Coherent Spherical Range-Search for Dynamic Points on GPUs

Lianping Xing[1], Charlie C.L. Wang[1,2]*, Kin-Chuen Hui[1]

[1]*Department of Mechanical and Automation Engineering, The Chinese University of Hong Kong, P. R. China*

[2]*Department of Design Engineering, Delft University of Technology, The Netherlands*

*(*Corresponding Author. Tel: +31 1 527 88406. Email: c.c.wang@tudelft.nl)*

Abstract

We present an approach to accelerate *spherical range-search* (SRS) for dynamic points that employs the computational power of many-core GPUs. Unlike finding *k approximate nearest neighbours* (ANNs), exact SRS is needed in geometry processing and physical simulation to avoid missing small features. The spatial coherence of query points and the temporal coherence of dynamic points are exploited in our approach to achieve very efficient range-search on AABB-trees. We test our coherent SRS in several applications including point-point-set geometry processing, distance-field generation and particle-based simulation, which are best scenarios to present the spatial and the temporal coherence of spherical queries on dynamic points. On a PC with NVIDIA GTX 660 Ti GPUs, our approach can take 1M queries on 1M dynamic points at a rate of 1600 queries/ms, where 49 neighbours are found on average within the range of 1/100 of the bounding-box's diagonal length. We observe an increase of up to 4x compared with conventional voxel-based GPU searching approaches in the benchmark of particle-based fluid simulation. Moreover, the speedup can be scaled up to 150x when being applied to highly non-uniform distribution of particles in the simulation.

Keywords: Range-search, coherence, dynamic points, parallel algorithm, GPU

1. Introduction

Range-search with a radius r for a query $\mathbf{q} \in \Re^d$ on a set of data points $\mathcal{P}$ is an operation to find all the neighbours $\mathbf{p} \in \Re^d$ within the distance r to $\mathbf{q}$. As the range of search is a d-dimensional sphere, this is called *spherical range-search* (SRS). The technique to conduct SRS efficiently is crucial to the success of many applications, such as point-set geometry processing [1, 2, 3], distance-field generation [4], particle-based simulation [5, 6, 7] and photon mapping [8, 9]. A straightforward way to conduct SRS is the brute force algorithm conducting an exhaustive search among all points in $\mathcal{P}$, which could be slow even when using the computational power of GPUs [10]. Various structures for acceleration have been developed to improve the efficiency of this process. kd-trees are one of the widely used hierarchical structures for low-dimensional neighbour search (e.g., ANN in [11] and FLANN in [12]). Other data structures such as BV-trees, trapezoidal maps, range-trees, and Gh-trees, are alternative choices. A comprehensive lecture on multidimensional and metric data structures can be found in the book by Samet [13]. In this paper, we develop acceleration techniques for point based search queries for low-dimensional dynamic points on GPUs. The main contribution is a highly parallel approach employing the temporal coherence of dynamic points and the spatial coherence of query points. .

Temporal Coherence: kd-tree is widely used in low-dimensional neighbouring search because of its efficiency. However, the construction time remains prohibitively expensive. The problem becomes more serious when the data points in $\mathcal{P}$ dynamically move during the computation. This is quite a common scenario in the applications of geometry processing and physical simulation. For example in Fig. 1(a), a kd-tree must be constructed again after moving the data points in $\mathcal{P}$. In literature, many attempts have been made to speedup the construction of kd-trees either on the multi-cores of CPUs [14, 15, 16] or on the many-cores of GPUs [17, 18, 19]. However, the speed of rebuilding a kd-tree is still expensive. Therefore, the voxel-based search is popular in the approaches of particle-based simulation (e.g., [20]), where the voxels containing dynamic points can be easily rebuilt by using the highly parallel sorting algorithm [21]. Inspired by the work of [22], the hierarchy of *axis aligned bounding boxes* (AABBs) is used here to conduct efficient SRS on dynamic points. After construction, an AABB-tree can be updated by a highly parallel bottom-up refitting scheme – i.e., only the shapes of AABBs, and not the structure of a tree, are modified (see Fig. 1(b)). The temporal coherence of keeping dynamic points in the same tree-structure can be employed to reduce the computational cost. We also investigate a heuristic method to activate the process of tree rebuilding based on the volume variation of AABB.

Spatial Coherence: In most applications, the queries in SRS are independent of each other, and are thus able to be performed in parallel. However, as will be analyzed in Section 3, a large portion of the queries share the similar patterns of traversal on the AABB-tree, which results in many redundant traversals on the tree. In our approach, this spatial coherence is employed to develop a new querying scheme that increases speed by 2 to

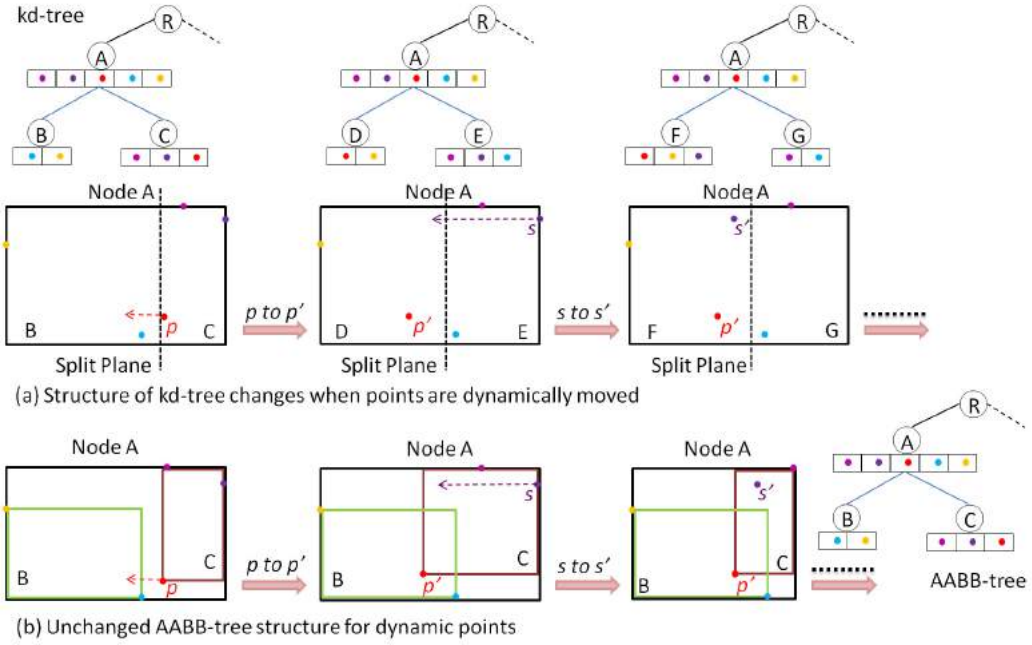


Figure 1: Comparison of the evolution on (a) kd-tree and (b) AABB-tree for dynamic data points. The topology of a kd-tree has to be changed, while an AABB-tree can change only its AABBs while keeping the same structure. This is a temporal coherence to be used in an efficient tree update for dynamic points. As a result, efficient SRS of dynamic points can be conducted on AABB-trees, together with the step of rebuilding when it is necessary.

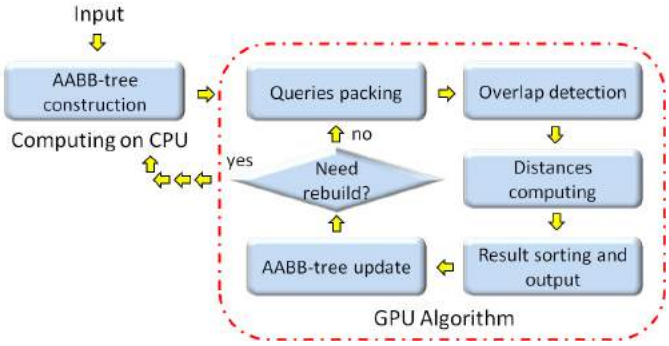


Figure 2: Pipeline of our coherent SRS algorithm.

4 times compared with the primary scheme of GPU-based parallel SRS. Example applications, including point-set geometry processing, distance-field generation and particle-based simulation, are used as best candidates to demonstrate the performance of our method.

Main Results: We address the problem of highly parallel SRS for dynamic points with the help of AABB-trees. Here AABB-tree is chosen in our implementation because of the trade-off between simplicity and tightness in bounding. In our setup, all the queries are assumed to have the same searching radius, r; however, it is easy to extend our approach to let different queries have different search radii. To fully exploit the respective superiorities of CPUs and GPUs, the construction of the AABB-tree is conducted on the CPU and then copied into the graphics memory for the highly parallel SRS. In the concurrent search step, query points are first packed into bins by the leaf-nodes of the AABB-tree. For each leaf-node v containing a set of query points Q_v, its swept volume with a sphere of radius r is used to conduct overlap detection. All data points in the overlapped AABBs are potential SRS results of queries in Q_v. After that, distances between the potential results and the query points are evaluated. Finally, the real SRS results are picked out by using the *scan* and *sort* primitives [23]. When the data points in an AABB-tree are moved in small distances, AABBs on the tree are updated in a bottom-up manner by using the temporal coherence of keeping dynamic points in the same tree-structure. Meanwhile, the volume variations of AABBs are evaluated. When the maximal volume change has exceeded a threshold (e.g., $> 10\times$), the AABB-tree is rebuilt. When data points are moved at a reasonable speed, this mechanism works well. The computational pipeline of our approach is shown in Fig. 2. The effectiveness of our approach will be demonstrated in the applications of point-set geometry processing, distance-field generation and particle-based simulation. Note that, there are other hierarchical structures can be used for dynamic environment – such as spherical tree used in [24].

In short, our technical contribution is an integrated framework to borrow the capability of high performance computing provided by dynamic hierarchical tree traversal/update and packet query. The framework is implemented on a simple but effective AABB-tree and demonstrated by a few examples in geometric modeling and physical simulation, where the queries are easy to group and cluster. Actually, there are many other applications in solid and physical modeling sharing the same nature of point queries (e.g., [25, 26, 27]), which can all be benefit from this work.

Our approach shows a better performance than the current state-of-the-art methods. When testing on a PC with NVIDIA

GTX 660 Ti card, more than 1600 queries/ms can be performed for 1M queries on a set of 1M dynamic points reporting 49 neighbours/query on average. We observe up to two order performance improvement over the sequential algorithm [11] and up to a 4× increase in speed over the voxel-based [20] and the shifted-sorting based search [28] running on GPUs. Unlike the ANN library in [11] that can report the exact results of SRS[1], a search based on shifted-sorting may miss the real nearest-neighbours.

The rest of this paper is organized as follows. After reviewing the related work in Section 2, the construction of AABB-trees on CPU and a primary hierarchical traversal on GPU are briefed in Section 3. After analyzing the problem of the primary scheme in Section 4.1, Section 4.2 presents the coherent SRS algorithm running on many-core GPUs. Section 4.3 introduces the parallel algorithm for updating AABBs on a tree. Lastly, experimental results and the applications of our approach are presented in Section 5.

2. Related Work

The spherical range-search is a common tool that has a variety of applications in machine learning, database, computer vision, geometry processing, physical simulation, design and manufacturing, etc. The problem has been studied for many years and many related approaches can be found in the literature where a review can be found in [29, 13]. Giving a comprehensive review, where different type of hierarchies and search queries must be compared and discussed, is beyond the scope of this paper. In this section, we only review the recently developed algorithms of low-dimensional range-search and ANN search that run on GPUs, which are most relevant to our work.

To remove the redundant distance calculations and queries in the brute force *k nearest neighbour* (*k*-NN) search [10], Green partitions the space of query into voxels [20]. For a query point $\mathbf{q}$, only data points inside the voxel containing $\mathbf{q}$ and the neighbouring voxels are checked instead of checking all the data points. The voxels containing dynamic points can be efficiently constructed by using the parallel sorting algorithm. Therefore, this strategy is popular in particle-based simulations (e.g., [5, 30]). Recently, an out-of-core SRS algorithm [31] (also called ϵ-NN) was proposed in the context of particle simulation. This also partitions the space of data points uniformly. The common drawback of a uniform partition based approach is that the performance of queries degrades rapidly where there is a non-uniform distribution of data points.

A kd-tree is usually employed in low-dimensional range-search as a structure adaptive to the distribution of points. To borrow the computational power provided on GPUs, Zhou et al. [17] developed the first GPU-based algorithm for kd-tree construction. Their algorithm exploits the GPU's streaming architecture and can construct kd-trees for moderate sized models at a very fast speed. However, the kd-trees constructed by their algorithm are not balanced as spatial median splitting is used for upper-level nodes. Moreover, their algorithm consumes a large amount of memory so that it cannot be applied to large models. Qiu et al. [32] developed an ANN search running on GPUs based on the approach of Arya and Mount [33]. The kd-tree is built on the CPU and then transferred to the GPU before running ANN. In their approach, each thread traverses the tree for one query point. The spatial coherence between query points has not been exploited to improve efficiency. Moreover, kd-trees do not work well in the scenario of dynamic points as the hierarchy has to be reconstructed after each iteration of point movements.

Bounding volume hierarchies (BVHs) are widely used to accelerate intersection computations for ray tracing, collision detection, visibility culling, and other similar applications. There is an extensive literature on fast computation of BVHs [34, 35, 36, 37]. Lauterbach et al. proposed a method in [35] to construct BVHs with the help of Morton codes. After that, the proximity queries can be undertaken efficiently on GPUs [38]. However, they did not take advantage of the coherence on queries to improve the efficiency of searching. To further accelerate the algorithms running on GPUs, coherence has recently been employed in [39] and [40] for the distance-field evaluation and the fast collision culling respectively. There are many different types of bounding volume primitives can be used to develop fast query by BVH – such as, spherical tree [24], OBB-tree [41], *k*-DOP tree [42], ellipsoid-tree [43], etc. In this paper, we employ the hierarchy of AABB to conduct SRS as it is easy to construct, fast in traverse and allows quick refitting for dynamic points. Comparing to other BVHs (e.g., sphere-tree [24]), AABB-tree is chosen by the trade-off between simplicity and tightness in bounding. Most recently, Li et al. [28] introduced a GPU-based ANN algorithm based on Morton codes and shifted sorting. They align query points together with data points in the shifted sorting to speedup the search of *k*-ANN. Unfortunately, such an algorithm for ANN search may miss some real nearest neighbours. Moreover, the comparison shows that our method proposed in this paper outperforms ANN based on shifted sorting [28].

Another thread of research focuses on solving the *k*-NN problem in high-dimensional space. Pan et al. [44] use a GPU-based *locality sensitive hashing* (LSH) algorithm to perform approximate *k*-NN search in high-dimensional spaces. Sundaram et al. [45] developed a new parallel variant of the LSH algorithm that supports similarity search on massive streaming data sets across multiple nodes. Nevertheless, the LSH-based algorithms do not perform as well as the hierarchy-based range search approaches in relatively low-dimensional spaces.

By analyzing the nature of the algorithms, it becomes apparent that hierarchical structure based methods should be faster than the voxel-based and the LSH-based approaches for range-search in low-dimensional problems. We observe the same results in the experimental tests undertaken in Section 5. When considering the construction time, the kd-tree is slower than LSH while the most efficient one is the sorting based construction of voxels. However, we will show that using temporal coherence to update an AABB-tree without changing its structure

[1] In all our comparisons with ANN [11], the exact and comprehensive neighbours are generated by an option provided in the library.

is even faster than the voxel-based approach. And our study shows that, in our applications, trees need to be rebuilt only after tens of iterations of point movement.

In real-time ray tracing, query-packets are usually employed to improved the efficiency of computation (ref. [46, 47, 48, 49]). A common strategy is to first group coherent rays, bound them within a tight frustum, and then perform the same traversal for the whole group of rays. In this paper, we extend this idea to SRS. Besides of this spatial coherence, we also employ the temporal coherence of dynamic points to reduce the computational cost in our method.

3. Preliminary

This section first presents the method for constructing an AABB-tree $\mathcal{T}$ from a set of data points, $\mathcal{P}$. After that, the primary scheme of GPU-based SRS on $\mathcal{T}$ is introduced.

3.1. Construction of hierarchy

An AABB-tree is a binary tree of AABBs. The construction of an AABB-tree for a set of data points in $\Re^d$ is similar to that for a kd-tree. The space spanned by $\mathcal{P}$ is recursively subdivided into disjointed hyper-rectangular regions in a top-down manner. When constructing a node v on the tree, the AABB of points contained by v is first computed. Then, the points are reordered to be split into two children nodes of v by a well-chosen partitioning plane. The process is continuously performed until each subset contains not more than a certain number of points (e.g., 20 in our implementation). This number is called *bucket size*.

To fully exploit the respective superiorities of CPUs and GPUs, we chose to construct an AABB-tree on the CPU and then copy it into the graphics memory to undertake range-search. To have an efficient SRS algorithm running on GPUs, we need to bound the length of the search path on the trees. An unbalanced tree leads to a longer searching path and also results in highly unbalanced workload among different threads. We chose to construct a balanced AABB-tree to solve these problems. Specifically, when splitting the data points of a tree-node v to generate v's children, the median plane is searched to partition n points into two subsets with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ points respectively. This guarantees that we can obtain an optimally balanced tree with a bound of depth as $\lceil \log_2 n \rceil$. Our implementation of tree construction follows the algorithm of ANN library [11], where the Hoare's selection algorithm [50] is used to find the splitting plane with the complexity of $O(n \log n)$ in the best case and $O(n^2)$ in the worst case.

After constructing a balanced AABB-tree $\mathcal{T}$, all the nodes of $\mathcal{T}$ are stored into a 1D array Θ according to the order of *breath-first traversal* (BFT) of the tree. As $\mathcal{T}$ is a balanced binary tree, the index of a node in Θ can be used to find out the locations of its parent and children nodes easily. Θ is copied to the global memory of GPUs to undertake the spherical range-search in parallel. Note that, as nodes are stored in the order of BFT, the nodes at the same level of $\mathcal{T}$ are stored together in Θ. This arrangement benefits the step of hierarchical update in Section 4.3.

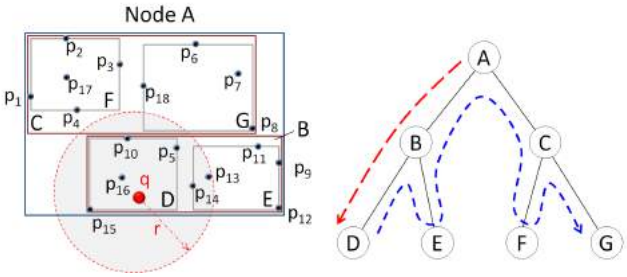


Figure 3: An illustration of the AABB-tree traversal for spherical range searching: For a given query $\mathbf{q}$, the leaf-node 'D' containing $\mathbf{q}$ is first found by DFS (the path is shown in red). Then, a back-traverse is taken to check all the data points in the leaf-nodes that intersect the sphere centered at $\mathbf{q}$, where the path of back-traverse is shown in blue. In this example, $\mathbf{p}_5$, $\mathbf{p}_{10}$, $\mathbf{p}_{13}$, $\mathbf{p}_{14}$, $\mathbf{p}_{15}$ and $\mathbf{p}_{16}$ are extracted as $\mathbf{q}$'s neighbors.

3.2. Hierarchical search: a primary scheme

Among the variety of range-search algorithms, the approach of Arya and Mount [33] is widely employed as it is more efficient than the conventional top-down BFS [29]. When running on a balanced binary tree, the size of stack is bounded by the tree's depth. It is an important property of having a constant memory consumption to fit the architecture of GPUs. Taking the same strategy as [33], we develop a primary scheme to conduct the SRS queries in parallel on an AABB-tree (instead of a kd-tree in [33]).

Data points in the leaf-nodes whose AABB extents intersect the querying sphere centered at $\mathbf{q}$ (with the radius r) are found to check their distances to $\mathbf{q}$. Therefore, the key is how to find these leaf-nodes efficiently. For a query point $\mathbf{q} \in Q$, a *depth-first search* (DFS) can be used to locate the leaf-node which *contains* $\mathbf{q}$. However, it is possible to have a query point outside all AABBs meanwhile having data points covered by its spherical range. To overcome this problem, a weak form of containing is defined as *touching*.

Definition 1 A query point $\mathbf{q}$ with searching radius r is defined as *touching* a node v on a tree $\mathcal{T}$ when $d(v.aabb, \mathbf{q}) \leq r$.

Here $d(v.aabb, \mathbf{q})$ defines the Euclidean distance between $\mathbf{q}$ and the AABB of v. A query point can *touch* more than one leaf-nodes, in which case the first detected node will be used in DFS. After taking a DFS to find a leaf-node touching $\mathbf{q}$, all the nodes on the path of this DFS and their subtrees are checked to see if their AABB extents intersect the querying sphere. This can be realized by a back-traversal with the help of a stack with its size bounded by the tree's depth. All the leaf-nodes on a subtree overlapping the querying sphere can be found by a DFS. Lastly, distances between $\mathbf{q}$ and the points in these overlapped leaf-nodes are evaluated to get the real results of SRS. An illustration is given in Fig.3, and a pseudo-code of this primary scheme of GPU-based SRS is given in **Algorithm** *PrimaryGPUBased-SRS*. A stack is allocated for each querying thread with its maximal size the same as $\mathcal{T}$'s depth. The final searching results are stored in an array with the help of atomic operators on GPUs.

Algorithm 1: PrimaryGPUBasedSRS

Input: AABB-tree $\mathcal{T}$, query set Q, radius r
Output: neighbours within radius for each $\mathbf{q} \in Q$

```
1  foreach q ∈ Q in parallel do
2      Find a leaf node v that touches q by DFS;
3      Check all points in v to find the real neighbors of q;
4      Set cur = v;
5      while cur.parent ≠ ∅ do
6          parent = cur.parent;
7          Find the brother node srt of cur by parent;
8          if d(srt.aabb, q) ≤ r then
9              Apply DFS on a subtree rooted at srt to find
               out all neighbours of q;
10         end
11         cur = parent;
12     end
13 end
```

4. Coherent Searching and Updating

This section analyzes the problem of primary scheme and proposes a coherent searching algorithm that can achieve a 2 to 4 times increase in speed. After that, an efficient algorithm is presented to update the AABB-tree by using the temporal coherence of dynamic data points. The criterion on rebuilding a tree is also studied.

4.1. Problem of primary scheme

The primary scheme for GPU-based SRS is easy to implement. However, it has not taken full advantage of the high parallelism in the architecture of modern graphics hardware for the following reasons.

- First, the searches of every query point are running independently, where each thread needs a stack to store the intermediate information of traversal. The size of the graphics memory on consumer-level hardware is limited (i.e., usually less than 4GB). For the cases with a massive number of query points, the hierarchical traversals have to be subdivided into many segments to be run in different rounds.

- Second, different threads traversing the tree with different paths could have tremendous variation in lengths. According to our observation, the maximal length of a path could be more than three times the average length. This leads to significantly unbalanced workload among the many-cores of GPUs.

To improve the situation of unbalanced workload on different threads, a lightweight algorithm akin to [38] is developed for dynamic workload balancing. After implementing this strategy, we observe about a 20% performance improvement on the primary scheme when taking 1M queries on a set of 1M data points.

More importantly, the SRS algorithm can be more efficient if the number of stack-based traversals can be reduced. In the following subsection, the coherence of paths in back-traverse are considered to merge similar paths so that we can reduce the number of traversals.

4.2. Coherent SRS

The coherent spherical range-search introduced below consists of three phases: 1) packing of queries, 2) common back-traversal and 3) result extraction. Basically, the first phase merges queries with similar traversal patterns, the second phase finds all potential neighbours, and the last phase figures out the real neighbours of every query point. A pseudo-code of our coherent SRS can be found in **Algorithm** *CoherentGPUBased-SRS*.

4.2.1. Packing of queries

Our investigation finds that the query points falling in the same leaf-node of $\mathcal{T}$ have very similar traversal patterns for SRS (see Fig.4 for an example). Statistics show that, when query points are well aligned with data points stored in $\mathcal{T}$, up to 99% of the tree traversals can be merged into 1% common traversals by packing the query points according to this spatial coherence. In practice, we first apply the *depth-first search* (DFS) to each query point, $\mathbf{q}$, to find the leaf-node, v_q, containing (or touching) $\mathbf{q}$. After that, a highly parallel sort using the IDs of leaf-nodes as keys is applied to align the query points in the same leaf-nodes together [23]. Pseudo-code for the query packing can be found in lines 1-8 of **Algorithm** CoherentGPUBasedSRS. Note that, the set of data points $\mathcal{P}$ can also be the set of query points Q (i.e., $\mathcal{P} \equiv Q$). In these cases, we can bypass the DFS step in line 4 to further speedup the computation.

4.2.2. Common back-traverse with swept volume

For all query points in a leaf-node v_q, a common back-traverse is applied to find candidates for the result of SRS. For all the query points in v_q, the candidates of SRS results must fall in a swept volume around the AABB of v_q by a sphere with radius r (as also illustrated in Fig.5).

Definition 2a The swept volume of a leaf-node v_q for SRS is defined as

$$v_q.swp = \{\mathbf{p} \mid \exists \mathbf{v} \in v_q.aabb, \|\mathbf{p} - \mathbf{v}\| \leq v_q.r\} \quad (1)$$

where $v_q.r = r$ represents the radius of the sweeping sphere and $v_q.aabb$ denotes the AABB of v_q.

For a leaf-node v_q, if it is touched by a query point $\mathbf{q}$ not contained by $v_q.aabb$, $\mathbf{p} \in \mathfrak{R}^d$ can only be excluded when $\|\mathbf{p} - \mathbf{q}\| > 2r$. Therefore, its swept volume needs to be enlarged. It is important to note that bound of the swept volume introduced here based on AABB is much tighter than spherical primitives in sphere tree (see the volume of node C in Fig.5 for an example).

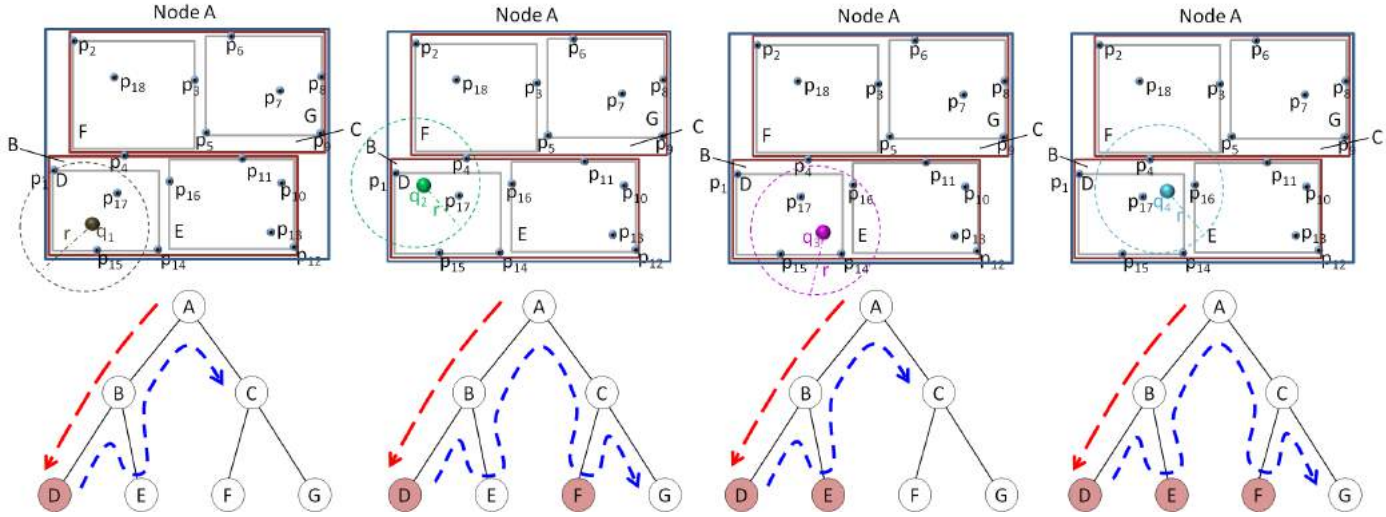


Figure 4: Query points touched by the same leaf-node of a tree usually have similar traversal patterns on the tree. Here, red paths are the traversal to find the leaf-node touching a query, and blue curves illustrate the paths of back-traverse. As these four queries share similar traversal patterns, they can be packed into one common back-traverse. AABBs of the leaf-nodes displayed in red intersect the spherical ranges of query points, and the data points in these nodes are the candidates of search results.

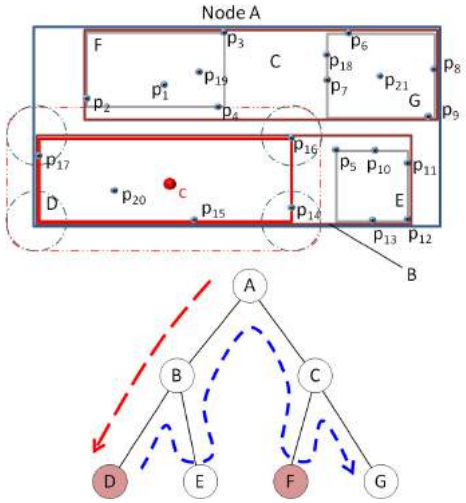


Figure 5: An illustration for the common back-traverse with swept volume. The swept volume of a sphere with radius r along the AABB of the leaf-node D is employed to detect the leaf-nodes containing candidates of SRS results – data points in the nodes D and F are obtained as candidates in this case. The blue curve shown on the tree is the path of back-traverse.

Definition 2b For a leaf-node v_q touched by any query point that is *not* in $v_q.aabb$, its swept volume $v_q.swp$ is defined by Eq.(1) but with

$$v_q.r = 2r. \tag{2}$$

By this, candidates for the result of SRS must be in the leaf-nodes that intersect with $v_q.swp$. Overlap between any leaf-node v_i and the swept volume $v_q.swp$ can be conservatively detected by the overlap between their AABBs.

Remark 1 For any leaf-node v_i, if its AABB has no overlap with the AABB of a swept volume $v_q.swp$, v_i and $v_q.swp$ have no overlap.

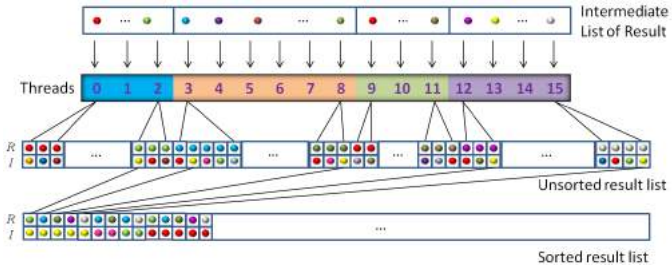


Figure 6: An illustration for the result extraction step of our coherent SRS algorithm – the number of threads allocated is according to the number of entries in the intermediate list S.

According to this remark, our back-traverse takes a DFS to find all the leaf-nodes, v_is, that have $(srt.aabb \cap v_q.swp.aabb) \neq \emptyset$ with the AABB of $v_q.swp$ denoted by $v_q.swp.aabb$. Data points in the overlapped leaf-nodes are the candidates of SRS results. The common back-traverses for all leaf-nodes that contain query points are run in parallel. Pseudo-code for the back-traverse can be found in lines 10-22 of **Algorithm** CoherentGPUBasedSRS.

All the data points in the leaf-nodes, v_is, overlapped with $v_q.swp.aabb$ are copied into a 1D array S to be further refined to find the real SRS results. Note that, as the candidate points for different v_qs obtained on different GPU cores can be added into S at the same time, the atomic operator provided by modern GPUs is employed to solve the *read-modify-write* (RMW) problem. The data points are stored in S together with an ID key indicating from which querying leaf-node v_q they are generated (in line 18 of **Algorithm** CoherentGPUBasedSRS).

4.2.3. Result extraction

In this final phase of coherent SRS, the real results within the querying range r of points in Q will be extracted. For a

Algorithm 2: CoherentGPUBasedSRS

Input: AABB-tree $\mathcal{T}$, query set Q, radius r
Output: neighbors within radius for each $\mathbf{q} \in Q$

1 // Phase I: Packing of Queries
2 Initialize two empty lists, I and F;
3 **foreach** $\mathbf{q}_i \in Q$ ***in parallel*** **do**
4 Find the leaf-node v that *touches* $\mathbf{q}$ by DFS;
5 $I \leftarrow i$ and $F \leftarrow v.id$;
6 **end**
7 Sort I keyed by F to group the queries in the same leaf-node together;
8 Compact F into a reduced list $\tilde{F}$ by removing the repeated entries;
9 // Phase II: Back-traversal with swept volume
10 **foreach** $f_q \in \tilde{F}$ ***in parallel*** **do**
11 Get the leaf-node v_q according to the ID, f_q;
12 Add all points in v_q into a list S as the candidates of SRS results;
13 Set $cur = v_q$;
14 **while** $cur.parent \neq \emptyset$ **do**
15 Find the brother node srt of cur;
16 **if** $(srt.aabb \cap v_q.swp.aabb) \neq \emptyset$ **then**
17 Apply DFS on the subtree of srt to find all the leaf-nodes v_i that have $(v_i.aabb \cap v_q.swp.aabb) \neq \emptyset$;
18 All points in the these nodes are added into S together with the ID of v_q;
19 **end**
20 $cur = parent$;
21 **end**
22 **end**
23 // Phase III: Result extraction
24 **foreach** *data point* $\mathbf{p}_i \in S$ ***in parallel*** **do**
25 Get the querying leaf-node v_q of $\mathbf{p}_i$;
26 **foreach** *query point* $\mathbf{q}_j \in v_q$ **do**
27 **if** $\|\mathbf{p}_i - \mathbf{q}_j\| \leq r$ **then**
28 $R \leftarrow \mathbf{p}_i$ and $I \leftarrow \mathbf{q}_j.id$;
29 **end**
30 **end**
31 **end**
32 Sort the list R keyed by the list I;
33 Compact I by removing the repeated entries to build the offset table T for accessing R;
34 **Return** R and T;

point $\mathbf{p}_i$ in the intermediate list S, we can easily obtain its corresponding querying leaf-node v_q by the ID key stored with it (details can be found above). We then check the distance between $\mathbf{p}_i$ and all query points $\mathbf{q}_j \in v_q$. When $\|\mathbf{p}_i - \mathbf{q}_j\| \leq r$, $\mathbf{p}_i$ is added into a resultant list R and $\mathbf{q}_j$'s ID is added into a querying index list I in the same order as $\mathbf{q}_j$ in R. After that, the points in R are sorted by using the entries of I as the keys. Now all the resultant data points of SRS are listed in R in the order of querying points' IDs. Finally, the offset table can be constructed by removing repeated entries in the sorted I so that the results in R can be accessed according to the ID of querying points. Pseudo-code for the result compaction can be found in lines 23-34 of **Algorithm** CoherentGPUBasedSRS (see also Fig.6 for an illustration). Note that each intermediate resultant point is allocated with an independent thread in this phase to build the unsorted list of resultant points, R. As a result, each thread (according to an entry in S) takes a smaller amount of work than allocating threads according to the leaf-nodes containing query points. This leads to better balanced workloads between different threads.

Extension for independent radii: Our coherent SRS can be extended to support independent searching radii r_j on each query $\mathbf{q}_j$. When this is required for an application, we can first replace r in Eq.(1) (or Eq.(2)) by the maximal radius of all query points contained (or touched) by the leaf-node v_q. Then, in the step of result extraction, $\mathbf{p}_i$ is added into R only when $\|\mathbf{p}_i - \mathbf{q}_j\| \leq r_j$. By this extension, different query points are allowed to have different searching radii.

4.3. Hierarchical updating

The structures of AABB trees make them easy to use in the scenarios of dynamic models where the positions of primitives change over time [22]. Instead of rebuilding the AABB tree after every modification of data points, the temporal coherence of data points can be employed to update the tree by only refitting the AABBs. Specifically, the new extents of AABBs on the leaf-nodes are first updated by the new positions of data points. Then, the AABBs of non-leaf-nodes are updated level by level in bottom-up order. For any node v on the i-th level of an AABB tree, the AABB of v can be obtained directly from the AABBs of its two children by min and max comparisons. Note that, the update of a node's AABB will *not* affect the AABBs of any other nodes located at the same level of the tree. They are independent of each other, a feature which makes them ideal for a highly parallel implementation. We can update the AABBs of the nodes at the same level in parallel without the RMW problem.

All the nodes of a tree $\mathcal{T}$ are stored in a 1D array, Θ. As the nodes of $\mathcal{T}$ are packed in Θ by the order of BFT, nodes at the same level have been aligned together in Θ. With the help of an offset table that can be generated during the construction of Θ, the AABB refitting kernel (by min and max comparisons) can be efficiently applied to the nodes at the same level of $\mathcal{T}$. When the nodes are randomly aligned in Θ, the refitting kernel takes a lot of redundant effort to extract the nodes on the same level. A comparison on the efficiency has been given in Fig.7.

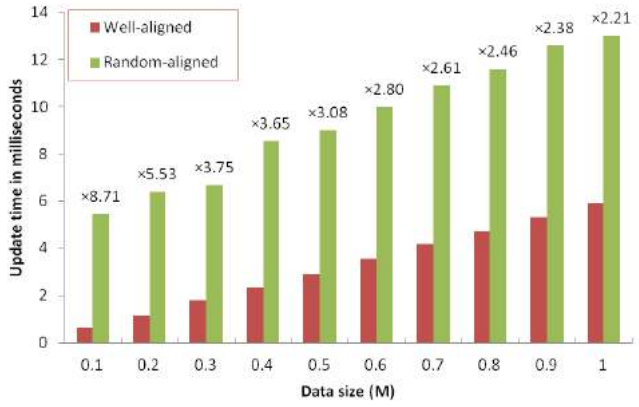


Figure 7: Well-aligned tree nodes by placing together the nodes on the same level can significantly improve the efficiency of hierarchical updating.

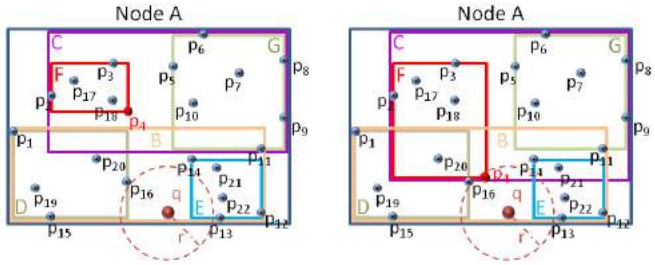


Figure 8: After moving the red point $\mathbf{p}_4$ to a new position (see right), the original AABB-tree on the left is updated to the one on the right where the AABB of the node F and its parents, C and A, are recomputed accordingly. For the same SRS query centered at $\mathbf{q}$, three more nodes C, F and G need to be visited in the back-traverse phase of query.

4.4. Rebuilding AABB-tree

Due to the position change of data points, the AABBs in a tree updated by the above method may have a large volume of overlap. An extension of overlap on AABBs can lead to more visits of nodes during the back-traverse (see Fig.8 for an illustration).

Among all the applications tested in this paper, the particle-based simulation is an application where the data points can move promptly. As a result, the AABBs may change significantly so that the issue of overlap between AABBs becomes serious enough to affect the performance of SRS. The volume of an AABB can be up to 150 times its original AABB after 400 steps of iteration. Study has been undertaken of such a scenario with prompt point movement. For parallel computing on GPUs, the performance of the whole algorithm is usually determined by the slowest thread. Therefore, we measure the *maximum volume variation ratio* (MVVR) versus the change of updating rate in *frames per second* (FPS) during the simulation. As shown in Fig.9, the value of FPS does not change significantly when MVVR increases at the very beginning of simulation. However, when MVVR becomes larger and larger, FPS starts to drop steeply. The AABB-tree then needs to be rebuilt by the CPU-based construction algorithm. We rebuild the AABB-tree when MVVR is greater than a threshold μ. In our implementation, $\mu = 10$ is used. This GPU/CPU hybrid

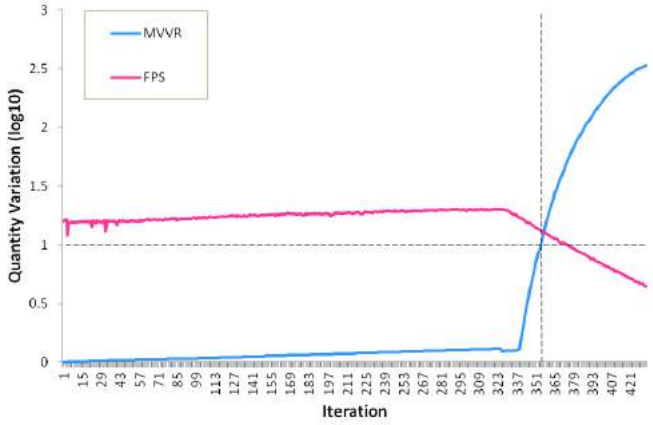


Figure 9: Study is undertaken of the scenario with prompt point movement – particle simulation. The changes of MVVR and FPS during the simulation are plotted with the vertical axis presenting $\log_{10}$ values.

approach gives a very good result in practice (see the experimental tests and comparisons in Section 5). Our approach can be observed to deliver an increase in speed of up to 5.6× over the state-of-the-art method (see the table in Fig.18).

The variations of AABBs in other applications (e.g., point-set geometry processing) are much smaller than that in the particle simulation. Therefore, overlap between AABBs is trivial and we rarely rebuild the AABB-tree.

5. Results and Discussion

We have implemented the coherent SRS using C++ together with the NVIDIA CUDA library. The performance of our approach is tested on a PC equipped with 3.4GHz CPU + 8GB RAM and a NVIDIA GeForce GTX 660 Ti graphics card with 3GB memory. In our implementations, the kernels are called by using 32 blocks and 256 threads/block. *Structure of Arrays* (SoA) instead of *Array of Structures* (AoS) is employed to get optimal GPU cache performance.

5.1. Experimental Tests

Experimental tests have been conducted to verify the performance of our coherent SRS, comparing it with a variety of prior approaches. In these tests, both the data points in $\mathcal{P}$ and the query points in $\mathcal{Q}$ are randomly drawn in $\Re^3$ in the range $[-1.0, 1.0]$. The bucket size for both the CPU-based kd-tree construction in [11] and our AABB-tree construction is set to 20. The parameter for activating the tree rebuilding is set as $\mu = 10$ in all our tests. Our coherent SRS is compared with the state-of-the-art method in different aspects below.

Querying time

To simulate dynamic data points, we randomly move all points along three axes within a fixed range after each iteration. In each test, the iteration of random movement followed by SRS is taken in 50 steps, and the average querying time is measured by using the total time divided by 50. This actually

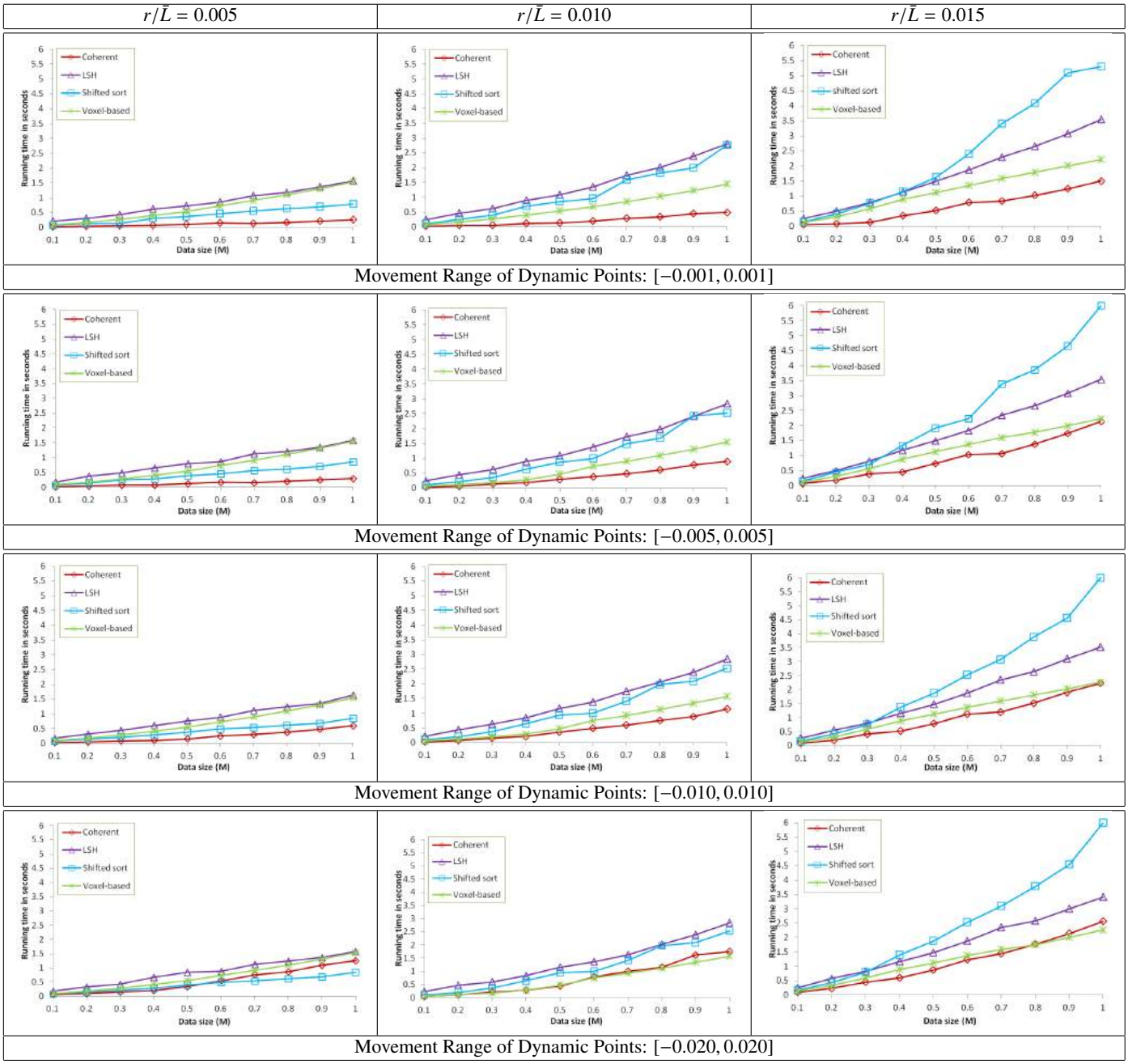


Figure 10: Statistics of the querying time on different approaches for SRS on dynamic points in different ranges of movements and different search radii r (w.r.t. the diagonal length $\bar{L}$ of all points' bounding box), where our coherent approach is compared with the LSH approach [44], the shifted-sorting [28] and the voxel-based search [20]. All are running on GPUs.

includes the time of tree-updating and tree-rebuilding in our coherent SRS. $\mathcal{P}$ and $\mathcal{Q}$ have the same number of points in these tests. Tests are undertaken on different ranges of movement and different radii of SRS.

The querying time of our coherent SRS is compared with that of the other approaches running on GPU. Data sets with different number of points ranging from 100k to 1M are tested. Statistics are illustrated in Fig.10. It can be seen that our approach outperforms others in relative smaller range of point movement – the first three rows in Fig.10. In these scenarios, only tree-updating is needed in most steps. When different searching radii are used, all methods become slower. However, our method is less affected by large searching radii than other approaches. When the points are moved in tremendous speeds (e.g., as shown in the last row of Fig.10), AABB-tree used in our approach needs to be rebuilt in every step. The performance could become worse than one of the other three approaches in some cases. Nevertheless, considering about the efficiency in all cases, our method still outperforms others as a whole. The coherent SRS method has been observed up to 6.72× faster than the voxel-based approach [20], up to 6.69× faster than shift-sort [28] and up to 13.2× faster than *locality sensitivity hashing* (LSH) [44]. When comparing with the CPU-based ANN [11], 10.4 – 53.7 times increase in speed is observed for our coherent SRS approach. Note that, although the ANN library is mainly used for the k-ANN search, it also provides an option for exact SRS. Our tests are run using this option. The shift-sort ANN is tested by setting the number of NN as the average number of NN obtained in our SRS. For the LSH-based query, we modify the code provided by [44] to make it capable of computing SRS results.

	Querying Time (sec.)		Usage of Memory (MB)	
$r/\bar{L}$	All Points	NN Only†	All Points	NN Only†
0.005	0.101	0.091	18.33	17.13
0.010	0.155	0.144	31.34	21.78
0.015	0.233	0.206	60.47	28.50
0.020	0.335	0.279	112.7	37.62
0.025	0.475	0.396	194.7	49.41
0.030	0.660	0.531	312.9	64.18
0.035	0.871	0.683	473.9	82.27
0.040	1.133	0.872	683.5	103.9
0.045	1.449	1.128	947.5	129.5
0.050	1.848	1.397	1,271	158.9

†Different querying time and memory consumption can be observed when only the nearest neighbor (NN) is searched.

Table 1: Statistics when increasing the searching radius r

Searching radius

Now we study the performance of our coherent SRS with different search radii. Starting from 0.005 of the bounding box's diagonal length, we incrementally enlarge the search radius on a set of $300k$ data points with $300k$ queries. The statistics of querying time and memory usage are listed in Table 1. It is easy to find that the querying time increases slightly faster than the radius (i.e., about 18.3 times when the radius is increased by 10 times). The bottleneck is memory usage. This is because the number of resultant points in SRS could increase by a factor of τ^3 when increasing the radius by τ times. This prevents applying SRS with large radii. In practice, we actually observe better results. As shown in Table 1, about 69.3× memory is used after increasing r by 10×. Nevertheless, this is not a problem for those SRS applications which only need to report the nearest neighbour (e.g., distance-field evaluation). As shown in Table 1, the increase in memory usage is very slow in such cases.

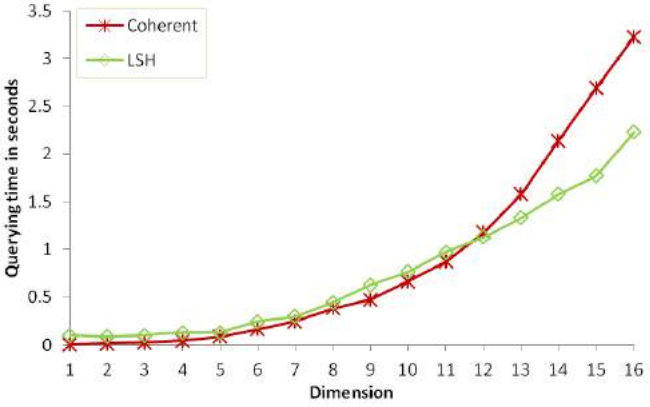


Figure 11: For high-dimensional data sets, LSH [44] has better performance. The curves show the comparison on the querying times.

Performance in higher dimensions

It is interesting to compare our coherent SRS approach with LSH on data sets in spaces with higher dimensions, where LSH-based approaches are supposed to have better performance. In this test, the LSH-based kNN [44] is employed and the number of neighbours is set to be $k = 20$. To make a fair comparison, for a query set with $|\mathcal{Q}|$ points, we adjust the search radius r to make the number of querying results meet the same value as that of the LSH-based method (i.e., $k|\mathcal{Q}|$). The comparison results in different dimensions are shown in Fig.11. Note that, when dimension increases tremendously, the memory consumption becomes a bottleneck for our approach. Thus, a set with 20k points is employed here. Our method is fast in low dimensional queries but becomes slower when dimension goes up to 12. One major problem of our approach in high dimensional cases is that the radius needs to be very large to meet the criterion of getting $k|\mathcal{Q}|$ resultant points. This results in a lot of candidates in the 2nd phase of our algorithm, which significantly slows down the computation.

Hierarchical updating time

We now study the performance of our tree updating scheme. Our approach is compared with the GPU-based kd-tree construction [17], the voxel-based GPU approach and the CPU-based ANN. In the kd-tree based approaches, the structure of the kd-tree needs to be rebuilt for dynamic points after each iteration. The voxel-based GPU approach rebuilds the voxel-based structure and reconstructs the hash table by using the highly parallel *scan*, *sort*, and *compact* primitives [23]. According to the statistics shown in Table 2, our updating method is faster

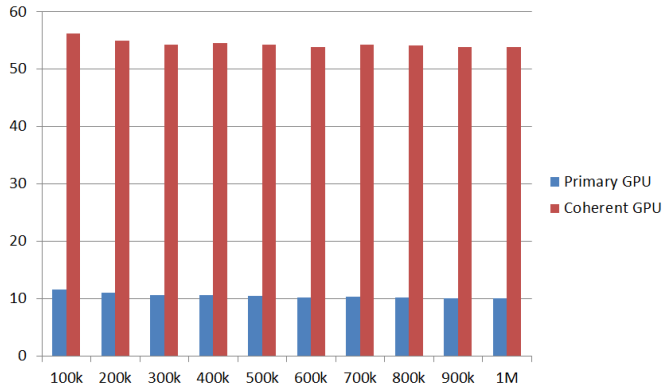


Figure 12: Comparison of WEE between the primary GPU SRS and the coherent GPU SRS on the sets of different number of points – the same number (horizontal axis) of data points and query points are conducted in these tests. Percentages of WEEs are shown along vertical axis.

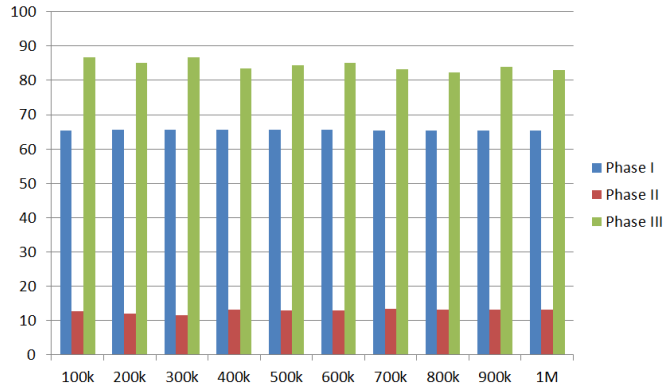


Figure 13: The analysis of WEE in different phases of our coherent SRS algorithm: (Phase I) packing of queries, (Phase II) common back traversal with swept volume and (Phase III) result extraction.

than all other approaches based on rebuilding.

As mentioned in Section 4.3, we need to rebuild the AABB-tree when the overlap between AABBs becomes serious. It is also interesting to know the speed of tree-construction in our CPU/GPU hybrid approach. For the same setup as above, the times of construction (by only using a single core on CPU) plus CPU/GPU communication are about $1.6-2.8\times$ of the querying time of coherent SRS. Therefore, the additional time cost of rebuilding an AABB-tree is not very significant if it is not applied very frequently. This can also be verified from the particle simulation example shown below.

Data Size	AABB-tree GPU refitting	kd-tree construction		voxel-base rebuild [20]
		GPU† [17]	CPU [11]	
100k	0.000624	0.0320	0.0406	0.0185
200k	0.00116	0.0470	0.106	0.0206
300k	0.00178	0.0630	0.184	0.0243
400k	0.00234	0.0780	0.298	0.0267
500k	0.00292	-	0.413	0.0319
600k	0.00356	-	0.530	0.0358
700k	0.00418	-	0.687	0.0396
800k	0.00471	-	0.827	0.0414
900k	0.00530	-	0.967	0.0453
1M	0.00589	-	1.17	0.0488
2M	0.0132	-	2.93	0.0804

† Note that, the GPU-based kd-tree construction algorithm [17] has very high cost in memory consumption so that it cannot compute data sets with more than 400k samples.

Table 2: Comparison of the updating time vs. the construction times in prior approaches (in sec.)

Workload balancing

As one of the key issues to leverage the parallelism on GPUs, our coherent GPU based SRS shows good balance on the distribution of workload running on different cores. Comparisons have been conducted between the primary GPU SRS (Algorithm 1) and the coherent GPU SRS (Algorithm 2) with the help of the NVIDIA Visual Profiler. Average of the *Warp Execution Efficiency* (WEE) of all the kernels in the algorithms is recorded to represent the kernel performance of the algorithms. WEE is the average percentage of active threads in each executed warp. When the parallelism on many-cores is fully utilized, the value of WEE should be 100%.

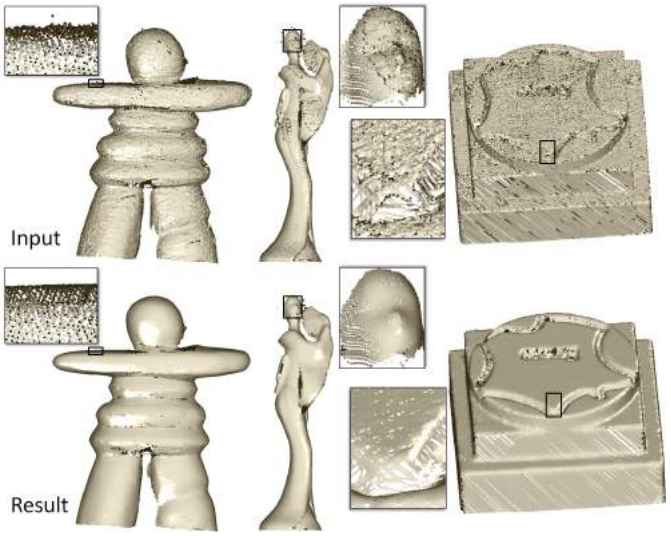
In our experimental tests for evaluating WEE, both the data points in $\mathcal{P}$ and the query points in $\mathcal{Q}$ are randomly drawn in $\Re^3$ in the range $[-1.0, 1.0]$. Search radius is assigned as $1/100$ of the bounding box's diagonal length. Data sets with different number of points ranging from $100k$ to $1M$ are tested, and statistics are shown in Fig.12. It is easy to observe that the coherent algorithm provides much better WEE comparing to the primary scheme in all tests. The improvement is mainly caused by aligning similar traversals near to each other (by the sorting in Step 7 of Algorithm 2). As has been proved in [51], assigning similar tree-traverses to the same batch of thread-execution can reduce the complexity of parallel algorithms. The performance improvement in terms of WEE on our coherent SRS algorithm is also caused by this same reason. Moreover, to further identify the bottleneck of our algorithm, WEE analysis for different phases of our algorithm is taken and shown in Fig.13. It is found that the back traversal phase is still a bottleneck. It may be further improved by using the sophisticated workload balancing algorithm (e.g., Algorithm 4 in [51]).

5.2. Applications

We have also tested the performance of our approach in different applications, including point-set geometry processing, distance-field evaluation and particle-based simulation.

Processing of Point-Sampled Geometry

Point-based geometry processing has been studied for more than a decade, where *Moving Least Square* (MLS) surface [52] is a widely used representation. Point projection is an operator that is intensively used. When iteratively computing the new position $\mathbf{q}'$ of a projected point $\mathbf{q}$, all the data points within the spherical range h are searched out to jointly determine $\mathbf{q}'$. Here h is a fixed parameter reflecting the anticipated spacing between neighbouring points. We test the performance of our

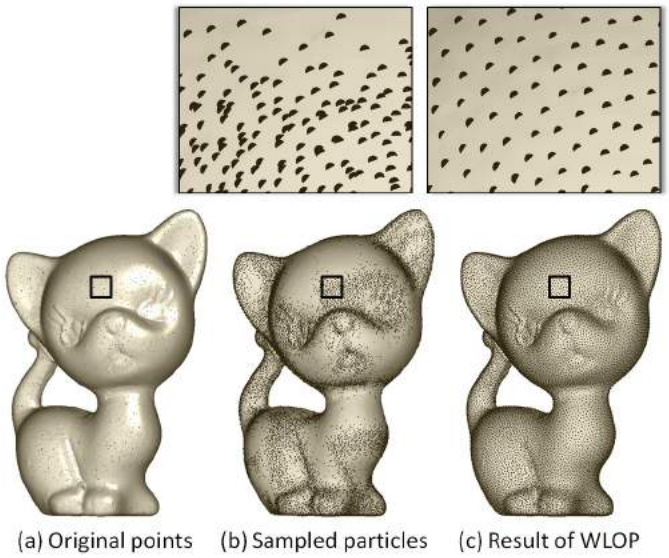


Model	Methods of SRS	Average SRS Time (sec.)	Total Time (sec.)
Inukshuk (206k pts.)	Coherent	0.162	6.33
	Shifted-sort	0.292 (×1.80)	10.4
	Voxel-based	0.663 (×4.09)	22.3
	ANN	5.36 (×33.1)	208
Japanese-Lady (176k pts.)	Coherent	0.0794	3.67
	Shifted-sort	0.129 (×1.62)	6.22
	Voxel-based	0.245 (×3.09)	12.1
	ANN	4.49 (×56.5)	166
Nasa (889k pts.)	Coherent	0.281	9.98
	Shifted-sort	0.557 (×1.98)	20.0
	Voxel-based	0.636 (×2.26)	22.1
	ANN	14.9 (×53.0)	689

Figure 14: MLS based smoothing is applied to three models with different support size h: (left) Inukshuk ($h = 70(\frac{\bar{L}}{n})^{1/2}$), (middle) Japanese-Lady ($h = 20(\frac{\bar{L}}{n})^{1/2}$) and (right) Nasa ($h = 20(\frac{\bar{L}}{n})^{1/2}$), where $\bar{L}$ is the bounding box's diagonal length and n is the number of sample points. The total running time of 30 iterations and the average SRS time in every iteration are also reported in the table.

coherent SRS in this application and compare with the ANN-based (on CPU), the shifted-sort based ANN (on GPU) and the voxel-based SRS (on GPU with resolution 256^3). Note that $\mathcal{P}$ and $\mathcal{Q}$ are the same in this application. MLS projection based smoothing is applied to three models shown in Fig.14 by *Point-Set Surface* (PSS) [53]. In all examples, 30 iterations of projections are applied for smoothing. Comparison of SRS times and total running times by using different methods is also given in Fig.14. Here, the total running time includes the memory allocation, copying and memory release on both the CPU and the GPU sides. Our coherent SRS is up to 4× faster than the voxel-based GPU search, 1.62 – 1.98× faster than the shifted-sort, and more than 50× faster than the CPU-based ANN.

Another example of SRS for dynamic points in geometry processing is surface approximation by using the *Weighted Locally Optimal Projection* (WLOP) [54]. Given the set of data points $\mathcal{P} = \{\mathbf{p}_j \in \Re^3\}$, the operator of WLOP projects a set of particles $\mathcal{X} = \{\mathbf{x}_i \in \Re^3\}$ onto the surface approximating $\mathcal{P}$ by



Methods of SRS	Average SRS Time (sec.)		Total (sec.)
	$\mathcal{X} \Rightarrow \mathcal{P}$	$\mathcal{X} \Rightarrow \mathcal{X}$	
Coherent	0.0840	0.0268	13.7
Shift-sort	0.133 (×1.58)	0.0432 (×1.61)	19.9
Voxel-based†	0.183 (×2.18)	0.0820 (×3.06)	29.7
ANN	2.30 (×27.4)	0.645 (×24.1)	462

†With the resolution: 128^3

Figure 15: Applying WLOP operators on a Kitten model with 137k points – the positions of 20k particles are progressively updated in 100 iterations. The total running time and the average SRS time are reported.

iteratively updating the positions of $\mathbf{x}_i$. When applying WLOP, the SRS needs to be intensively used among the points in $\mathcal{X}$ and between $\mathcal{P}$ and $\mathcal{X}$ by using all particles in $\mathcal{X}$ as the query points. An example is shown in Fig.15. Again, the computational time of our coherent SRS approach is compared with the state-of-the-art method. The radius of SRS is set to $r = 40(\frac{\bar{L}}{m})^{1/2}$ with m being the number of particles to be projected. 100 iterations are taken to generate optimal distribution of particles, and the average SRS times are reported. Again, the total running time includes the memory allocation, copying and memory release on both the CPU and the GPU sides.

Model	$\lvert\mathcal{P}\rvert$	r	Methods	Time (sec.)
Vase-Lion (Fig.16(a))	731k	$4w$	Coherent	0.880
			Voxel-based	1.84 (×2.09)
			ANN	210 (×239)
Buddha (Fig.16(b))	678k	$4w$	Coherent	1.05
			Voxel-based	1.64 (×1.56)
			ANN	231 (×220)
Offsetting Vase-Lion (Fig.17)	731k	$0.2\bar{L} + 2w$	Coherent	1.99
			Voxel-based	Failed
			ANN	236 (×119)

Table 3: Statistics of distance-field evaluation (res.: 257^3)

Distance-Field Evaluation

Our coherent SRS approach is also used to evaluate distance-

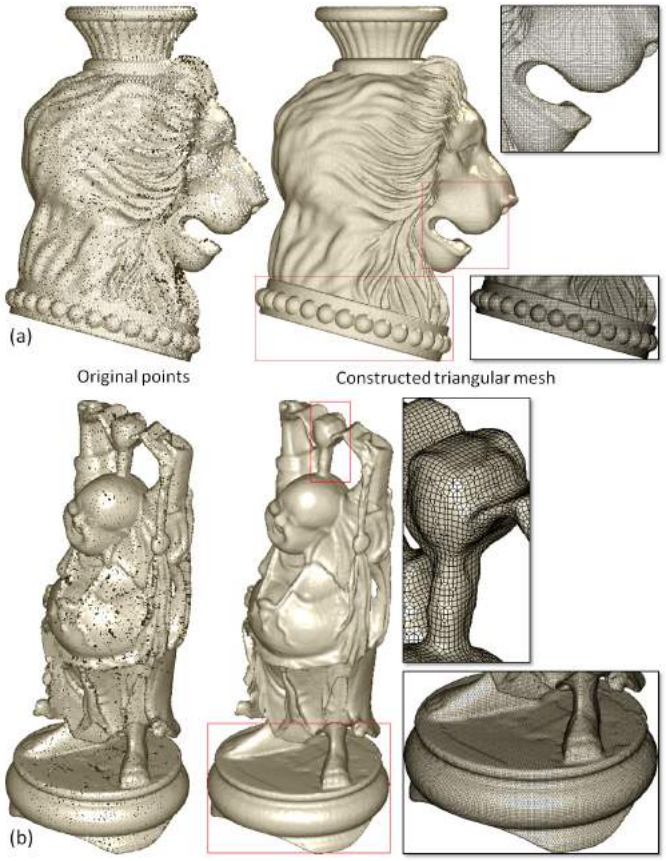


Figure 16: Distance-fields (res.: 257^3) are evaluated for (a) the Vase-lion model (with 731k points) and (b) the Buddha model (with 678k points), and the mesh surface can be extracted from the distance-fields at zero-level.

fields, which have many applications in geometric modelling. In our tests, a signed distance is computed from a set of points $\mathcal{P} = \{\mathbf{p}_i\}$ equipped with consistently oriented normal $\{\mathbf{n}_i\}$. The signed distance of a solid can be evaluated by a set of points, $\mathcal{P}$, sampled from its boundary

$$d_S(\mathbf{q}) = sgn((\mathbf{q} - \mathbf{x}_c) \cdot \mathbf{n}_c) \inf_{\mathbf{x}\in\mathcal{P}} \|\mathbf{x} - \mathbf{q}\|,$$

where $\mathbf{x}_c$ is the closest point of $\mathbf{q}$ in $\mathcal{P}$ and $\mathbf{n}_c$ is the normal of $\mathbf{x}_c$.

In Fig.16, we evaluate the signed distance-fields of two models on a 3D regular grid at a resolution of 256^3. The grid width is denoted by w. Specifically, every grid node is used as a query point to conduct SRS with radius $r = 4w$. There are $257^3 = 16,974,593$ query points in total. For those grid nodes having no data point found in their search range, the field values are assigned to *undefined*. In this way, a narrow-band distance field can be constructed. After obtaining a distance-field, the dual contouring algorithm [55] can be employed to extract the iso-surfaces: $d_S(\mathbf{q}) \equiv 0$ (see the results shown in Fig.16).

Figure 17 shows an example by using the signed distance-field to compute the offset surface of a given model. The grown and shrunk offset surfaces with distance as $0.2\bar{L}$ are generated, where $\bar{L}$ is the diagonal length of the model's bounding box. To obtain a narrow-band distance field to extract the offset surfaces, we conduct the coherent SRS with $r = 0.2\bar{L} + 2w$ to evaluate the values on 257^3 grid nodes. Computational statistics are shown in Table 3.

Figure 17: Distance-fields (res.: 257^3) are evaluated for computing the offset surfaces: (left) the shrunk offset with distance $0.2\bar{L}$, (middle) the original model and (right) the grown offset with distance $0.2\bar{L}$.

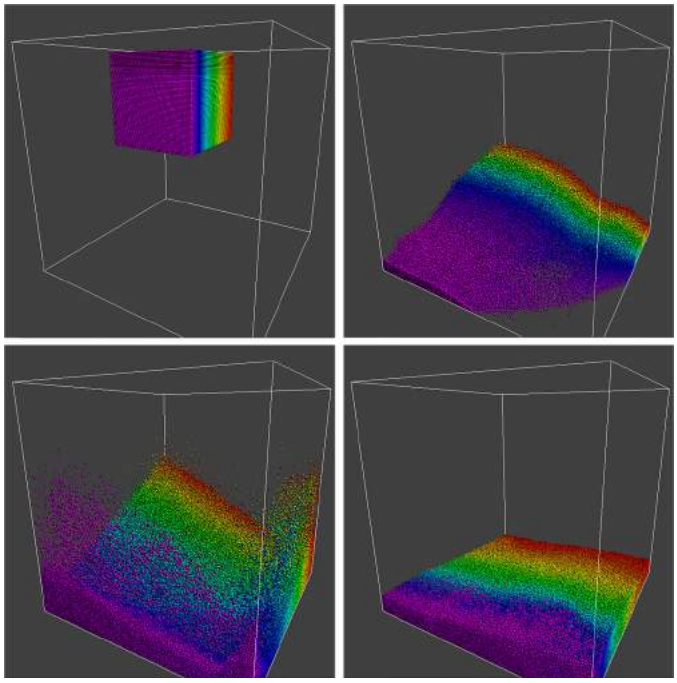
Particle-Based Simulation

Lastly, our coherent SRS is used to speed up particle-based physical simulation. Systems with a massive number of particles are a commonly used technique to simulate different physical behaviours. Due to the capability of highly parallel computing on modern GPUs, simulating particle systems at interactive rates becomes possible. In [20], voxel-based SRS is employed on GPUs to generate the repulsion forces between particles to simulate the splashing of fluids (see Fig.18). Here, we replace the voxel-based SRS by our coherent SRS and can observe up to 3.78× increase in the update rate in the simulator (see Fig.18). Moreover, when more particles are involved, higher speed increases can be observed. We also compare the performance of shifted-sorting based ANN [28] in this application. To conduct a fair comparison, the number of neighbours in this test is chosen as the maximal one obtained in the coherent SRS. From the statistic shown in Fig.18, we find that voxel-based SRS outperforms shifted-sorting in the scenario with more particles, but both are much slower than ours.

A more interesting study is about the distribution of computation time on each step during the whole simulation with $250k$ particles (see Fig.19). According to the criterion of rebuilding in Section 4.4, the AABB-tree for SRS is rarely rebuilt – only 33 times in 2,000 iterations. The total time of our coherent SRS in each step is also compared with other GPU approaches. It is found that the computation of our approach is much faster even in the step with tree rebuilding. In summary, our coherence SRS is also more efficient than the prior approaches in the scenario of promptly moved points.

6. Conclusion

This paper presents a highly parallel algorithm of spherical range-search for dynamic points that exploits the computational power of many-core GPUs. The spatial coherence of query points and the temporal coherence of dynamic points are exploited to achieve very efficient range searching in our approach. In the coherent SRS, query points are first packed.



Particle Number	Average Steps per Second		
	Coherent	Shifted-sort	Voxel-based
50k	60.98	15.41 (×3.96)	21.75 (×2.80)
100k	25.41	8.421 (×3.02)	10.63 (×2.39)
150k	20.91	4.155 (×5.03)	6.656 (×3.14)
200k	16.55	3.121 (×5.30)	4.762 (×3.48)
250k	13.84	2.473 (×5.60)	3.662 (×3.78)

Figure 18: Application of particle simulation: 2.39 – 5.60× increase in speed can be observed on the simulator of a particle system by replacing the voxel-based SRS with our coherent SRS. The statistic shows the average number of steps which can be computed per second during the whole simulation with 2,000 steps.

Then, the spatial coherence of these query points is employed to reduce more than 90% of the back-traverse on the tree. After getting the intermediate candidates of SRS, the final results are extracted by a parallel compaction. The technique has been tested in several applications with both slowly and promptly moved points to verify its efficiency. Compared with existing techniques, we can achieve up to two-order performance improvement over the sequential algorithm [11] and around 4× to 5× faster than the voxel-based [20] and shift-sorting based [28] approaches running on GPUs.

7. Discussion

In an extension of this work, we find that a much higher ratio of speedup can be achieved in the benchmark of particle-based simulation with highly uniform distribution and variation of searching radii (see Fig.20 for the benchmark). In this benchmark of particle-based simulation, 30k particles are clustered into three groups:

1. 10k red and yellow particles in the upper-left region (with searching radius $3r$;
2. 10k green particles in the upper-right part having a smaller searching radius $2r$;
3. the bottom 10k blue and purple particles using the search radius r.

The distribution of particles in the upper region are much sparser than the lower region at the beginning of simulation. While the value of r changes from $0.25/100\bar{L}$ to $0.5/100\bar{L}$ and then to $1/100\bar{L}$ with $\bar{L}$ being the diagonal length of the simulation envelope, the average *frames per second* (fps.) in the simulation are recorded as around 66fps., 62fps. and 62fps. In short, the increase of time-cost is trivial. However, when the same searching radii are used in the voxel-based search on GPU, computations at the speed of 21fps., 4fps. and 0.4fps. are observed. That mains the speedup of 3.1×, 15.5× and 155× accordingly. A similar observation is reported in the prior work of Liu's [40] on collision detection – when the searching ranges have large variation, a much higher ratio of speedup can be achieved comparing to the voxel-based searching approach.

A hybrid approach is conducted in our implementation for the tree-construction (on CPU) and the query-and-update (on GPU). Although as analyzed very few times of construction are needed in most scenarios, it is still worth to study whether there is enough benefit for a full GPU-based approach like [38, 39]. A common problem of existing GPU-based tree construction approaches is that the constructed tree is not well-balanced in many cases. On the other aspect, we also plan to extend Liu's approach [40] for SRS and study its efficiency, where their method is fully GPU-based and does not need a tree-construction step.

The current implementation of Phase II in our coherent SRS algorithm is a bottleneck of the overall algorithm. One possible work is to add a sophisticated dynamic workload balancing strategy as what is conducted in [51]. Moreover, we will also further optimize our code to achieve better memory usage. We plan to make the source code of our method publicly available.

Acknowledgement

The authors would like to acknowledge the valuable comments made by Young J. Kim in private discussion. This work is supported by the HKSAR RGC/GRF Grant (CUHK/14207414) and also partially by the Hong Kong Innovative Technology Fund (ITS/247/11).

[1] H. Huang, D. Li, H. Zhang, U. Ascher, D. Cohen-Or, Consolidation of unorganized point clouds for surface reconstruction, ACM Transactions on Graphics 28 (5) (2009) 176:1–176:7.
[2] Y. Lipman, D. Cohen-Or, D. Levin, H. Tal-Ezer, Parameterization-free projection for geometry reconstruction, ACM Trans. Graph. 26 (3).
[3] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C. T. Silva, Point set surfaces, in: Proceedings of the IEEE Conference on Visualization '01, 2001, pp. 21–28.
[4] M. W. Jones, J. A. Bærentzen, M. Sramek, 3d distance fields: A survey of techniques and applications, IEEE Transactions on Visualization and Computer Graphics 12 (4) (2006) 581–599.
[5] M. Macklin, M. Müller, Position based fluids, ACM Trans. Graph. 32 (4) (2013) 104:1–104:12.

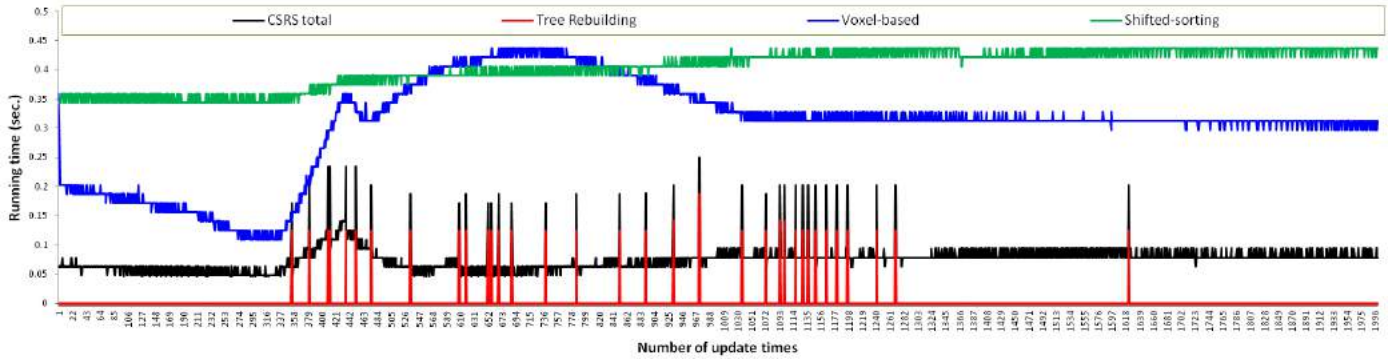


Figure 19: The distribution of computing time in each iteration during 2,000 iterations of the whole simulation. The curve in red shows the time for AABB-tree rebuilding when it is necessary. The total time of our coherent SRS together with the tree rebuilding is still less than the voxel-based SRS and the shifted-sorting based ANN.

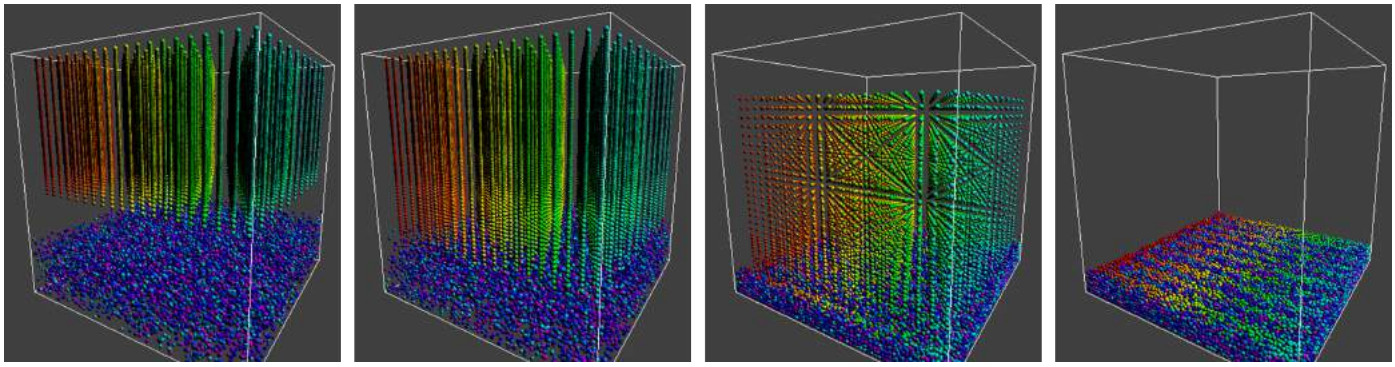


Figure 20: In a benchmark having non-uniform distribution of particles with the variation of searching radii, much higher speedup can be observed by using our CSRS approach vs. the voxel-based GPU searching. The figures show the progressive results of simulation. Within the 30k particles, 10k (see the red and yellow ones in the upper-left region of the leftmost figure) and 10k (the green ones in the upper-right region) of them have the searching radii as 3× and 2× of the bottom 10k particles (blue and purple) respectively. When $r = 3/100\bar{L}$, $r = 2/100\bar{L}$ and $r = 1/100\bar{L}$ are used for the searching radii of these three groups of particles, the average update rate of our CSRS approach in this simulation benchmark is over 150× of the voxel-based approach running on GPU.

[6] M. Becker, M. Teschner, Weakly compressible SPH for free surface flows, in: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2007, pp. 209–217.

[7] Y. Zhu, R. Bridson, Animating sand as a fluid, ACM Trans. Graph. 24 (3) (2005) 965–972.

[8] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan, Photon mapping on programmable graphics hardware, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, 2003, pp. 41–50.

[9] T. Davidovič, J. Křivánek, M. Hašan, P. Slusallek, Progressive light transport simulation on the gpu: Survey and improvements, ACM Trans. Graph. 33 (3) (2014) 29:1–29:19.

[10] V. Garcia, E. Debreuve, M. Barlaud, Fast k nearest neighbor search using GPU, in: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008, pp. 1–6.

[11] D. M. Mount, S. Arya, ANN: A library for approximate nearest neighbor searching, `http://www.cs.umd.edu/~mount/ANN/` (2010).

[12] M. Muja, D. G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration, in: International Conference on Computer Vision Theory and Application (VISSAPP'09), 2009, pp. 331–340.

[13] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] S. Popov, J. Günther, H.-P. Seidel, P. Slusallek, Experiences with streaming construction of SAH KD-trees, in: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 89–94.

[15] M. Shevtsov, A. Soupikov, E. Kapustin, Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes, Computer Graphics Forum 26 (3) (2007) 395–404.

[16] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, J. C. Hart, Parallel SAH k-D tree construction, in: Proceedings of the Conference on High Performance Graphics, 2010, pp. 77–86.

[17] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time KD-tree construction on graphics hardware, ACM Trans. Graph. 27 (5) (2008) 126:1–126:11.

[18] Z. Wu, F. Zhao, X. Liu, SAH KD-tree construction on GPU, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, 2011, pp. 71–78.

[19] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, D. Manocha, Memory-scalable GPU spatial hierarchy construction, IEEE Transactions on Visualization and Computer Graphics 17 (4) (2011) 466–474.

[20] S. Green, Particle simulation using CUDA, Online Document (2012).

[21] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009, pp. 1–10.

[22] X. Zhang, Y. J. Kim, Interactive collision detection for deformable models using streaming AABBs, IEEE Transactions on Visualization and Computer Graphics 13 (2) (2007) 318–329.

[23] M. Harris, S. Sengupta, J. Owens, Parallel prefix sum (scan) with cuda, GPU Gems 3 (39) (2007) 851–876.

[24] D. L. James, D. K. Pai, Bd-tree: Output-sensitive collision detection for reduced deformable models, ACM Trans. Graph. 23 (3) (2004) 393–398.

[25] C. Hoffmann, V. Shapiro, V. Srinivasan, Geometric interoperability via queries, Computer-Aided Design 46 (2014) 148–159.

[26] J. Qi, V. Shapiro, e-topological formulation of tolerant solid modeling, Computer-Aided Design 38 (4) (2006) 367–377.

[27] Meshfree simulation of deforming domains, Computer-Aided Design 31 (7) (1999) 459–471.

[28] S. Li, L. Simons, J. B. Pakaravoor, F. Abbasinejad, J. D. Owens, N. Amenta, kANN on the GPU with shifted sorting, in: Proceedings

of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics, 2012, pp. 39–47.

[29] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, 2nd Edition, Springer-Verlag, 2000.

[30] P. Goswami, P. Schlegel, B. Solenthaler, R. Pajarola, Interactive sph simulation and rendering on the GPU, in: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2010, pp. 55–64.

[31] D. Kim, M.-B. Son, Y. J. Kim, J.-M. Hong, S.-E. Yoon, Out-of-core proximity computation for particle-based fluid simulations., in: Proceedings of High Performance Graphics 2014, 2014.

[32] D. Qiu, S. May, A. Nüchter, GPU-accelerated nearest neighbor search for 3d registration, in: Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems, Springer-Verlag, 2009, pp. 194–203.

[33] S. Arya, D. M. Mount, Algorithms for fast vector quantization, in: Proc. of DCC '93: Data Compression Conference, IEEE Press, 1993, pp. 381–390.

[34] I. Wald, On fast construction of sah-based bounding volume hierarchies, in: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, 2007, pp. 33–40.

[35] C. Lauterbach, M. Garland, S. Sengupta, D. P. Luebke, D. Manocha, Fast BVH construction on GPUs, Comput. Graph. Forum 28 (2) (2009) 375–384.

[36] M. Stich, H. Friedrich, A. Dietrich, Spatial splits in bounding volume hierarchies, in: Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 7–13.

[37] A. García, S. Murguia, U. Olivares, F. F. Ramos, Fast parallel construction of stack-less complete lbvh trees with efficient bit-trail traversal for ray tracing, in: Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry, 2014, pp. 151–158.

[38] C. Lauterbach, Q. Mo, D. Manocha, gProximity: Hierarchical GPU-based operations for collision and distance queries, Comput. Graph. Forum 29 (2) (2010) 419–428.

[39] F. Liu, Y. J. Kim, Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus, IEEE Transactions on Visualization and Computer Graphics 20 (5) (2014) 714–725.

[40] F. Liu, T. Harada, Y. Lee, Y. J. Kim, Real-time collision culling of a million bodies on graphics processing units, ACM Trans. Graph. 29 (6) (2010) 154:1–154:8.

[41] E. Larsen, S. Gottschalk, M. C. Lin, D. Manocha, Fast proximity queries with swept sphere volumes, in: of International Conference on Robotics and Automation, 2000, pp. 3719–3726.

[42] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, K. Zikan, Efficient collision detection using bounding volume hierarchies of k-dops, IEEE Transactions on Visualization and Computer Graphics 4 (1) (1998) 21–36.

[43] S. Liu, C. C. L. Wang, K.-C. Hui, X. Jin, H. Zhao, Ellipsoid-tree construction for solid objects, in: Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling, 2007, pp. 303–308.

[44] J. Pan, D. Manocha, Fast GPU-based locality sensitive hashing for k-nearest neighbor computation, in: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2011, pp. 211–220.

[45] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, P. Dubey, Streaming similarity search over one billion tweets using parallel locality-sensitive hashing, Proceedings of the VLDB Endowment 6 (14) (2013) 1930–1941.

[46] J. Gunther, S. Popov, H.-P. Seidel, P. Slusallek, Realtime ray tracing on gpu with bvh-based packet traversal, in: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, 2007, pp. 113–118.

[47] I. Wald, S. Boulos, P. Shirley, Ray tracing deformable scenes using dynamic bounding volume hierarchies, ACM Trans. Graph. 26 (1).

[48] T. Aila, S. Laine, Understanding the efficiency of ray traversal on gpus, in: Proceedings of the Conference on High Performance Graphics 2009, 2009, pp. 145–149.

[49] S. Guntury, P. J. Narayanan, Ray tracing dynamic scenes with shadows on gpu, in: Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization, 2010, pp. 27–34.

[50] C. A. R. Hoare, Algorithm 65: Find, Commun. ACM 4 (7) (1961) 321–322.

[51] J. Pan, D. Manocha, Gpu-based parallel collision detection for fast motion planning, International Journal of Robotics Research 31 (2) (2012) 187–200.

[52] D. Levin, Mesh-independent surface interpolation, in: H. Brunnett, Mueller (Eds.), Geometric Modeling for Scientific Visualization, Springer-Verlag, 2003, pp. 37–49.

[53] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, C. T. Silva, Point set surfaces, in: Proceedings of the IEEE Conference on Visualization '01, 2001, pp. 21–28.

[54] H. Huang, D. Li, H. Zhang, U. Ascher, D. Cohen-Or, Consolidation of unorganized point clouds for surface reconstruction, ACM Trans. Graph. 28 (5) (2009) 176:1–176:7.

[55] T. Ju, F. Losasso, S. Schaefer, J. Warren, Dual contouring of hermite data, ACM Trans. Graph. 21 (3) (2002) 339–346.