ArborX: A Performance Portable Geometric Search Library

D. LEBRUN-GRANDIÉ, A. PROKOPENKO, B. TURCKSIN, and S. R. SLATTERY, Oak Ridge National Laboratory

Searching for geometric objects that are close in space is a fundamental component of many applications. The performance of search algorithms comes to the forefront as the size of a problem increases both in terms of total object count as well as in the total number of search queries performed. Scientific applications requiring modern leadership-class supercomputers also pose an additional requirement of performance portability, i.e., being able to efficiently utilize a variety of hardware architectures. In this article, we introduce a new open-source C++ search library, ArborX, which we have designed for modern supercomputing architectures. We examine scalable search algorithms with a focus on performance, including a highly efficient parallel bounding volume hierarchy implementation, and propose a flexible interface making it easy to integrate with existing applications. We demonstrate the performance portability of ArborX on multi-core CPUs and GPUs and compare it to the state-of-the-art libraries such as Boost.Geometry.Index and nanoflann.

CCS Concepts: • **Mathematics of computing → Mathematical software**;

Additional Key Words and Phrases: Bounding volume hierarchy, performance portable algorithm, geometric search

ACM Reference format:
D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery. 2020. ArborX: A Performance Portable Geometric Search Library. *ACM Trans. Math. Softw.* 47, 1, Article 2 (December 2020), 15 pages.
https://doi.org/10.1145/3412558

Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy.
This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
This research used resources of the Compute and Data Environment for Science (CADES) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
Authors' addresses: D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery, Oak Ridge National Laboratory, PO Box 2008, Bldg 5700, MS 6085, Oak Ridge, TN 37831; emails: {lebrungrandt, prokopenkoav, turcksinbr, slatterysr}@ornl.gov.

0098-3500/2020/12-ART2 $15.00
https://doi.org/10.1145/3412558

1 INTRODUCTION

Performing proximity searches on collections of geometric objects is an inherent component of applications in many fields. Finding the nearest neighbors of a point, or finding all objects within a certain distance, are common tasks in shape registration methods [Besl and McKay 1992] in computer vision and pattern recognition, special effects in games and movies [Karras 2012b], anomaly detection [Breunig et al. 2000], machine learning [Pedregosa et al. 2011], cosmology [Sewell et al. 2015], data transfer in multiphysics simulations [Slattery 2016], contact detection in computational mechanics [Feng and Owen 2002], and many others. Such algorithms involve multiple searches through thousands or millions of objects. The performance of search algorithms is thus crucial for the overall performance of an application. Brute force computations are prohibitively expensive for all but the simplest applications with very few objects of interest. Instead, methods employing tree-based data structures are preferred due to their inherent logarithmic cost.

Many libraries dedicated to geometric search algorithms have been developed. A major choice in developing such a library is the underlying tree data structure. This choice dictates both the complexity of implementation and the resulting performance, including the tradeoff between the time to construct the data structure and the time to perform search queries. A data structure that is fast in performing the search may require a longer time setting up, and vice versa. The way it is used in each application dictates the desired tradeoff. Two tree structures, k-d and R-tree, are particularly suitable for geometry-based search, and are commonly implemented in libraries.

k-d tree [Bentley 1975] is a binary space partitioning data structure. The construction algorithm chooses a suitable hyperplane to split a given set of points into two, and continues recursively for each subset, deciding on a new hyperplane each time. The internal nodes of the tree correspond to such hyperplanes, with the parent-child relationship formed through a single recursive iteration. The hyperplane orientations are typically switched at each level so as not to produce very skewed sets, with cyclic rotation amongst dimensions being the simplest approach. Once the algorithm terminates, the leaf nodes contain the original set of points. Variants of the k-d tree are widely used in libraries, e.g., FLANN [Muja and Lowe 2009] and nanoflann [Blanco and Rai 2014].

The R-tree [Guttman 1984] is an alternative data structure used for spatial search. The leaf nodes of an R-tree are multidimensional rectangles bounding the objects of interest, and higher level nodes of a tree are aggregations of an increasing number of objects. This is the data structure that was chosen in the Boost.Geometry.Index [Gehrels et al. 2017] library.

Both nanoflann and Boost.Geometry.Index libraries are widely used in applications. Both, however, are less suitable for high-performance computing (HPC) applications as they do not take advantage of multi-threading, nor do they consider the variety of different architectures available today. Current HPC trends require search algorithms to perform well on a variety of hardware architectures, including GPUs and other accelerators provided by a variety of vendors. This is particularly true within the Exascale Computing Project of the U.S. Department of Energy [DOE 2016] where significant resources are devoted to porting applications to utilize both CPUs and GPUs, and preparing for upcoming new architectures such as APU and FPGA. Given the variety of the current hardware landscape and some uncertainty in future hardware directions, a search library that is developed from scratch should be performance portable.

With this goal in mind, we introduce a new open-source library ArborX.[1] It is a header-only C++ library with a focus on performance portability for both current and known future leadership-class supercomputers. The implemented algorithms were carefully chosen to be efficient on the multiple architectures, and rely on the C++ Kokkos [Edwards et al. 2014] library to provide performance

[1] ArborX is available at https://github.com/arborx/ArborX.

portability. We focus on low order dimensional space, building the data structures from scratch (i.e., no incremental updates).

The article is organized as follows. In Section 2, we describe the algorithms that are used in ArborX. In Section 3, we compare our library to other state-of-the-art libraries and demonstrate its performance on different architectures. Finally, we present our conclusions in Section 4.

2 SEARCH ALGORITHM

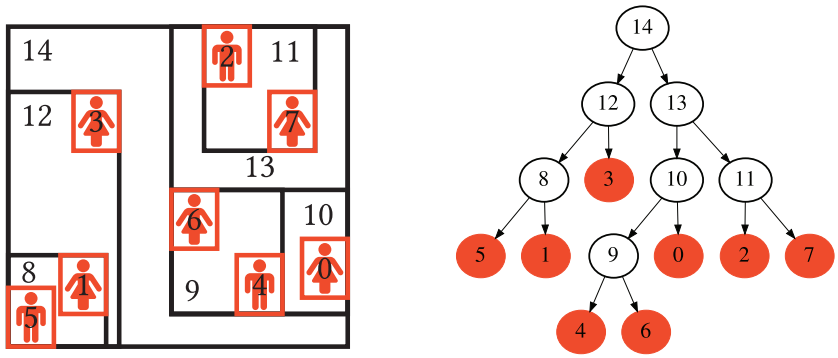
Search algorithms are memory bound by nature. The fundamental parts of any good search algorithm include visiting as few tree nodes of the search tree as possible, reducing the amount of memory required by each tree node, and using inexpensive computations to construct and query the tree data structure. Furthermore, reducing thread execution divergence (executing different code) and data divergence (reading or writing disparate locations in memory) is highly desirable in parallel implementations, particularly for accelerators with thousands of threads (such as GPUs) and architectures that improve performance via vectorization (e.g., modern CPUs). Below we present a bounding volume hierarchy (BVH) tree data structure that was carefully chosen to satisfy all of these requirements on modern architectures.

A BVH is a tree structure created from a set of geometric objects in a multi-dimensional space. The objects are wrapped in simple geometric form (*bounding volumes*) that form leaf nodes of the tree. Similarly to the R-tree, each node of a BVH is an aggregate of its children, enclosing the group within a larger bounding volume. The root node of the hierarchy corresponds to the bounding volume around all objects (called scene bounding volume). Binary BVH is by far the most popular choice and is what we have chosen for our implementation in this work. For multithreaded and GPU implementations the binary BVH has the convenient property that the number of internal nodes in the tree is equal to the number of leaf nodes decreased by one that allows for static memory allocations once the input geometry is known.

The choice of the geometry of a bounding volume is crucial for performance. Bounding volumes should require little data to store, be fast to test for intersection, have fast distance computations, and fit closely to the underlying object (to avoid unnecessary traversal). In practice, axis-aligned bounding boxes (AABB), which are boxes aligned with axes of the coordinate system, are often a good choice [Haverkort 2004]. They require minimal space to store (two opposite corner points or six floating point numbers in three dimensions (3D)) and are fast to test for intersections. Computing the distance from a point to an AABB is also inexpensive. This often outweighs the main drawback of AABB of not fitting tightly to the underlying data in some situations. Figure 1 demonstrates an example of a BVH tree formed for a set of eight geometric objects represented by human figures. The red bounding volumes correspond to leaf nodes (nodes 0–7) and tightly surround the objects themselves. They are then combined to form larger bounding volumes corresponding to internal nodes (nodes 8–14), culminating with the root node bounding volume surrounding the whole scene. The corresponding BVH tree is shown in Figure 1(b).

The idea of parallelization of BVH construction by using a space-filling curve (called linear BVH) was proposed in Lauterbach et al. [2009]. In that approach, the leaves of a tree are ordered based on a space-filling Z-curve using Morton codes. The algorithm was improved in Pantaleoni and Luebke [2010] and Garanzha et al. [2011] by allowing processing each level in parallel. In Karras [2012a], a new approach allowing all internal nodes to be constructed concurrently was introduced. In Apetrei [2014], the algorithm was further improved by merging hierarchy construction with bounding volume computations in a single bottom-up pass.

The question of the quality of the constructed hierarchy often arises in applications. For ray tracing applications, there is a rich literature devoted to the question (see, for example, Domingues and Pedrini [2015] and the references within). Typically, the goal is to minimize the surface area



(a) Geometric objects and bounding volumes (b) BVH tree

Fig. 1. Bounding volumes and corresponding BVH tree for a set of eight geometric objects represented by human figures.

heuristic (SAH), which is often achieved through rearranging subtrees. It is possible that many of the techniques could be applicable to scientific applications once a suitable heuristic is determined. Other approaches may warrant examination. In Vinkler et al. [2017], the authors propose extending Morton codes to incorporate additional geometric information, such as object size, leading to higher quality BVH. Uneven primitive distribution is addressed in Hu et al. [2019] through computation of a local density information to inform the BVH construction.

Another active area of research is reducing the amount of bandwidth used by BVH. Howard et al. [2019] replaces single precision floating points used to store a bounding volume with two 32-bit integers containing quantized bounds. Benthin et al. [2018] reduces the depth of the hierarchy through compressing leaf nodes into wide multi-nodes.

The focus of this work has so far been on the speed and portability of the library, rather than the quality of the constructed hierarchy. In scientific applications, it is typical that the tree is rebuilt multiple times (e.g., for each timestep of a time-dependent application), placing lower importance on the quality. We chose Karras [2012a] for our initial implementation due to its simplicity. Addressing the quality is deferred to the future work.

We provide an overview of the core algorithms for the construction of such tree in Section 2.1. The corresponding traversal algorithms are detailed in Section 2.2. Finally, in Section 2.3, we discuss a user interface to allow flexibility in interaction with user data.

2.1 BVH Construction

In this section, we describe the construction of the linear BVH used to accelerate the search for a given set of geometric objects. The degree of parallelism in BVH trees is severely limited in a typical bottom-up construction (i.e., constructing a node only after its children). On the other hand, the linearity imposed by a Z-curve implicitly partitions the objects based on the highest differing bit of their Morton codes, allowing for a top-down approach. A clever numbering of internal nodes as described in Karras [2012a] then allows for a fully parallel algorithm. We implement the original algorithm with only minor changes (such as removing parent pointers from tree nodes, and storing the leaf node permutation index in a leaf), with an intent to incorporate [Apetrei 2014] in the near future. A brief description of the involved steps is provided below.

Construct AABBs. As in any BVH algorithm, the first step is to compute the bounding boxes of the user provided objects. The only requirement on the objects is that they are *boundable*. For

certain classes of objects, such as polyhedrons, this step is inexpensive. The computed boxes may be degenerate, such as those produced for points or objects of a dimension lower than that of AABBs, giving one or more dimensions with an extent of zero.

Calculate the scene bounding box. The scene bounding box is an AABB that contains the bounding boxes of all objects. It is easily computed by a reduction of the corners of the bounding boxes.

Assign Morton codes for each AABB. Morton codes, or Z-order codes, are used to map multidimensional data to a single dimension, while preserving the spatial locality of the data. Given a point, a Morton code can be efficiently computed by interleaving bits of the point coordinates. The Morton code of a bounding box is computed as the Morton code of its centroid scaled using the scene bounding box. This guarantees that all coordinates lie within $[0, 1]^3$ cube. In general, Morton codes are not guaranteed to be unique. Thus, if multiple objects share the same Morton code, they are augmented with an index to differentiate them.

Sort the bounding boxes using their Morton code. The bounding boxes may now serve as leaf nodes. The goal of this procedure is to decrease the size and the overlap of bounding boxes of internal nodes that will be generated.

Generate the bounding volume hierarchy. With a linear order imposed by sorted Morton codes, construction of a hierarchy can be seen as a recursive partitioning of the range of Morton indices so that each internal node in a tree corresponds to an interval of Morton codes. The recursion terminates when a range contains only one item, which is to be a leaf node. The described partitioning is based on selecting a position called *split* to cut a given range in two. The splits are based on the highest differing bits of Morton codes within a given range. The range of each internal node is computed independently, allowing a parent node to determine its children and record the parent-child relationships without waiting for the construction of other nodes. A more detailed discussion can be found in Karras [2012a]. The permutation indices computed in the previous step are stored in the leaf nodes.

Calculate internal nodes bounding boxes. The final step is to compute the bounding boxes of internal nodes by traversing the tree bottom-up. In parallel, each thread is assigned a leaf node and traverses towards the tree root. Upon encountering an internal node, only one of the children's threads is allowed to proceed further. As parent pointers are not used in hierarchy traversal, we avoid storing the pointer to the parent node inside a child node to minimize used memory. Instead, the parent pointers are kept in an auxiliary array that is dismissed after construction.

2.2 BVH Traversal

Once constructed, the tree data structure may be used as many times as needed to complete the search process. Each query of the data structure results in a traversal of the tree where the approximation of objects by AABBs allows for a preliminary coarse search that is responsible for listing all boxes with potential collisions. It is then followed by a fine search where a user-specified search criteria is used to trim the results. The tighter the bounding volumes are to the real objects, the more accurate the results of the coarse search are, and the fewer expensive fine search queries that are needed.

We distinguish two kinds of (search) queries: spatial and nearest. A spatial query searches for all objects within a certain distance of an object of interest. A nearest query, on the other hand, looks for a certain number of the closest objects regardless of their distance from an object of interest. These two query kinds require fundamentally different tree traversal algorithms. The spatial query has to necessarily explore all nodes in a tree that satisfy a given distance-based predicate. The

nearest query, however, can terminate early when it can be guaranteed that the already found candidates are the best possible ones.

It is very common to execute multiple search queries simultaneously. In parallel, the threads are executed in *batched* mode, with each thread assigned a range of search queries (on CPU) or a single search query (on GPU). While it may be possible to further improve the performance by having multiple threads work on the same query, we do not address this in the current ArborX implementation, and each query is performed exclusively by a single thread.

As threads traverse the tree, the attention to execution and data divergence is paramount for performance. We next describe our strategies for traversal with each query type.

2.2.1 Traversal for Spatial Queries. In spatial query traversal, each node can be tested independently for predicate satisfaction. A simple distance-based predicate tests for whether a distance from a bounding volume to a point is less than a given radius.

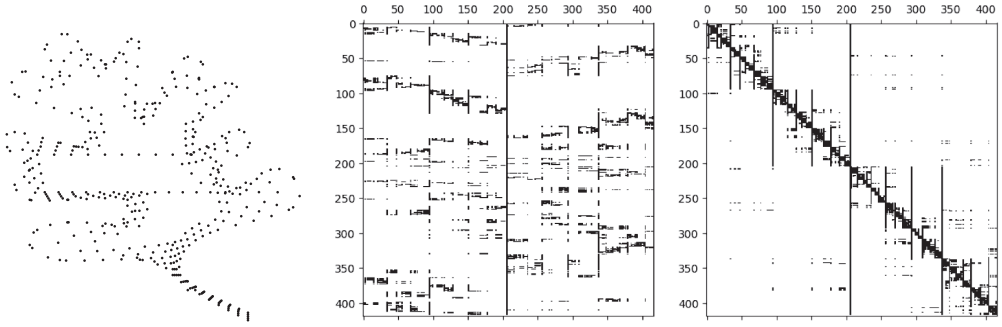
Spatial traversal is executed top-down, starting from the root node. A naive recursive implementation may lead to a high execution divergence as shown in Karras [2012c]. Instead, an iterative traversal is preferred, using a stack to keep track of nodes to visit. In the beginning of the traversal the root node is added to the stack. The algorithm proceeds by popping a node from the stack, and testing its children for predicate satisfaction, upon which they are either added to the stack (internal nodes) or to the output (leaf nodes). The algorithm terminates upon an empty stack.

An important issue associated with spatial traversals is that the number of found objects is not known *a priori*. This issue is typically not addressed in computer graphics applications as the results are processed on the fly in many cases. However, storing the results plays an important role in scientific applications, where further processing of the results is required (e.g., halo finding algorithm [Sewell et al. 2015] calculates clusters based on the computed data). It is well known that dynamic memory allocation is inefficient in multithreading, and is problematic on GPUs. This can be avoided by using a count-and-fill technique, i.e., by doing two passes (2P). The first pass just counts the number of found objects. Then, the required storage is allocated, and the process is repeated in the second pass, this time storing the results.

The 2P approach, while robust, comes with a drawback of having to traverse the hierarchy twice. A superior alternative, when possible, is to only do the second pass once the preallocated memory is exceeded. In this approach (called 1P), an estimate for a maximum number of found objects per query is provided by a user. During the first pass, the found objects are both counted and stored. If the storage is exceeded, then the algorithm falls back to the 2P approach. If the estimate is correct, i.e., is an upper bound, then only a single pass is done, improving the overall performance. That single pass is then followed by "compacting" the results due to excess allocation. Such a technique is typically less costly than performing the traversal twice.

2.2.2 Traversal for Nearest Queries. Nearest traversal proceeds in top-down fashion, similarly to spatial-based traversal. However, in this case, the number of found neighbors (or, rather, its upper bound) is known in advance, and thus allows for the preallocation of memory and to avoid the second pass through the tree.

A typical implementation of nearest traversal uses a priority queue based on distances, using the closest node in each iteration. An alternative and better performing approach, first derived for k-d trees in Patwary et al. [2016], is to use a stack. As stack is a Last-In-First-Out data structure, it is possible to get a behavior similar to the one of a priority queue by adding a child with a shorter distance second (so that it sits on top of the stack). The algorithm terminates when the remaining candidates in the stack are guaranteed to result in worse results, or the stack is empty. The final (optional) step is to clean the results by purging missing data (if, for example, the number of found objects is less than specified).



(a) Point cloud (b) Queries in original order (c) Queries in Morton-based sorted order

Fig. 2. Effect of query ordering on nearest traversal.

2.2.3 Query Ordering. Both execution and data divergence depend heavily on whether the nearby computational threads traverse the tree similarly. Sorting the queries may have a significant impact on the overall performance as was noted in Karras [2012c]. One way to accomplish this is by making sure that the search queries for nearby threads are "close" to each other, i.e., the corresponding tree traversals would be very similar in both nodes they visit and the order they visit them. This can be achieved by computing Morton codes for query objects and then using them for pre-sorting.

To illustrate this phenomenon, consider the nearest traversal of a point cloud of 418 points (representing a leaf) shown in Figure 2(a). Figure 2(b) represents a binary matrix of size 418×418 corresponding to the original ordering of search queries. Each row represents a search query assigned to a single thread, and each column corresponds to an internal node in the tree. A value in the matrix is nonzero (black) when a thread accesses a bounding box of a corresponding node. As one can see, using the original ordering of search queries results in little correlation of accessed nodes of two nearby threads. The performance suffers in this case due to poor memory access pattern. Figure 2(c), however, corresponds to the queries reordered based on their Morton codes. It is clear that the nearby threads now share many nodes of the tree in their traversal. The apparent hierarchical pattern of the matrix indicates the concentration of queries in certain subtrees before switching to sibling trees.

2.3 Library Interface

ArborX implements a performance portable interface through the use of Kokkos [Edwards et al. 2014], a C++ library providing a uniform programming interface for various backends, such as OpenMP or CUDA. Using Kokkos allows for running the same code on CPUs or GPUs by simply changing the backend through a template parameter.

The construction procedure begins with a set of bounding boxes, provided by a user as a `Kokkos::View`, a Kokkos data structure corresponding to a multi-dimensional array. At a high level, `Kokkos::View<T*, DeviceType>` can be thought of as an array containing objects of type `T`. The `DeviceType` template argument indicates both the memory in which the data reside (e.g., host or device memory) and the place to execute the code (e.g., CPU or GPU). Once bounding boxes are constructed, they are passed to the constructor of `BVH`, the ArborX class containing the hierarchy (see Figure 3).

Next, the queries are built. Each query corresponds to a pair of a query point and a number of neighbors to be found (nearest query), or a pair of a query point and a radius (spatial query).

```
// Create the View for the bounding boxes
Kokkos::View<ArborX::Box*, DeviceType> bounding_boxes("bounding_boxes", n_boxes);
// Fill in the bounding boxes Kokkos::View
...
// Create the bounding volume hierarchy
ArborX::BVH<DeviceType> bvh(bounding_boxes);
```

Fig. 3. BVH construction interface.

```
// Create the View for the spatial-based queries
Kokkos::View<ArborX::Within *, DeviceType> queries("queries", n_queries);
// Fill in the queries
using ExecutionSpace = typename DeviceType::execution_space;
Kokkos::parallel_for("setup_queries",
    Kokkos::RangePolicy<ExecutionSpace>(0, n_queries), KOKKOS_LAMBDA(int i) {
      queries(i) = ArborX::within(query_points(i), radius);
   });
// Perform the search
Kokkos::View<int*, DeviceType> offsets("offset", 0);
Kokkos::View<int*, DeviceType> indices("indices", 0);
bvh.query(queries, indices, offsets, buffer_size);
```

Fig. 4. BVH search interface (spatial queries).

The spatial-based version is shown in Figure 4, with queries being filled in a device parallel loop. Once the queries are constructed, two views are allocated to store the results: the `indices` view to contain the indices associated with the bounding boxes that satisfy the queries and the `offsets` view to contain the offsets in `indices` associated with each query. Two views are necessary as the number of results for each query may differ (for example, the number of results within specific radius for spatial-based search).[2] The search is done by invoking the `query` function of BVH. The additional parameter `buffer_size` is optional, and only used for spatial-based queries. It indicates the user-provided estimate for the number of returned results, and if accurate, allows to do a single pass (1P).

3 NUMERICAL RESULTS

The numerical studies presented in the article were performed on two systems:

- CADES system with each node containing two Intel Xeon E5-2695 v4 18-core CPUs running at a clock speed of 2.1 GHz with 256 GB of main memory;
- OLCF Summit system with each node having two IBM POWER9 AC922 21-core CPUs, each having 4 hardware threads with 6 Nvidia Volta V100 GPUs connected by NVLink 2.0 [OLCF 2018].

We used the Google Benchmark tool [Google 2018] in our experiments, using the median of the runs for the results we have reported here.

In the results that follow, the spatial search performed using 2P approach is denoted by "ArborX (2P)."

[2]This format is similar to that of compressed sparse row format that is commonly used to store sparse matrices.

3.1 Experimental Datasets

In our experiments, we use several artificial datasets proposed in Elseberg et al. [2012]. We consider two shape forms, cube and sphere. For a given shape, a set of points is then chosen either from within the selected shape (filled variant), or from its boundary (hollow variant). To generate p points, set $a = p^{1/3}$, $\Omega = [-a, a]^3$ and proceed as follows:

- *filled cube:* Each random point is drawn randomly from Ω with uniform distribution;
- *hollow cube:* Points are placed on the faces of Ω in a cyclic manner, with the position of the point on each face being random with uniform distribution;
- *filled sphere:* Points are randomly chosen from Ω and accepted based on being within a sphere of radius a centered at 0;
- *hollow sphere:* Points are first generated within $[-1, 1]^3$ cube and then projected to the sphere of radius a centered at 0.

In our experiments, we consider two cases: searching for a filled sphere cloud of query points in the filled cube cloud (filled case), and searching for a hollow sphere cloud in the hollow cube cloud (hollow case). The major difference between these two cases is the workload per thread. For the filled case, the data and the query results are balanced between threads. The hollow case, however, presents a challenge due to a wide imbalance of query results, as only a few threads will produce positive results for a spatial search.

For a given problem, m source points and n target points are generated using one of the described four shapes. The number of neighbors k for the nearest search is fixed to 10 in all experiments. The radius r for spatial search is chosen in such a way that on average there are k neighbors within radius r in a filled cube shape.

3.2 Comparison with Available Libraries

In this section, we compare the performance of ArborX with that of two state-of-the-art existing libraries, Boost.Geometry.Index and nanoflann.

The Boost.Geometry.Index [Gehrels et al. 2017] library implements different algorithms for R-trees. For the purpose of performance comparison, we used the packing algorithm [García et al. 1998; Leutenegger and Edgington 1997], which is the most performant algorithm contained in Boost.Geometry.Index. The performance comes at the cost of flexibility, since the tree has to be built statically. We used version 1.67.0 of the library.

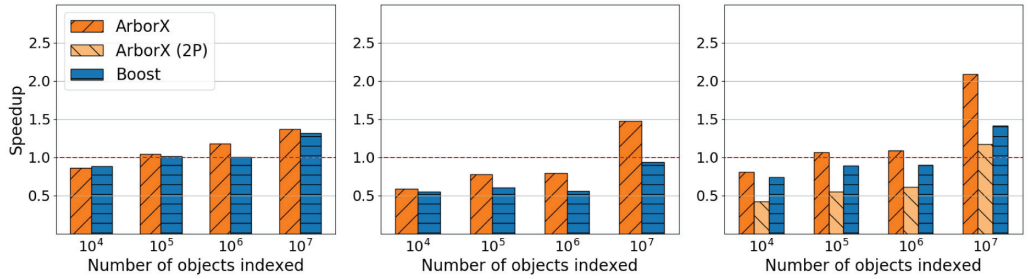
nanoflann [Blanco and Rai 2014] is a header-only library for building k-d trees. We used nanoflann hash 3b2065e.

As Boost.Geometry.Index and nanoflann are implemented only in serial,[3] the comparisons in this subsection were done using one thread. The scaling of ArborX with the number of OpenMP threads is demonstrated in the next section.

The experiments were performed on the CADES system. They were run for the increasing number of source points m, ranging from 10^4 to 10^7. The number of the target points n was chosen to be the same as the number of source points, $n = m$. Such configuration is common in many applications, e.g., finding potentially colliding pairs of objects in graphics applications, or finding nearby particles for pairwise interactions in physics simulations.

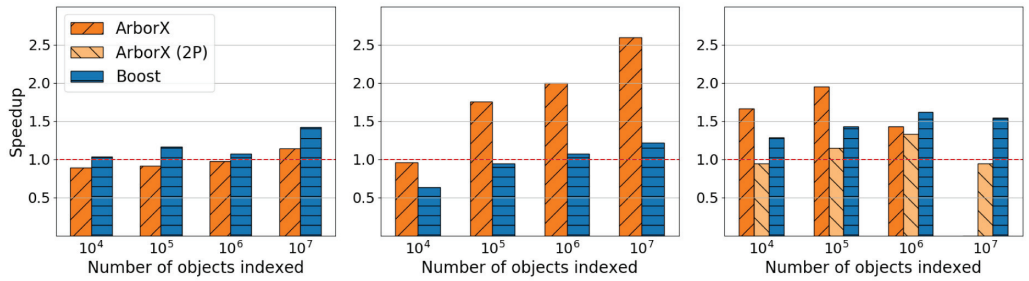
Figures 5 and 6 demonstrate the relative speedup or slowdown of the libraries relative to nanoflann. Figure 5(a) (Figure 6(a)) shows tree construction speedup for the filled (hollow) case, respectively. We observe that ArborX and Boost.Geometry.Index libraries perform similarly, while

[3] As Boost.Geometry.Index is thread safe, it is theoretically possible to run it in batched mode. However, this will require a user to write the necessary parallel implementation.



(a) Construction (b) Nearest search (c) Spatial search

Fig. 5. Comparison of libraries for the filled case. The speedup is shown with respect to nanoflann.



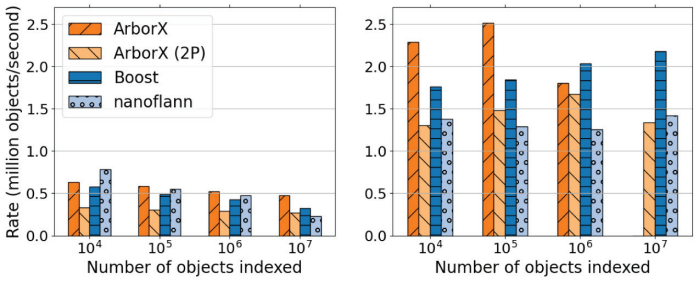
(a) Construction (b) Nearest search (c) Spatial search

Fig. 6. Comparison of libraries for the hollow case. The speedup is shown with respect to nanoflann.

nanoflann starts to lose its competitiveness for large number of objects. Comparing the query performance of the libraries, we observe that for the nearest search (Figures 5(b) and 6(b)), ArborX significantly outperforms both Boost.Geometry.Index and nanoflann libraries for larger numbers of objects, particularly for the hollow case. For the spatial search, the performance of the 1P variant is about twice faster than that of the 2P variant in the filled case (Figure 5(c)), where workloads are balanced. However, for a larger number of objects (larger than 10^6) in the hollow case, the 1P variant could not be run due to requiring too much memory to preallocate the storage for the results based on the maximum estimate. Therefore, no results from the 1P variant are given for these larger cases in Figure 6(c).

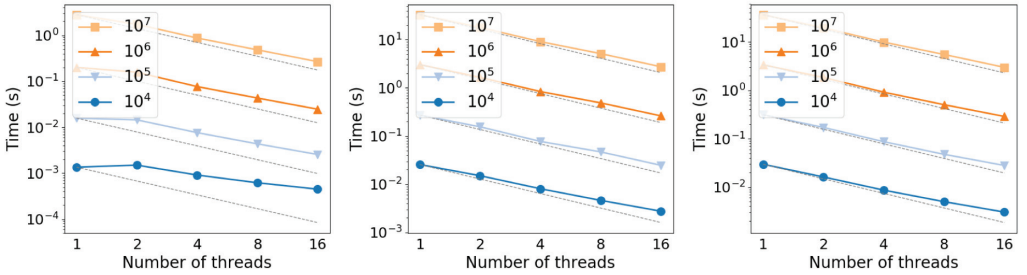
Figure 7(a) (Figure 7(b)) demonstrates the rate of spatial-based search for a filled (hollow) case, respectively. The main difference between the filled and hollow variants is the number of results returned by queries. Specifically, for the spatial-based search, the filled variant returns 10 neighbors on average (with the minimum being 0 and the maximum being 32). However, for the hollow variant the number of neighbors is much more imbalanced, ranging from 0 to 522, with the average being 2. This is due to (a) the fact that the hollow sphere touches the hollow cube in just a few places (centers of the faces), and (b) both being two-dimensional objects thus having a significantly higher density than in 3D for the same number of points. As expected, the rate for the hollow variant is significantly higher than that of the filled variant due to most queries returning empty result.

An acute observer may notice that there is little difference between 1P and 2P variants for the hollow case for large number of objects. It turns out that the extra memory required to store all the results of the first pass becomes a drawback at some point, with filtering out unused entries taking a significant portion of time. Another interesting observation is the drop of the rate for 2P



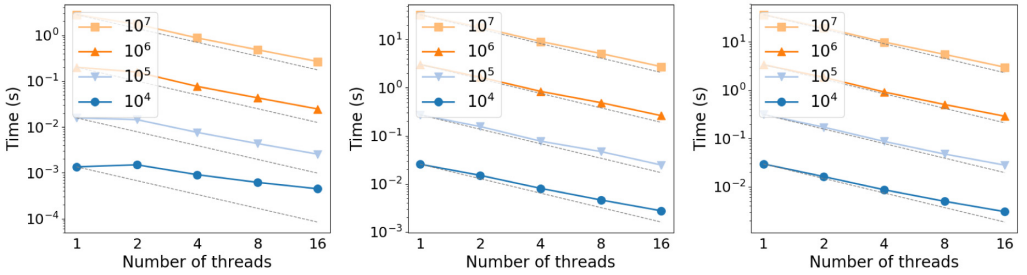
(a) Filled case (b) Hollow case

Fig. 7. Spatial search rates for the libraries.



(a) Construction (b) Nearest search (c) Spatial search

Fig. 8. ArborX scaling for the filled case.



(a) Construction (b) Nearest search (c) Spatial search

Fig. 9. ArborX scaling for the hollow case.

spatial search for the hollow case going from 10^6 to 10^7. Examination revealed that for the latter problem sorting queries is less effective than retaining their original order. Sorting the queries in serial may not be necessary, and ArborX provides an option to disable that.

3.3 Multi-threaded Strong Scaling

We next examine the scalability of ArborX using OpenMP on the CADES system.

For the strong scaling, the number of source points m is fixed to a value from 10^4 to 10^7, and the number of OpenMP threads increased from 1 to 16. The number of target points n was chosen to be the same as the number of source points, $n = m$.

Table 1. ArborX Scaling Results for the Filled Case for $n = 10^4$ and $n = 10^7$

Threads	Construction		Spatial search		Nearest search	
	$n = 10^4$	$n = 10^7$	$n = 10^4$	$n = 10^7$	$n = 10^4$	$n = 10^7$
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.90	1.62	1.83	1.87	1.71	1.84
4	1.48	3.21	3.42	3.69	3.18	3.62
8	2.18	5.82	5.93	6.64	5.55	6.44
16	3.01	10.50	9.74	12.24	9.31	12.06

Table 2. ArborX Scaling Results for the Hollow Case for $n = 10^4$ and $n = 10^7$

Threads	Construction		Spatial search		Nearest search	
	$n = 10^4$	$n = 10^7$	$n = 10^4$	$n = 10^7$	$n = 10^4$	$n = 10^7$
1	1.00	1.00	1.00	1.00	1.00	1.00
2	0.94	1.46	1.49	1.57	1.71	1.87
4	1.48	2.87	2.62	3.05	3.14	3.62
8	2.19	5.05	4.11	5.39	5.59	6.44
16	2.99	8.61	5.64	11.20	8.52	9.84

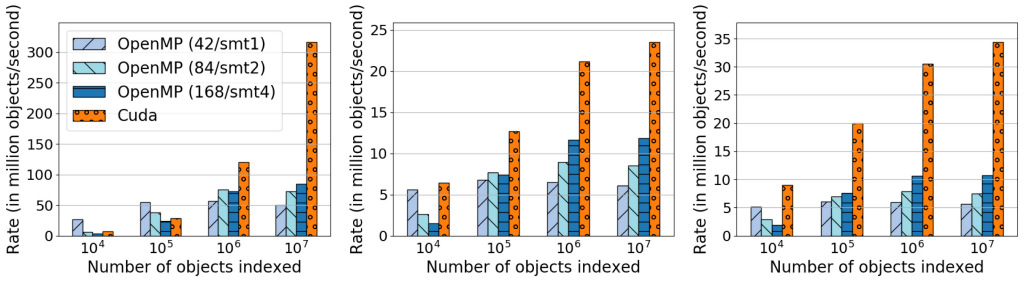
The results are presented in Figures 8 and 9 and Tables 1 and 2 (the spatial search is performed using the 2P approach). ArborX demonstrates good scalability for a large number of objects. However, having too few objects per thread for smaller simulations results in suboptimal scaling. Upon further inspection, the sorting routine used for sorting Morton indices was identified to be the limiting factor, having poor scalability in cases where every thread has fewer than 1000 objects. We attempted to use few available parallel sort algorithms (e.g., `__gnu_parallel::sort`) instead of the default Kokkos sort. However, we found this led to only a minor improvement in our performance results. This issue affects both construction (sorting of Morton codes, see Section 2.1), and search (sorting of queries, see Section 2.2.3) and will be a topic of future work.

3.4 Accelerator Comparison

We now compare the performance of ArborX using OpenMP and CUDA on the OLCF Summit system. We compare the performance of the two POWER9 CPUs on a single Summit node (42 physical cores) with that of a single Volta V100 GPU.[4] POWER9's physical cores consist of four "slices" that can be used in a variety of configurations. In *smt4* mode, each slice operates independently of the other three, allowing for separate streams of execution for multiple OpenMP threads on each physical core. In *smt2* mode, pairs of slices work together to run tasks. Finally, in *smt1* mode the four slices work together to execute the task/thread assigned to the physical core.

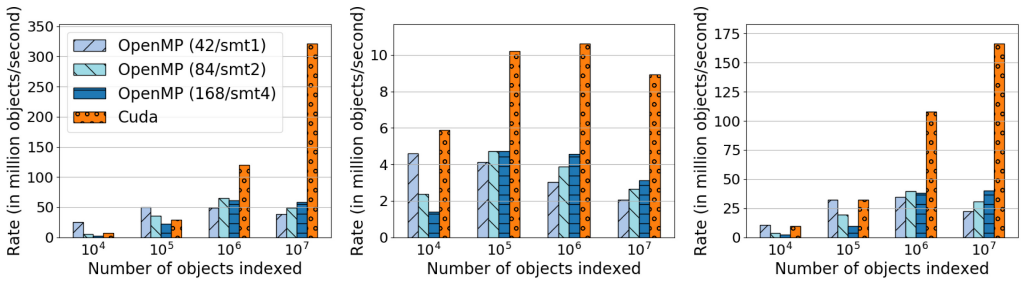
The results are presented in Figures 10 and 11 (the spatial search is performed using the 2P approach). We observe that executing in *smt4* mode usually leads to better performance (especially, for larger problem sizes), though that is not always the case. We also note that a single Summit GPU significantly exceeds the performance of the full node of CPUs, exhibiting shortcomings only for smaller problems that are less suitable for the high parallelism provided by the accelerator. As each node of Summit has six GPUs connected by NVLink, it is expected that using only accelerators

[4]Currently, Kokkos (and thus ArborX) does not support multiple GPUs within the same process. The six V100 GPUs on a Summit node are typically used by having a single MPI rank manage a single dedicated GPU.



(a) Construction (b) Nearest search (c) Spatial search

Fig. 10. Comparison of OpenMP and CUDA on Summit for filled cube source and filled sphere target clouds.



(a) Construction (b) Nearest search (c) Spatial search

Fig. 11. Comparison of OpenMP and CUDA on Summit for hollow cube source and hollow sphere target clouds.

would dramatically outperform using only CPUs when using an MPI+Kokkos approach in the future.

We also concede that another way to compare the performance would have been to normalize the data by the power drawn by the corresponding hardware. Unfortunately, we were not able to obtain such data as it is not yet provided by the facility or the performance monitoring software. While the published specifications of V100 and POWER9 do have some information, including thermal design power, those were observed to be inaccurate in practice by others, and thus were not relied on in our reporting.

4 CONCLUSION AND OUTLOOK

In this article, we presented a new library ArborX for searching close geometric objects in space. ArborX's strength lies in its performance and its ability to be run on multiple hardware architectures using a single interface definition. Experiments were conducted to compare its performance with existing popular libraries, such as nanoflann and Boost.Geometry.Index, and to demonstrate its scalability and performance on accelerators. Our results show that our implementation is competitive with these libraries in single thread execution and is also able to effectively leverage both the multithreaded CPU and GPU compute power on modern leadership-class supercomputers such as Summit.

There are two natural directions for future work. One is addressing the current scalability limitations through careful analysis and profiling of the library. The second is implementing the distributed search algorithms using MPI to address the requirements of exascale applications where

the objects indexed by the tree as well as the query objects are distributed across many MPI ranks. This creates additional challenges to those presented in this article as it is likely that the data that one searches for may not belong to the same node, or that the data distribution among MPI ranks may be imbalanced. Thus, a communication layer deploying a load balancing strategy will be required to be effectively scale to thousands of accelerated compute nodes.

REFERENCES

C. Apetrei. 2014. Fast and simple agglomerative LBVH construction. In *Proceedings of the Annual Conference on Computer Graphics and Visual Computing (CGVC'14)*, Rita Borgo and Wen Tang (Eds.). The Eurographics Association. DOI: https://doi.org/10.2312/cgvc.20141206

C. Benthin, I. Wald, S. Woop, and A. T. Áfra. 2018. Compressed-leaf bounding volume hierarchies. In *Proceedings of the Conference on High-Performance Graphics (HPG'18)*. Association for Computing Machinery, New York, NY, Article Article 6. DOI: https://doi.org/10.1145/3231578.3231581

J. L. Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (September 1975), 509–517. DOI: https://doi.org/10.1145/361002.361007

P. J. Besl and N. D. McKay. 1992. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.* 14, 2 (February 1992), 239–256. DOI: https://doi.org/10.1109/34.121791

J. L. Blanco and P. K. Rai. 2014. nanoflann: A C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. Retrieved from https://github.com/jlblancoc/nanoflann.

M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. 2000. LOF: Identifying density-based local outliers. In *ACM SIGMOD Record*, Vol. 29. ACM, 93–104.

DOE. 2016. Exascale Computing Project. Retrieved from https://www.exascaleproject.org.

L. R. Domingues and H. Pedrini. 2015. Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics (HPG'15)*. ACM, New York, NY, 13–20. DOI: https://doi.org/10.1145/2790060.2790065

H. C. Edwards, C. R. Trott, and D. Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput.* 74, 12 (2014), 3202–3216. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003

J. Elseberg, S. Magnenat Rol, and S. A. Nüchter. 2012. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *J. Softw. Eng. Robot.* 3, 1 (2012), 2–12.

Y. T. Feng and D. R. J. Owen. 2002. An augmented spatial digital tree algorithm for contact detection in computational mechanics. *Int. J. Num. Methods Eng.* 55, 2 (2002), 159–176.

K. Garanzha, J. Pantaleoni, and D. McAllister. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 59–64.

R. García, J. Yván, M. A. López, and S. T. Leutenegger. 1998. A Greedy algorithm for bulk loading R-trees. In *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (GIS'98)*. ACM, New York, NY, 163–164. DOI: https://doi.org/10.1145/288692.288723

B. Gehrels, B. Lalande, M. Loskot, A. Wulkiewicz, M. Karavelas, and M. Fisikopoulos. 2017. Boost geometry. Retrieved from https://www.boost.org/doc/libs/1_67_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html.

Google. 2018. Google benchmark. Retrieved from https://github.com/google/benchmark.

A. Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. ACM, New York, NY, 47–57. DOI: https://doi.org/10.1145/602259.602266

H. Haverkort. 2004. *Results on Geometric Networks and Data Structures*. Ph.D. Dissertation. University of Utrecht.

M. P. Howard, A. Statt, F. Madutsa, T. M. Truskett, and A. Z. Panagiotopoulos. 2019. Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units. *Comput. Mater. Sci.* 164 (2019), 139–146.

Y. Hu, W. Wang, D. Li, Q. Zeng, and Y. Hu. 2019. Parallel BVH construction using locally-density clustering. *IEEE Access* PP (07 2019), 1–1. DOI: https://doi.org/10.1109/ACCESS.2019.2932151

T. Karras. 2012a. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the 4th ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (EGGH-HPG'12)*. Eurographics Association, Goslar, Germany, 33–37. DOI: https://doi.org/10.2312/EGGH/HPG12/033-037

T. Karras. 2012b. Thinking parallel, Part I: Collision detection on the GPU. Retrieved from https://devblogs.nvidia.com/thinking-parallel-part-i-collision-detection-gpu.

T. Karras. 2012c. Thinking parallel, Part II: Tree traversal on the GPU. Retrieved from https://devblogs.nvidia.com/thinking-parallel-part-ii-tree-traversal-gpu.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 375–384.

M. A. Leutenegger, S. T. Lopez, and J. Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering*. 497–506. DOI: https://doi.org/10.1109/ICDE.1997.582015

M. Muja and D. G. Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proceedings of the International Conference on Computer Vision Theory and Application (VISSAPP'09)*. INSTICC Press, 331–340.

OLCF. 2018. OLCF Summit. Retrieved from https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

J. Pantaleoni and D. Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*. 87–95.

M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, Prabhat, and P. Dubey. 2016. PANDA: Extreme scale parallel K-nearest neighbor on distributed architectures. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. 494–503. DOI: https://doi.org/10.1109/IPDPS.2016.57

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* 12, 85 (2011), 2825–2830.

C. Sewell, L. Lo, K. Heitmann, S. Habib, and J. Ahrens. 2015. Utilizing many-core accelerators for halo and center finding within a cosmology simulation. In *Proceedings of the 2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 91–98.

S. R. Slattery. 2016. Mesh-free data transfer algorithms for partitioned multiphysics problems: Conservation, accuracy, and parallelism. *J. Comput. Phys.* 307 (2016), 164–188. DOI: https://doi.org/10.1016/j.jcp.2015.11.055

M. Vinkler, J. Bittner, and V. Havran. 2017. Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of the High Performance Graphics (HPG'17)*. ACM, New York, NY, Article 9, 9:1–9:8 pages. DOI: https://doi.org/10.1145/3105762.3105782

Received August 2019; revised June 2020; accepted July 2020