Efficient BVH Construction via Approximate Agglomerative Clustering

Yan Gu Yong He Kayvon Fatahalian Guy Blelloch

Carnegie Mellon University

Abstract

We introduce Approximate Agglomerative Clustering (AAC), an efficient, easily parallelizable algorithm for generating high-quality bounding volume hierarchies using agglomerative clustering. The main idea of AAC is to compute an approximation to the true greedy agglomerative clustering solution by restricting the set of candidates inspected when identifying neighboring geometry in the scene. The result is a simple algorithm that often produces higher quality hierarchies (in terms of subsequent ray tracing cost) than a full sweep SAH build yet executes in less time than the widely used top-down, approximate SAH build algorithm based on binning.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: Ray tracing, bounding-volume hierarchy, agglomerative clustering

1 Introduction

In recent years there has been considerable interest in the design of efficient, parallel algorithms for constructing bounding volume hierarchies (BVH) for ray tracing. The current best practice technique [Wald 2007] for constructing a high-quality hierarchy in parallel with reasonable cost on a multi-core CPU uses a divide-and-conquer algorithm (top-down) that establishes geometry groupings according to an approximate surface area heuristic [Goldsmith and Salmon 1987]. While higher quality structures can be generated using agglomerative (bottom-up) construction schemes [Walter et al. 2008], these methods, even with recent optimizations, have not been demonstrated to be performance competitive with best practice divisive implementations.

In this paper we introduce a new algorithm that efficiently constructs BVHs in a bottom-up manner using agglomerative clustering. The resulting hierarchies are often of higher quality than those produced by a top-down, full sweep SAH build, and they can be constructed in parallel in less time than a fast approximate SAH build that employs the widely used optimization of "binning" [Wald 2007]. Unlike previous work on bottom-up BVH construction, our algorithm realizes high performance by generating a structure that is a close approximation to (but does not necessarily match) the scene geometry's greedy agglomerative clustering solution.

2 Background

The pursuit of practical, real-time ray tracing systems for many-core CPUs and GPUs has inspired many efforts to reduce the time required to build high-quality ray-tracing acceleration structures.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPG 2013, July 19 – 21, 2013, Anaheim, California.
Copyright © ACM 978-1-4503-2135-8/13/07 $15.00

Algorithm 1 BVH construction using $O(N^3)$ agglom. clustering

```
Input:   scene primitives P = {P_1, P_2, ..., P_N}.
Output: a scene BVH (returns root node)
  Clusters C = P;              // initialize with singleton clusters
  while Size(C) > 1 do
    Best = ∞;
    for all C_i ∈ C do
      for all C_j ∈ C do
        if C_i ≠ C_j and d(C_i, C_j) < Best then
          Best = d(C_i, C_j);
          Left = C_i; Right = C_j;
        end if
      end for
    end for
    Cluster C' = new Cluster(Left, Right);
    C = C − {Left} − {Right} + {C'};
  end while
  return C;
```

(A high-quality structure reduces the cost of performing ray-scene intersection queries.) Promising techniques have been documented extensively, and we refer the reader to [Wald 2007; Wald et al. 2007] and [Karras 2012] for excellent summaries of the design space and key challenges of modern approaches.

In this paper, we focus on the specific problem of performing a full ("from scratch") construction of a bounding volume hierarchy from scene geometry. At present, the preferred approach to constructing a BVH on many-core CPU architectures is to build the hierarchy top-down, using primitive binning as a cheap approximation to evaluating the full surface area heuristic. Parallel implementations of this approach on modern architectures are described in detail in [Wald 2007; Wald 2012]. GPU-based implementations of BVH construction discard use of the surface area heuristic in favor of maximizing parallelism during construction of the upper levels of the hierarchy [Lauterbach et al. 2009; Karras 2012]. These implementations achieve high performance, but produce lower quality BVHs, making them less desirable for use in ray tracing.

Our efforts are inspired by previous work by [Walter et al. 2008] which explores the use of agglomerative clustering (as opposed to top-down divisive methods) to build high-quality BVHs. Agglomerative clustering is a greedy algorithm that is initialized with each scene primitive as a singleton geometry cluster, then repeatedly combines the two nearest clusters in the scene into one. This process continues until a single cluster, representing all primitives in the scene, remains. Agglomerative clustering is a popular technique in many fields as it can produce high-quality hierarchies and also generalizes to arbitrary cluster distance functions, but its use has been limited in computer graphics due to its high cost. Solutions have $O(N^2)$ complexity [Olson 1995], but for clarity we provide a naive, $O(N^3)$ algorithm for constructing a BVH using agglomerative clustering in Algorithm 1. In the pseudocode $d(C_i, C_j)$ is the cluster distance function. In this paper we follow [Walter et al. 2008] and define $d(C_i, C_j)$ as the surface area of the bounding box enclosing primitives from both clusters.

An expensive operation in an agglomerative BVH build is finding

the nearest neighbor (e.g., closest cluster) to a given cluster. Walter et al.'s key idea was to accelerate this search by storing remaining clusters in a K-D tree constructed using a low-cost, top-down partitioning strategy such as coordinate bisection. Thus their approach uses a low-quality, but cheap-to-compute, hierarchical acceleration structure to accelerate construction of a high-quality one. Walter et al. used this idea to design two agglomerative BVH build algorithms (heap-based clustering and locally-ordered clustering) that were empirically shown to scale to large scenes and, for random rays, reduced tracing cost 15-30% when compared to a top-down binned SAH build. Although this work established agglomerative clustering as a viable strategy for BVH construction, the resulting build times remained slow compared to top-down methods. The algorithm required complicated insertion and deletion of clusters from a K-D tree (or a periodic K-D tree rebuild) and its parallelization requires speculative execution. Although Walter et al. reported performance within a factor of two of a top-down build for unoptimized single-core implementations in Java, our experiences reimplementing and optimizing these techniques in C++ yield single-core performance that is (even after non-trivial effort) about four to seven times slower than a binned SAH build (see Table 2). The primary contribution of this paper is a new algorithm, based on agglomerative clustering, that yields similar quality BVHs to Walter et al.'s solutions, but is simpler to implement, and offers performance comparable to, and often better than, a top-down binned BVH build. Furthermore, it parallelizes well on a modern many-core CPU.

3 Approximate Agglomerative Clustering

The key observation underlying our new agglomerative clustering technique is that the cost of forming new clusters via locally-ordered or heap-based agglomerative clustering [Walter et al. 2008] is *highest* at the beginning of the construction process when the number of clusters is large. Since these techniques are initialized with one cluster per scene primitive (N initial clusters), each nearest neighbor search at the beginning of hierarchy construction is a global operation incurring cost at least $O(\log N)$. Thus, constructing the bottom-most nodes of the resulting BVH dominates overall computation cost. This property is particularly undesirable since combining the original singleton clusters is a highly local operation that need not incur the high cost of a scene-wide search.

Thus, the main idea of our approach is to accelerate BVH construction using an *approximate agglomerative clustering* (AAC) method that restricts nearest neighbor search to a small subset of neighboring scene elements. We show that this approximation minimally impacts resulting BVH quality but significantly accelerates the speed of BVH construction.

Conceptually the idea of the AAC method (illustrated in Figure 1) is to quickly organize scene primitives into a binary tree based on recursive coordinate bisection—i.e., at each level of the tree one coordinate (x, y, or z) is bisected. Leaf nodes in this tree (which we call the *constraint tree*) contain a small set of scene primitives (singleton clusters) whose centers fall within the corresponding region of space. BVH construction via agglomerative clustering then proceeds bottom up, with the constraint tree controlling what primitives can be clustered. In particular each node of the constraint tree generates a set of un-combined clusters by taking the union of just the un-combined clusters generated by its two children and reducing the size of this union by greedily combining some of the clusters. The number of un-combined clusters returned by each node is controlled by the algorithm and is somewhere between 1 (everything has been combined into a single cluster) and the number of primitives in the subtree (nothing has been combined yet). At the root of the constraint tree the clusters are reduced to a single cluster using a final step of true greedy agglomerative clustering.

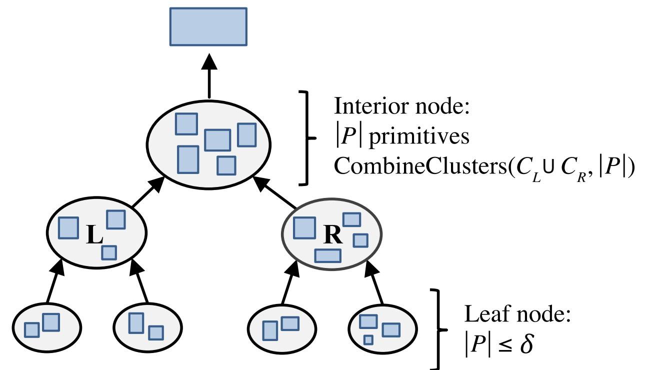


Figure 1: *Illustration of the AAC algorithm's "constraint tree" with groups of at most δ primitives at the leaves. Each node represents a computation that combines clusters from its two children using agglomerative clustering (see* `CombineClusters`*). The size of a node in the figure represents the number of clusters returned by the node, which increases going up the tree. The last step of the algorithm (top node) combines all remaining clusters into a single cluster corresponding to the BVH root node.*

An important property of the method that balances efficiency and accuracy is to increase the number of clusters generated by the nodes as we proceed up the tree, but decrease the total number across each level. This keeps the computation required near the leaves small, but allows for more accuracy at modest costs higher in the tree where there are fewer nodes. We control the number of clusters generated by a node using a cluster count reduction function f, described below.

3.1 Algorithm Basics

We now go into more detail on how we efficiently implement the AAC method. Pseudocode is given in Algorithms 2, 3 and 4.

Primitive sort. The algorithm first sorts scene primitives according to the Morton code (z-order) of their bounding box center [Gargantini 1982; Bern et al. 1999]. Generating the Morton code from primitive centers and subsequent radix sort are cheap to compute and easily parallelized. Our implementation constructs a Morton code with $\lceil \log_4 N \rceil$ bits per dimension and uses a variable-bit radix sort that executes in time linear in the number of primitives (see Section 3.4.1 for details). Indeed the Morton code is a common technique used in several high-performance (but lower tree quality) algorithms for BVH construction [Lauterbach et al. 2009; Karras 2012]. Unlike these algorithms, the AAC algorithm does not use the ordering directly to build a BVH. Rather it uses the implicit structure to define constraints on cluster combining.

Constraint tree traversal ("downward phase"). Following the sort, the AAC algorithm invokes the recursive function `BuildTree` (Algorithm 3) to traverse the implicit constraint tree structure organizing the primitives. (Note that the constraint tree is implicit in the Morton code sort, it is never explicitly constructed.) Following [Lauterbach et al. 2009], in each traversal step, `MakePartition` (line 5) bisects the current spatial extent based on the next bit in the Morton code. This serves to partition the primitive set P into subsets P_L and P_R. The partition is easily computed using binary search through P to find where the Morton code for the primitives switches from 0 to 1 at the current bit position. As noted previously [Gargantini 1982; Lauterbach et al. 2009], the Morton code sort ensures that all primitives on the 0 side of the flip point in P are on the low side of the coordinate bisection (in P_L), and primitives on the 1 side are on the high side (P_R), so no data movement is

Algorithm 2 AAC(P)

Input: list of all scene primitives P
Output: BVH containing primitives in P

```
1: Compute Morton code for centers of P;
2: RadixSort P by Morton code;
3: C = BuildTree(P);
4: return CombineClusters(C, 1);
```

Algorithm 3 BuildTree(P)

Input: subset of scene primitives P
Output: at most $f(|P|)$ clusters containing primitives in P

```
1: if (|P| < δ) then
2:    Initialize C with P;
3:    return CombineClusters(C, f(δ));
4: end if
5: ⟨P_L, P_R⟩ = MakePartition(P);
6: C = BuildTree(P_L) ∪ BuildTree(P_R);
7: return CombineClusters(C, f(|P|));
```

necessary when forming these subsets. Therefore the work spent on tree traversal is $T(n) = T(n-a)+T(a)+O(\log(\min(a, n-a)))$, which is linear in n for $a \geq 1$. In the rare event that the algorithm exhausts the Morton code bits during traversal without partitioning the primitives into a sufficiently small set, we continue traversal by partitioning P in half each step.

Bottom-up clustering ("upward phase"). Agglomerative clustering begins once the number of primitives in P drops below the threshold δ (Algorithm 3, line 1), where δ is a parameter of the algorithm. Like locally-ordered clustering, the AAC algorithm initializes cluster formation by creating one singleton cluster per primitive (line 2). We select δ to be small (in Section 4 we demonstrate high-quality builds using $\delta = 4$ and 20).

The algorithm `CombineClusters` is the key component of the bottom up clustering. It is used to reduce the number of clusters at every node of the constraint tree to $f(|P|)$, where the cluster count reduction function f is a parameter of the algorithm. At the leaves of the tree this reduces the initial set of primitives to the required size (line 3), and at internal nodes it reduces the union of the two sets of clusters from its children to the required size (lines 6 and 7). Since the number of clusters in C is never that large, we use a brute force method to reduce the number of clusters as described in Algorithm 4. The method `findBestMatch` implements linear search over all clusters in C. This algorithm takes at least $O(|C|^2)$ time. `CombineClusters` then greedily selects the closest pair of matches by find the best best-match, and combines this pair. This is repeated until n clusters remain. We defer further details about our optimized implementation of `CombineClusters` to Section 3.3.

Each invocation of `Buildtree`(P) returns a list of $f(|P|)$ clusters produced via agglomerative clustering of the contents of C. While the recursive splitting phase of `BuildTree` ("downward" phase) serves to partition scene primitives into local groups, the recombination phase that follows the return of child recursive calls ("upward" phase) serves to agglomerate the clusters resulting from the P_L and P_R primitive sets. The call to `CombineClusters` on line 7 of Algorithm 3 combines the clusters returned from the left and right constraint tree nodes into a smaller number of clusters determined by the function f. Notice that clusters containing primitives in *different child primitive sets* as given by the constraint tree can be combined into the same output cluster in this step. The recombination phase continues until reaching the root of the constraint tree, which returns $f(|P|)$ clusters. Last, the remaining clusters are reduced to a single cluster on line 4 of Algorithm 2. The result is a BVH containing all primitives in the scene.

Algorithm 4 CombineClusters(C, n)

Input: list of clusters C
Output: list of at most n clusters

```
1: for all C_i ∈ C do
2:    C_i.closest = C.FindBestMatch(C_i);
3: end for
4: while |C| > n do
5:    Best = ∞;
6:    for all C_i ∈ C do
7:       if d(C_i, C_i.closest) < Best then
8:          Best = d(C_i, C_i.closest);
9:          Left = C_i; Right = C_i.closest;
10:      end if
11:   end for
12:   c' = new Cluster(Left, Right);
13:   C = C − {Left, Right} + {c'};
14:   c'.closest = C.FindBestMatch(c');
15:   for all C_i ∈ C do
16:      if C_i.closest ∈ {Left, Right} then
17:         C_i.closest = C.FindBestMatch(C_i);
18:      end if
19:   end for
20: end while
21: return C;
```

Algorithm Parameters. The AAC algorithm has two parameters: the *cluster count reduction function* f and the *traversal stopping threshold* δ. These parameters modify the extent to which the algorithm approximates the full agglomerative clustering solution and thus affect the quality of the generated BVH and its construction cost. The result of the function $f(x)$ can vary between 1 and x. A value of 1 yields a cheap to compute, but poor-quality BVH that is similar to directly building the BVH from the results of the Morton sort [Lauterbach et al. 2009]. Setting $f(x) = x$ defers all cluster agglomeration to the root node of the bisection tree, causing the AAC algorithm to degenerate into a true greedy agglomerative clustering method. To achieve performance between these two extremes, our implementation of AAC algorithm uses functions of the form $f(x) = cx^\alpha$ for $0 \leq \alpha \leq 1$ and some constant c (see Section 3.2).

Discussion. Before analyzing the overall complexity of the algorithm and describing implementation details, we highlight key differences between AAC construction and locally-ordered clustering.

Firstly, the AAC build's output is an approximation to the results of locally-ordered clustering (the two algorithms may generate different BVHs). While locally-ordered clustering yields the same result as a brute-force clustering implementation run on all scene primitives (it will always cluster the two closest clusters), the primitive partitioning performed during AAC build's downward phase constrains what clusters can be formed. For example, it is possible for partitioning to separate the two closest primitives in the scene. Although these primitives will likely be grouped in some interior node of the resulting BVH, they will not be contained in the same leaf cluster.

Since each constraint tree node returns many un-combined clusters as determined by the cluster count reduction function f, the AAC algorithm has flexibility to delay combining of certain clusters until higher levels of the constraint tree. In particular clusters containing large primitives will get pushed up the constraint tree without combining until they get to a node which represents a spatial region of the "appropriate size"—i.e. for which all clusters are about as

large. This is because the greedy method combines clusters with the smaller spatial bounds first (recall $d(C_i, C_j)$ is the resulting aggregate bounding box size), leaving large ones un-combined. Combining large primitives high in the constraint tree is what makes agglomerative clustering work well in comparison to top-down methods when there is large variation in the size of scene primitives, and is a reason why the AAC algorithm works almost as well as the true greedy agglomerative clustering.

The second difference is that locally-ordered clustering uses a K-D tree to accelerate finding the minimum distance cluster from a potentially large collection (this adds significant complexity to the algorithm since the tree must support cluster insertions and deletions during the clustering process). In contrast, at both the leaf and interior levels of the constraint tree, the AAC build *never* performs clustering on a large number of elements. As a result, there is no need for a K-D tree to accelerate search. In an AAC build, presorting geometry and the constraint tree traversal process serve to limit the number of clusters processed at any one point by the algorithm (to a local set), not to accelerate nearest neighbor search over a scene-wide collection.

3.2 Cost Analysis

In this section we analyze the AAC algorithm for the case where recursive coordinate bisection yields a fully balanced constraint tree, as well as under the worst-case assumption of a completely unbalanced tree. In practice we observe performance close to the fully balanced case. In this analysis we also assume the execution of `CombineClusters` has $O(|C|^2)$ cost, which we observe to be the case in practice although it is theoretically possible to construct geometric configurations that require $O(|C|^3)$ work. As discussed earlier in Section 3.1, the cost of sorting the Morton codes and the combined cost of `MakePartition` is linear in the number of primitives.

A fully balanced constraint tree with height H and leaves containing δ primitives has $2^H = N/\delta$ leaves (where N is the total number of primitives contained in the tree). Thus, execution of `CombineClusters` for leaf nodes constitutes $O(2^H \times \delta^2)$ work. Processing `CombineClusters` in all H levels requires work:

$$O\left(\sum_{i=0}^{H} 2^{H-i} \cdot f\left(\delta \cdot 2^i\right)^2\right)$$

If we assume $f(x) = c \cdot x^{1/2}$, the amount of work done on each level is about the same and the time complexity is:

$$O\left(\sum_{i=0}^{H} 2^{H-i} \cdot \left(c\sqrt{\delta \cdot 2^i}\right)^2\right) = O(H \cdot 2^H c^2 \delta) = O(N \log N)\ .$$

A smaller choice of α in $f(x)$, allows the AAC build to run in linear time. In particular for $f(x) = c \cdot x^{0.5-\epsilon}$, $\epsilon > 0$, the work per level geometrically decreases going up the tree, and the time complexity is:

$$O\left(\sum_{i=0}^{H} 2^{H-i} \cdot c^2 \left(\delta \cdot 2^i\right)^{1-2\epsilon}\right) = O\left(\sum_{i=0}^{H} 2^{H-2\epsilon i}\right) = O(N)\ .$$

Thus, for the balanced case, as long as the number of clusters returned at each node is smaller than the square root of the number of primitives contained in the subtree, the algorithm takes linear time.

A completely imbalanced constraint tree will have depth $N - \delta$ and the size of each node is $O(f(N))$. Therefore assuming that $f(x) = c \cdot x^{\alpha}$ the AAC build will have $O\left(\min\left(N^2, N^{1+2\alpha}\right)\right)$ time complexity. In practice this worst case cannot occur, at least for fixed or floating-point representations of primitive coordinates, since such numbers do not have the precision to represent such an unbalanced tree.

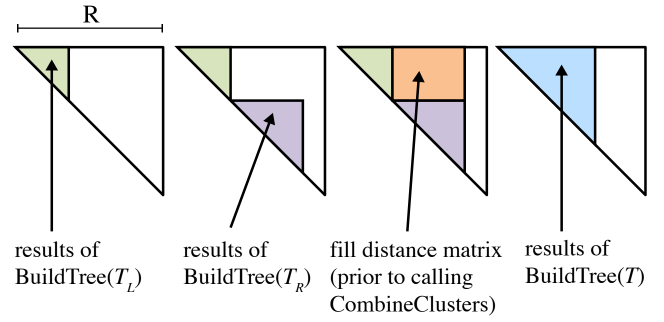


Figure 2: *Careful data arrangement allows reuse of cluster distance computations from child nodes. Four steps in computing clusters for a subtree T are shown. Distance results computed during processing of the left subtree (green) and right subtree (purple) are already stored in the correct location in the distance matrix for T.*

In practice, we choose $c = \frac{\delta^{0.5+\epsilon}}{2}$, which indicates that $f(\delta) = \frac{\delta}{2}$, so that AAC algorithm halves the number of clusters on leaf nodes.

3.3 Optimizations

Although the AAC build algorithm is conceptually quite simple, a number of important optimizations to `CombineClusters` were required to achieve the high performance reported in Section 4.2.

3.3.1 Construction Speed

Reduce redundant computation of cluster distances. The cluster distance function $d(C_i, C_j)$ is an expensive function that requires loading two cluster bounding boxes from memory, then calculating an aggregate bounding box and its surface area. Thus, at the start of `CombineClusters` we pre-compute all possible $d(C_i, C_j)$ for the clusters in C and save these results in a matrix. This matrix is symmetric, so only its upper-triangle need be calculated and stored. Moreover, each clustering step requires removal of two clusters from the set C (line 13). Our implementation replaces one deleted cluster with the newly formed cluster c', and the other deleted cluster with the last cluster in the set. Thus the precomputed matrix remains compact without much data movement, and its size decreases by one each iteration.

A second optimization is that cluster distance values available at the termination of `CombineClusters(C,n)` are values that are also needed during clustering at the next level up the constraint tree. These distances are retained and reused, as described further in the next optimization.

Reducing data movement. The storage required for the distance matrix computed by `CombineClusters` is greater at higher levels of the constraint tree (size of the set C is larger). Careful management of this temporary storage can eliminate the need for dynamic allocation during the build and also enable reuse of distances computed during child processing.

As illustrated in Figure 2, suppose we have a pre-allocated a distance matrix with R rows and columns and seek to process subtree T. Processing the left subtree T_L can be carried out using a fraction of the allocation shown in green. (Specifically, the final distance matrix for clusters returned from T_L is stored at this location.) Then T_R is processed using an adjacent storage region, producing the distance matrix shown in purple. The computation of

distances by node T can reuse these values. Only the new distances, shown in orange, must be computed before clustering is performed by `CombineClusters` for node T. The final distance matrix for the node T is given in blue.

On average, for n tree nodes, the size R of the pre-allocated matrix for the average case should be:

$$R = \sum_{i=1}^{\log(n/\delta)} f\left(\frac{n}{2^i}\right) \leq f\left(\frac{n}{2}\right) \cdot \frac{1}{1 - 0.5^{0.5-\epsilon}}.$$

Our implementation sets $R = 4c \cdot n^{0.5-\epsilon/2}$, resulting in an allocation that is sublinear in the total number of primitives n. Although it is possible to create a scene for which construction requires more temporary storage than this bound (requiring dynamic reallocation), we have not encountered this situation in practice.

3.3.2 Subtree Flatting for Improved BVH Quality

After each cluster merge, the AAC algorithm performs a check to determine whether to maintain the BVH subtree corresponding to this cluster, or to flatten the subtree into a list of primitives. The optimization of electing not to build a subtree structure is a natural and standard optimization in top-down SAH builds, but an agglomerative build must detect and "undo" the subtree it has already constructed when a flat list of primitives will reduce expected ray tracing cost. To detect this condition, we employ a cost metric is similar to the SAH cost metric, however since the flattening check proceeds bottom-up the true cost of the subtrees is known. (No linear approximation of subtree cost is required.) The cost $\mathcal{C}(T)$ of BVH subtree T is:

$$\mathcal{C}(T) = \min \begin{cases} \mathcal{C}_t \cdot |T| \\ \frac{S(T_l)}{S(T)}\left(\mathcal{C}_b + \mathcal{C}(T_l)\right) + \frac{S(T_r)}{S(T)}\left(\mathcal{C}_b + \mathcal{C}(T_r)\right) \end{cases}$$

where $|T|$ is the number of primitives in T, T_l and T_r are the left and right subtrees of T, $S(T)$ is the surface area of the bounding box of T, and $\mathcal{C}_b$ and $\mathcal{C}_t$ are estimated ray-box and ray-triangle intersection costs. Nodes are flattened if the above metric indicates cost is minimized by doing so. This process is similar in spirit to the subtree flattening performed after a top-down SAH build by [Bittner et al. 2013]. We observe that tree flattening modestly improves BVH quality (reduces intersection and traversal computations during ray tracing) by 3-4%.

3.4 Parallelism

The AAC algorithm is amenable to parallel implementation on a multi-core CPU. As with SAH-based top-down builds, the algorithm's divide-and-conquer structure creates many independent subtasks corresponding to subtrees. These subtasks can be processed by different cores in parallel.

The lack of parallelism at the top of the constraint tree is less problematic for an AAC build than a traditional top-down SAH build, since the work performed in the upper nodes of the tree is substantially less. For example, for the 7.9 M triangle San Miguel scene (see Table 2), the topmost clustering step must only process 1,724 clusters. As a result, we do not attempt any intra-node parallelization of the algorithm and still observe good speedup on 32 cores (see Section 4.2). We note that multi-core parallelization of the clustering process within each node would incur similar complexities as those encountered in [Walter et al. 2008]. However, the brute-force implementation of `CombineClusters` is amenable to vector parallelization (we have not attempted this optimization).

During top-down bisection, our implementation creates approximately eight times as many parallel subtasks as cores. Subtrees associated with each task are evaluated in serial on a core, using all the storage and redundant-computation elimination optimizations described in Section 3.3. Note these optimizations induce a serial ordering on the processing of subtrees, so they cannot be employed for subtrees executed in parallel.

3.4.1 Radix Sort

Our AAC implementation employs a parallel radix sort that sorts $\frac{\log n}{2}$ bits per pass, where n is the number of keys being sorted. Since the number of bits we use for the Morton code is $O(\log n)$, the number of passes is a constant, and the total work of the sort is linear in N. The sort is based on the integer sort found in the Problem Based Benchmark Suite [Shun et al. 2012]. The algorithm partitions the array of size n into $\sqrt{n}$ blocks of size $\sqrt{n}$. Within each block we maintain $\sqrt{n}$ buckets and count the number of entries of each possible value of the $\frac{\log n}{2}$ bits being sorted. The blocks can be processed in parallel since the buckets are maintained separately. The result of the counting phase can be viewed as an $\sqrt{n} \times \sqrt{n}$ array of counts with the buckets across the rows and processors down the columns. By flattening this array in column-major order, then performing a parallel plus-scan on the result, we get the offset location for the appropriate bucket for each processor. A final pass over the data array can be used to permute the data to its sorted location. This is again parallel over the blocks.

4 Evaluation

We evaluated AAC BVH construction by comparing its performance and output BVH quality against that of three alternate BVH build implementations (1) a top-down full-sweep SAH build (SAH), (2) a top-down "binned" SAH build (SAH-BIN) that evaluates SAH-cost at the boundaries of 16 bins along the longest axis, and (3) locally-ordered clustering (Local-Ord) as described in [Walter et al. 2008]. Since implementation and machine differences can make it difficult to compare performance (and even tree-quality) measurements against those reported in previous publications, we developed our own implementations of each algorithm. Our implementations have undergone substantial optimization (with one exception: our implementations of the algorithms do not use SIMD intrinsics). Still, Table 2 indicates that SAH is over two times faster than that published in [Wald 2007], SAH-BIN is within a factor of 1.7 (adding SIMD execution would likely more than make up this difference), and Local-Ord is at least two times faster than the implementation in [Walter et al. 2008]. Our experiments include two configurations of AAC construction: AAC-HQ is configured conservatively to build a high-quality BVH ($\delta = 20$, $\epsilon = 0.1$). AAC-Fast is configured for high performance ($\delta = 4$, $\epsilon = 0.2$).

We executed all five build algorithms on the six scenes shown above Table 2. (Half-Life is a scene containing geometry exported from Half-Life 2: Episode 2.) Renderings used 16 diffuse bounce rays per pixel and four-to-five point light sources depending on scene. All experiments were run on a 40-core machine with four 10-core Intel E7-8870 Xeon processors (1066 MHz bus). Parallel implementations were compiled with CilkPlus, which is included in `g++`.

4.1 BVH Quality

Figure 3 compares the cost of tracing rays through the BVHs produced by each algorithm, normalized to that of SAH-BIN. The sum of node traversal steps (bottom part of bar) and intersection tests (top of bar in lighter color) is used as a simple proxy for tracing cost. (We find tracing wall clock times, despite their dependence

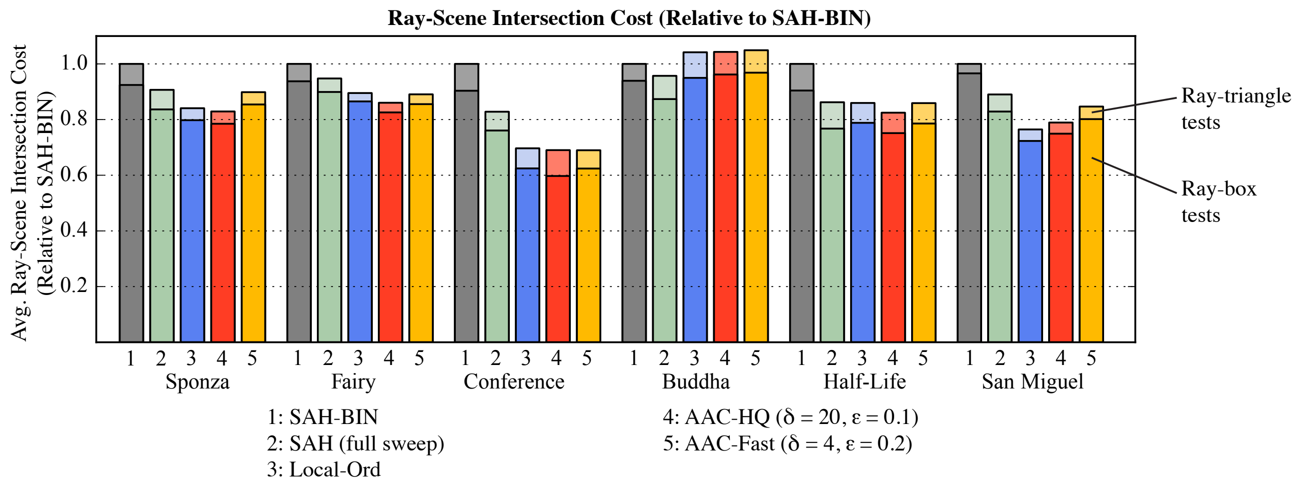


Figure 3: *Cost of ray-scene intersection (measured as the sum of BVH node traversals and triangle intersection tests). Values are averaged over all rays (primary, shadow, and gather) and normalized to the cost of traversal through the BVH constructed using SAH-BIN. In all scenes but Buddha, both the AAC-HQ and AAC-Fast configurations yield as good (or better) trees than a full sweep SAH build. The benefits of bottom-up agglomerative construction diminish when all scene triangles are similarly sized, as is the case in a model like Buddha.*

	SAH	Local-Ord	AAC-HQ	AAC-Fast
Sponza	.89 / .91	.86 / .84	.86 / .82	.90 / .90
Fairy	.97 / .94	.97 / .88	.95 / .84	.94 / .88
Conference	.86 / .82	.76 / .68	.79 / .67	.78 / .67
Buddha	.98 / .95	1.08 / 1.03	1.09 / 1.03	1.07 / 1.04
Half-Life	.81 / .87	.86 / .86	.80 / .83	.82 / .87
San-Miguel	.88 / .89	.82 / .75	.83 / .78	.90 / .84

Table 1: *Normalized ray-scene intersection costs for diffuse bounce (left value) and shadow (right value) rays. Costs are normalized to that of SAH-BIN. In general, the relative improvement of the agglomerative techniques is higher for shadow rays since large occluders are lifted to higher levels of the BVH. Primary ray results are similar to diffuse bounce rays and are not shown.*

on the ray tracer used, track these counts well.) Values in the figure are the result of averaging counts over primary, shadow, and diffuse bounce rays. More detailed individual statistics for shadow and diffuse bounce rays are given in Table 1.

Figure 3 shows that the approximate agglomerative clustering performed by the AAC build algorithm does not significantly impact the quality of the resulting BVH. AAC-HQ produces BVHs that have essentially the same quality as those produced by Local-Ord (in the worst case of San Miguel, AAC-HQ and Local-Ord costs differ by only 3%). Tracing cost ranges from 15% to 30% less than that of the SAH-BIN built BVH, and it remains lower than the full SAH build for all scenes but Buddha. More surprisingly, AAC-Fast also produces BVHs that are as good or better than the full sweep build in these cases.

We also observed that BVHs produced by the agglomerative methods realize the greater benefit for shadow rays than radiance rays, particularly when scenes contain primitives with large spatial extent (Table 1). This is because the agglomerative approaches are more likely to place large primitives higher in the tree. Thus large primitives, which are likely to be frequent occluders, are encountered quickly during traversal, leading to early ray termination.

All three agglomerative techniques (not only the AAC configurations) generate a lower quality BVH than the top-down SAH method for Buddha. The benefit of bottom-up methods is greatest when variability in scene triangle size is large, a condition that is

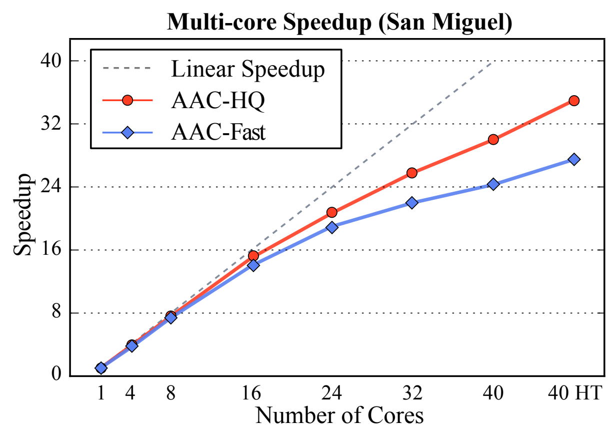
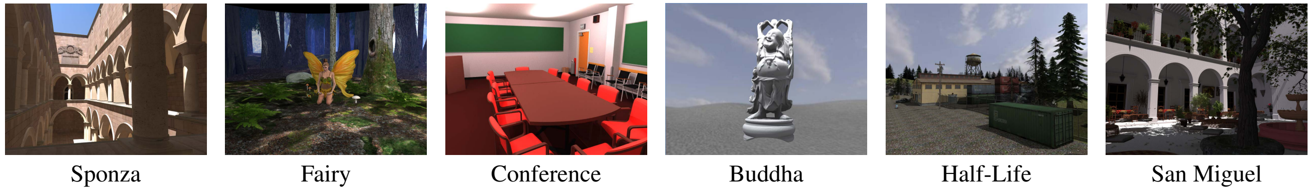


Figure 4: *AAC-HQ BVH construction achieves near linear scaling out to 16 cores and 35× speedup when executed using 80 threads on 40 cores with Hyper-Threading (40 HT). AAC-Fast achieves 27× speedup in the 40 HT configuration.*

not present in Buddha. We observe similar results (BVH quality of agglomerative clustering equal to or slightly worse than SAH) for scenes consisting of single high-resolution scanned models (e.g., the Stanford Dragon) or single hairball models. One technique for improving top-down SAH build quality in scenes with large variation in triangle size is to pre-split large triangles [Ernst and Greiner 2007]. Agglomerative methods handle this case well without introducing many new triangles into the scene.

4.2 BVH Construction Performance

Table 2 reports BVH build times for each of the five methods. Timings include all aspects of BVH construction, including Morton code sort in the AAC configurations. We find that the performance of a single-core AAC-HQ build is similar to that of SAH-BIN and approximately five to six times faster than Local-Ord (while producing BVHs of similar quality). AAC-Fast (which still generates higher quality trees than both SAH-BIN and SAH in all cases but Buddha) performs even less work, resulting in a build that is up to 4× faster than SAH-BIN.



Sponza | Fairy | Conference | Buddha | Half-Life | San Miguel

	Num Tri	SAH (1 core exec. time (ms))	SAH-BIN (1 core exec. time (ms))	Local-Ord (1 core exec. time (ms))	AAC-HQ 1 core (ms)	AAC-HQ 32 cores (ms)		AAC-Fast 1 core (ms)	AAC-Fast 32 cores (ms)	
Sponza	67 K	115	48	214	52	2	(24.0×)	20	1	(21.5×)
Fairy	174 K	364	148	1,004	117	5	(24.5×)	44	2	(22.4×)
Conference	283 K	560	228	1,279	225	10	(23.6×)	70	4	(19.4×)
Buddha	1.1 M	3,016	932	5,570	1,101	43	(25.8×)	397	16	(24.0×)
Half-Life	1.2 M	3,190	1,110	5,978	1,080	42	(25.7×)	359	15	(22.8×)
San Miguel	7.9 M	27,800	8,430	43,659	7,350	298	(24.6×)	2,140	99	(21.6×)

Table 2: *Single core BVH construction times for all build algorithms (+ 32-core timings for the AAC configurations). The AAC-Fast build is over three times faster than SAH-BIN while also producing a higher-quality BVH in all scenes except Buddha.*

Execution Time Breakdown: San Miguel

	AAC-HQ 1 core (ms)		AAC-HQ 32 cores (ms)		AAC-HQ Speedup	AAC-Fast 1 core (ms)		AAC-Fast 32 cores (ms)		AAC-Fast Speedup
Morton Code	351	(5%)	21	(7%)	16.8×	351	(16%)	21	(21%)	16.8×
Agglomeration	7,000	(95%)	277	(93%)	25.1×	1,789	(84%)	78	(79%)	23.0×
Total	7,350		298		24.6×	2,140		99		21.6×

Table 3: *Execution time breakdown for 1- and 32-core executions of AAC-HQ and AAC-Fast on San Miguel. The Morton code sort constitutes only a small fraction of total execution time in AAC-HQ (<7%), but a notable fraction in AAC-Fast (up to 21%). The sort consumes a larger fraction of total build time in the parallel executions since it parallelizes less efficiently than the agglomeration phase of the AAC algorithm.*

We also find that the AAC methods scale well to high core count (Figure 4). For the San Miguel scene AAC-HQ achieves nearly linear speedup out to 16 cores, and a 34× speedup on 40 cores (using 80 threads with Hyper-Threading enabled, see "40 HT" data point in Figure 4) . AAC-Fast also scales well out to 16 cores, but achieves a more modest 24× speedup on average on 40 cores.

Table 3 breaks down the execution time of 1- and 32-core configurations of AAC-HQ and AAC-Fast for San Miguel. The parallel Morton code sort occupies only 5% of total execution in the single core AAC-HQ configuration. This percentage increases to 7% in the 32-core configuration since the agglomeration phase of the algorithm parallelizes more efficiently. (The Morton code sort is bandwidth bound.) The Morton code sort consumes a larger fraction of AAC-Fast execution time (16% and 21% for the 1- and 32-core executions) since the agglomeration phase in AAC-Fast performs less work. This is a major reason why AAC-Fast achieves a lower overall parallel speedup than AAC-HQ.

5 Discussion

In this paper we demonstrated that bottom-up BVH construction based on agglomerative clustering can produce high-quality BVHs with lower construction cost than top-down SAH-based approaches. Our algorithm features abundant coarse-grained parallelism, making it easy to scale on many-core CPU architectures. While we have evaluated its utility in the context of ray tracing, we believe it will find use in other areas of computer graphics where high-quality object hierarchies are desired.

We have not yet investigated the possibility of a "lazy" variant of an AAC build. While lack of support for lazy builds is a known drawback of previous bottom-up approaches, it may be possible to leverage the top-down bisection process of the AAC build to achieve "lazy" behavior. Last, while our efforts in this paper focused on the implementation of the technique on many-core CPUs, an obvious area of future work is to consider if additional parallelism can be exploited to enable efficient implementation on a GPU.

6 Acknowledgments

This work is partially supported by the National Science Foundation under grant CCF-1018188, by the Intel Labs Academic Research Office under the Parallel Algorithms for Non-Numeric Computing Program, and by the NVIDIA Corporation.

References

Bern, M., Eppstein, D., and Teng, S.-H. 1999. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications 9*, 6, 517–532.

Bittner, J., Hapala, M., and Havran, V. 2013. Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1, 85–100.

Ernst, M., and Greiner, G. 2007. Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, IEEE Computer Society, RT '07, 73–78.

Gargantini, I. 1982. An effective way to represent quadtrees. *Communications of the ACM 25*, 12, 905–910.

GOLDSMITH, J., AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May), 14–20.

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and K-D trees. In *Proceedings of the Conference on High Performance Graphics*, Eurographics Association, HPG'12, 33–37.

LAUTERBACH, C., GARL, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *In Proc. Eurographics '09*.

OLSON, C. F. 1995. Parallel algorithms for hierarchical clustering. *Parallel Computing 21*, 1313–1325.

SHUN, J., BLELLOCH, G. E., FINEMAN, J. T., GIBBONS, P. B., KYROLA, A., SIMHADRI, H. V., AND TANGWONGSAN, K. 2012. Brief announcement: the Problem Based Benchmark Suite. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the art in ray tracing animated scenes. In *Computer Graphics Forum*.

WALD, I. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 33–40.

WALD, I. 2012. Fast construction of SAH BVHs on the Intel Many Integrated Core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 1 (Jan.), 47–57.

WALTER, B., BALA, K., KULKARNI, M., AND PINGALI, K. 2008. Fast agglomerative clustering for rendering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, 81–86.