the update performance of G-ML-Octree for N-body simulation and the effectiveness of our load-balancing scheme. Thus, we first measure the update performance of G-ML-Octree without load-balance scheme with 10 timesteps. For comparison, we used one GPU-aided KD-tree [17] structure for the N-body simulation. For convenience, we denoted the GPU-aided KD-tree as G-KDtree. Meanwhile, we also compared the update performances of ML-Octree on two different machines denoted as M1 and M2. M1 (see in Table 1) is one computer equipped with one CPU i7-4790 (3.6GHz, 4 cores) and 32GB memory. M2 (see in Table 2) is a rack server with one CPU Intel Xeon E5-2609



(a) N_{OB}=10K (b) N_{OB}=50K (c) N_{OB}=100K

Fig. 8. The comparison of space costs in the double-precision case



(a) N_{OB}=10K (b) N_{OB}=50K (c) N_{OB}=100K

Fig. 9. The comparison of space costs in the single-precision case

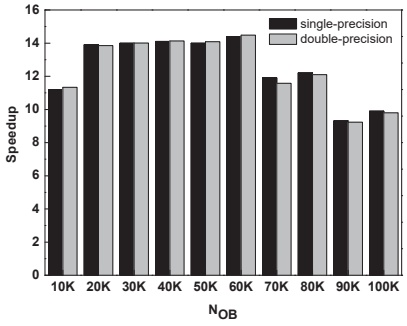


Fig. 10. The comparison of update performance of ML-Octree in the single-precision and double-precision cases

v4 (1.7GHz, 8 cores) and 32GB memory. For the sake of fairness, we implemented a parallel version of ML-Octree on both M1 and M2 by using the similar parallel method as distributed G-ML-Octree and OpenMP. Since M1 has a 4-core CPU and M2 owns a 8-core CPU, we only measured the update performance of parallel ML-Octree with 8 CPU threads. For convenience, we denoted the ML-Octree with 1 thread and 8 threads on M1 as ML-Octree-1 and ML-Octree-1^+, respectively. Similarly, the ML-Octrees with 1 thread and 8 threads on M2 are represented as ML-Octree-2 and ML-Octree-2^+, individually. In addition, we considered the data transfer time between CPU and GPU when evaluating G-ML-Octree and G-KDtree.

Fig. 11 shows that the update time costs of ML-Octree-1 and ML-Octree-2 range from 39.8s to 233s and 59s to 335s, respectively, as the number of moving objects changes from 100K to 500K. Comparing to ML-Octree-1 and ML-Octree-2, ML-Octree-1^+ and ML-Octree-2^+ can on average gain the speedups of 3.24× and 6.78× due to the parallel acceleration of 4 CPU cores on M1 and 8 CPU cores on M2. Furthermore, we can see that G-KDtree can averagely accelerate 2.17× and 1.53× , respectively, relative to ML-Octree-1^+ and ML-Octree-2^+. The performance gain of G-KDtree is attribute to its parallel construction algorithm in [17]. Finally, Fig. 11 reflects that G-ML-Octree averagely has a speedup of 3.24× compared with G-KDtree. The reason for the great performance of G-ML-Octree is that the combination of the update scheme of ML-Octree, multiple arrays and the parallel hash table can cater to the feature of the parallel access of massive GPU threads to progressively update its indexing structure as the simulation timestep increases. Comparing to G-ML-Octree, G-KDtree has to continuously reconstruct its structure since the space partition may not be effective once one simulation timestep ends.

Meanwhile, we also observed the transfer time of G-ML-Octree shown in Fig. 12 only takes 0.53% on average, when the number of moving objects changes from 100K to 500k. The reason for the results lies in only two times of transfers happen during the whole simulation, i.e., input indexing structure from host to device before the simulation starts, and return results from device to host after the simulation ends.

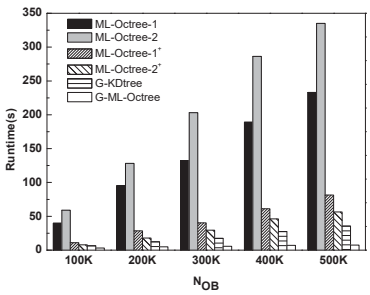


Fig. 11. Evaluating the update performance of G-ML-Octree with 10 timesteps

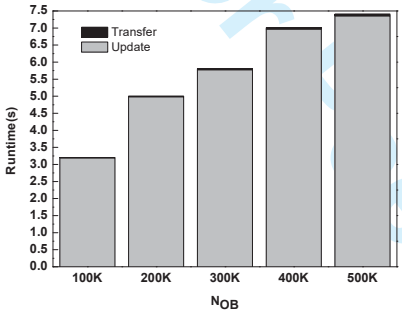


Fig. 12. Evaluating the transfer time of G-ML-Octree with 10 timesteps

Meanwhile, the transfer bandwidth of the GPU device is 16GB/s so that data can be quickly transferred between host and device.

Finally, we evaluate the effectiveness of our load-balancing scheme with 30 timesteps. For convenience, we label the G-ML-Octree with the load-balancing scheme as G-ML-Octree$^+$. As we can see in Fig. 13, compared with G-ML-Octree, G-ML-Octree$^+$ gains encouraging speedups that are 2.3$\times$ on average than G-ML-Octree. The great acceleration performance results from the joint contributions of the task stealing scheme and *dynamic parallelism* of the GPU.

6.4 Distributed G-ML-Octree Evaluation

We conduct experiments to verify the update performance of the distributed G-ML-Octree for large-scale N-body simulations. For comparison, We first build different-size G-ML-Octrees on node #2 in the GPU cluster (see in Table 3) by varying the number of moving objects N_{OB} from 5M to 25M. Then we execute N-body simulations with 10 timesteps on node #2. After that, we set the value of α as one to construct distributed G-ML-Octrees on the GPU cluster and run the same experiments on the distributed G-ML-Octrees. We compared the update time costs of G-ML-Octrees on node #2 and distributed G-ML-Octrees on the GPU cluster. For fairness, we considered the

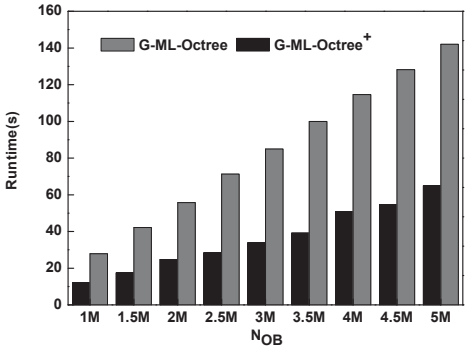


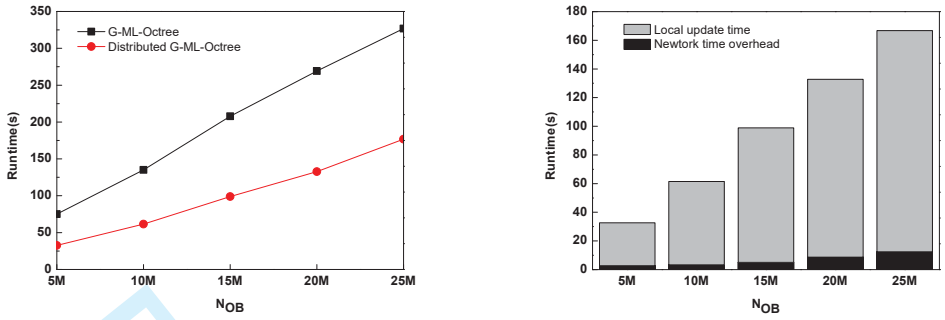
Fig. 13. Evaluating the scheme of load balance with 30 timesteps

maximum network transfer time for migrating moving objects between nodes and the time overhead for synchronization of nodes in a timestep simulation. The implementation of distributed G-ML-Octree employed a MPI message-passing library MPICH 3.2 for communications between nodes.

Fig. 14(a) shows that the distributed G-ML-Octree can averagely gain a speedup of 2.11$\times$ compared with the G-ML-Octree. The major reason for such results is that we dispatch the moving objects to four nodes in the GPU cluster according to their capacities of GPU memory. In our setting, the node #2 holds 12GB GPU memory while the other three nodes own 16GB GPU memory in all. Thus, node #2 is response for about 42.8% workload for simulations while the other three nodes hold 57.2 % workload. Therefore, the GPU cluster can achieve about 2.33$\times$ faster than node #2 in theory. However, the network transfer overheads in the GPU cluster, to a great extent, hinder this theoretical gain. Fig. 14(b) reflects the network time cost accounts for average 6.6% in the total update cost of distributed G-ML-Octree. The results indicate the network time cost only occupies for a relatively small percentage in the whole update time overhead of distributed G-ML-Octree.

7 Conclusions

In this paper, we propose an update-efficient indexing structure for managing massive 3D moving objects in large-scale simulations. The proposed indexing method ML-Octree combines a loose Octree and an update-memo structure to achieve a great update performance. Furthermore, we implement the ML-Octree structure on one Kepler GPU for a higher performance gain. Finally, a distributed ML-Octree has been implemented on a GPU cluster. The experimental results show that (1) ML-Octree can acquire the update performance gain of an order of magnitude,(2) the GPU-aided ML-Octree can accelerate 5.07$\times$ faster than a parallel ML-Octree with 8 CPU threads on average, (3) the load-balancing scheme can improve



(a) Comparison (b) Network time cost

Fig. 14. Evaluating the update performance of distributed G-ML-Octree

the GPU-aided ML-Octree by 2.3× on average, and (4) the distributed G-ML-Octree can efficiently support large-scale simulations.

ACKNOWLEDGMENTS

This work was supported in part by National Science and Technology Major Project of the Ministry of Science and Technology of China (2016ZX05014-003), the China Postdoctoral Science Foundation (2014M552112), China University of Geosciences (Wuhan) (No. 1610491B24).

REFERENCES

[1] M. Dou, J. Chen, D. Chen, X. Chen, Z. Deng, X. Zhang, K. Xu, and J. Wang, "Modeling and simulation for natural disaster contingency planning driven by high-resolution remote sensing images," *Future Generation Computer Systems*, vol. 37, pp. 367–377, 2014.

[2] T. Yu, M. Dou, and M. Zhu, "A data parallel approach to modelling and simulation of large crowd," *Cluster Computing*, vol. 18, no. 3, pp. 1307–1316, 2015.

[3] Z. Wei, R. Wang, and A. Liu, "A new finding of the existence of hidden hyperchaotic attractors with no equilibria," *Mathematics and Computers in Simulation*, vol. 100, pp. 13–23, 2014.

[4] M. S. Warren, "2hot: An improved parallel hashed oct-tree n-body algorithm for cosmological simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 72:1–72:12.

[5] T. Ishiyama, K. Nitadori, and J. Makino, "4.45 pflops astrophysical n-body simulation on k computer - the gravitational trillion-body problem," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[6] J. Makino and H. Daisaka, "Grape-8 – an accelerator for gravitational n-body simulation with 20.5gflops/w performance," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[7] R. Olfati-Saber, "Flocking for multi-agent dynamic systems:algorithms and theory," *IEEE Transactions on automatic control*, vol. 51, pp. 401–420, 2006.

[8] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic, "Interaction ruling animal collective behavior depends on topological rather than metric distance: Evidence from a field study," *PNAS*, vol. 105, pp. 1232–1237, 2008.

[9] Y. Inada and K.Kawachi, "Order and flexibility in the motion of fish schools," *Journal of Theoretical Biology*, vol. 214, pp. 371–387, 2002.

[10] P. L. Wong, M. A. Osman, A. Z. Talib, and K. Yahya, "Modelling of fish swimming patterns using an enhanced object tracking algorithm," *Frontiers in Computer Education Advances in Intelligent and Soft Computing*, vol. 133, pp. 585–592, 2012.

[11] S. Zeng, D. Zhou, and H. Li, "Non-dominated sorting genetic algorithm with decomposition to solve constrained optimisation problems," *IJBIC*, vol. 5, no. 3, pp. 150–163, 2013.

[12] C. Li, S. Yang, and T. T. Nguyen, "A self-learning particle swarm optimizer for global optimization problems," *IEEE Trans. Systems, Man, and Cybernetics, Part B*, vol. 42, no. 3, pp. 627–646, 2012.

[13] C. Li, S. Yang, and M. Yang, "An adaptive multi-swarm optimizer for dynamic optimization problems," *Evolutionary Computation*, vol. 22, no. 4, pp. 559–594, 2014.

[14] H. Samet, J. Sankaranarayanan, and M. A. and, "Indexing methods for moving object databases: Games and other applications," in *SIGMOD*, 2013, pp. 169–180.

[15] J. Barnes and P. Hut, "A hierarchical o(nlogn) force calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[16] J. E. Barnes, "A modified tree code: Don't laugh; it runs," *Journal of Computational Physics*, vol. 87, pp. 161–170, 1990.

[17] K. Kofler, D. Steinhauser, B. Cosenza, I. Grasso, S. Schindler, and T. F. and, "Kd-tree based n-body simulations with volume-mass heuristic on the gpu," in *IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, 2014, pp. 1256–1265.

[18] Y. Xu and G. Tan, "Sim-tree: Indexing moving objects in large-scale parallel microscopic traffic simulation," in *SIGSIM-PADS*, 2014, pp. 51–61.

[19] E. Gaburov, J. Bédorf, and S. P. Zwart, "Gravitational tree-code on graphics processing units: implementation in cuda," *Procedia Computer Science*, vol. 1, p. 1119–1127, 2012.

[20] T. Ulrich, "Loose octrees," *Game Programming Gems*, p. 444–453, 2000.

[21] Y. N. Silva, X. Xiong, and W. G. Aref, "The rum-tree: supporting frequent updates in r-trees using memos," *The VLDB Journal*, vol. 231, p. 719–738, 2008.

[22] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2008, pp. 57–64.

[23] NVIDIA. (2013) NVIDIA Corporation. KEPLER - THE WORLD'S FASTEST, MOST EFFICIENT HPC ARCHITECTURE. http://www.nvidia.com/object/nvidia-kepler.html.

[24] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *ICDE*, 2005, pp. 631–642.

[25] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," *Distributed and Parallel Databases*, vol. 15, pp. 117–132, 2004.

[26] C. BÖhm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-nn queries on data streams," in *ICDE*, 2007, pp. 156–165.

[27] K. Park, "Location-based grid-index for spatial query process-

ing," *Expert Systems with Applications*, vol. 41, p. 1294–1300, 2014.
[28] H.-L. Chen and Y.-I. Chang, "Nine-areas-tree-bit-patterns-based method for continuous range queries over moving objects," *IET Software*, vol. 5, pp. 54–69, 2011.
[29] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," in *ACM SIGMOD Conference*, 2000, pp. 331–342.
[30] M. Pelanis, S. Šaltenis, and C. S. Jensen, "Indexing the past, present, and anticipated future positions of moving objects," *ACM Transactions on Database Systems*, vol. 31, pp. 255–298, 2006.
[31] D. Kwon, S. Lee, and S. Lee, "Indexing the current positions of moving objects using the lazy update r-tree," in *Mobile Data Management*, 2002, pp. 113–120.
[32] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo, "Supporting frequent updates in r-trees: A bottom-up approach," in *International conference on Very large data bases*, 2003, pp. 608–619.
[33] H. Jung, Y. S. Kim, and Y. D. Chung, "Qr-tree: An efficient and scalable method for evaluation of continuous range queries," *Information Sciences*, vol. 274, p. 156–176, 2014.
[34] A. T.Lake and A. S.Kalra, "Grid-based loose octree for spatial partitioning," US Patent 7,002,571 B2, Feb 2006.
[35] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the gpus," *ACM Transactions on Graphics*, vol. 28, pp. 154:1–154:9, 2009.
[36] M. Tarman, K. Wang, J. Killough, and K. Sepehrnoori, "Automatic decomposition for parallel reservoir simulation," in *SPE Reservoir Simulation Symposium*, 2011, pp. 1–17.
[37] H. Rein and S.-F. Liu, "Rebound: An open-source multi-purpose n-body code for collisional dynamics," *ASTRONOMY & ASTROPHYSICS*, vol. 537, 2012.
[38] Codeforge. (2002) loose_octree.cpp. http://www.codeforge.com/read/175558/loose_octree.cpp.html.