# Sequential and Parallel Construction of Bounding Volume Hierarchies

Hoang Duong, Amir Sabbag

June 6, 2026

## 1 Introduction

Bounding Volume Hierarchies (BVH) are acceleration data structures that are widely used in Physically Based Ray Tracing (PBRT) and physics simulations. The BVH tree is constructed from a given number of objects that can be bounded by Axis-Aligned Bounding Boxes (AABB). By organizing geometry into a tree of nested bounding boxes, a BVH allows one to quickly traverse through the objects, for example during ray tracing where one has to compute millions of intersections.

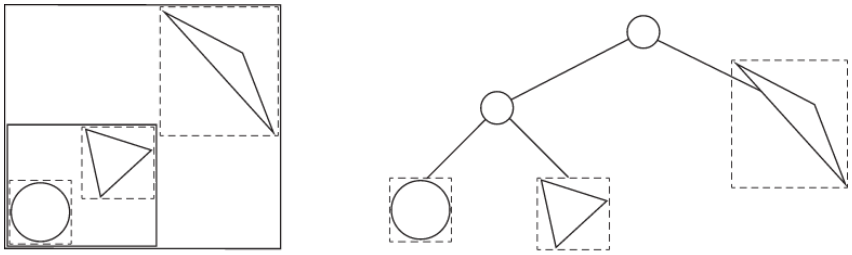



Figure 1: BVH for a simple mesh. (a) The primitives in rendering space with their bounding boxes, with the small triangle and circle primitive bounded and linked into a node, and all there linked into a larger node. (b) BVH tree from the scene, with root node holding the whole mesh, left children another node, right children the large triangle. Source

These data structures critical to PBRT, and have to be highly optimized for real-time rendering. The two important aspects to optimize for in BVHs are build-time and build-quality. The time to build the BVH matters for scenes or objects that are dynamic, i.e. a mesh that changes massively between each frame, for example in video games where scenes have to be renderer in real-time or animations. Build quality on the other hand effects the traversal time, with better constructed trees resulting in lower traversal times.

This leads to a complicated game of balancing the acceptable build time to traversing time ratio. In this report we will go over different BVH implementations, mainly sequential BVH with centroid sorting, sequential Surface Area Heuristic (SAH) BVH, naive BVH parallelisation by top down tree construction, and Linear BVH (LBVH) which changes the tree structure to allow massive parallelisation, increasing occupancy.

Our BVH implementations will focus purely on PBRT, and we will be testing them using benchmarks and scene rendering. The rendering code is provided as a boilerplate, and supports spheres, quads and triangle meshes, indirect and direct lighting, and anti-aliasing. The languages we chose to implement the BVHs in were C++ and Rust, both being highly performative and having strong parallelisation.

The two BVH implementations were divided between the group, with specifically Hoang implementing and benchmarking the top-down parallel BVH construction in C++ (Section 4), and Amir implementing and evaluating for the Linear BVH algorithm in Rust (Section 5). The comparison of the two approaches and the conclusions are at Section 7.3.

# 2 How Physically Based Ray Tracing works



Figure 2: The `Maria_C` mesh, the render goal of the project

One of the main applications of BVHs is physically based ray tracing (PBRT), a rendering technique used in computer graphics. PBRT is commonly used in applications such as animations, offline rendering, and modern video games.

The basic idea is simple. A camera launches rays into a 3D scene containing many objects. For each ray, we check which object it intersects first and use that information to determine the final pixel color. Since a rendered image may require millions of rays, these intersection tests become one of the most expensive parts of the rendering process.

Many scenes contain triangle meshes, which are objects made from large collections of triangles. Complex meshes can contain hundreds of thousands or even millions of triangles. A naive approach would test every ray against every triangle, resulting in a large amount of computation.

BVHs were created to reduce the cost of these intersection tests and make ray tracing practical for large scenes.

## 2.1 Motivation for BVH

A ray might pass through empty space or only intersect a small subset of nearby geometry. A Bounding Volume Hierarchy (BVH) exploits this by recursively grouping the scene into a tree of bounding volumes. Instead of testing a ray against every object, we first test it against these bounding volumes. If the ray misses a volume, all objects inside it can be skipped, avoiding expensive per-primitive intersection tests.

This hierarchical rejection significantly reduces the number of intersection tests compared to a naive $O(n)$ approach per ray, making ray tracing practical for complex scenes. BVHs are therefore called acceleration structures, since they speed up ray tracing by reducing unnecessary

computations. In this context, a Bottom Level Acceleration Structure (BLAS) stores the primitives of a single object, such as a triangle mesh, and a BVH is built over them. A Top Level Acceleration Structure (TLAS) is then a BVH built over these BLAS objects, forming the full scene hierarchy.

### 2.2 Axis Aligned Bounding Boxes

An Axis-Aligned Bounding Box (AABB) is the simplest and most widely used bounding volume in ray tracing acceleration structures. As the name suggests, an AABB is a rectangular box whose edges are parallel to the coordinate axes of the scene (typically the $x$, $y$, and $z$ axes). Any AABB can be defined by just two 3D points: a minimum corner and a maximum corner.

## 3 Sequential BVH construction

The simplest method of constructing a BVH is a recursive top down algorithm. Starting from the full set of primitives, we compute a bounding box that encloses every object. This box becomes the root node of the tree. The primitives are then partitioned into two subsets, and the same process is recursively repeated until a sufficiently small number of primitives remain in each leaf. Ideally, the partition should generate a tree as balanced as possible, allowing larger coverage and lower depth traversal for queries.

In our naive implementation, we used centroid splitting along the maximal axis. For every primitive, usually triangles in a mesh, we first compute the centroid of its bounding box. We then determine which axis of the current node has the largest coverage. The primitives are sorted using their centroids along this axis, and the list is split at the median, which produces two child nodes containing approximately equal numbers of primitives. The complexity of the build is roughly $O(n \log n)$.

The advantage of this method is its simplicity and relatively fast build time. Sorting primitives by centroid generally produces spatially coherent partitions, which improves traversal performance and node region representation. The recursive subdivision also creates a somewhat balanced tree, giving traversal complexity close to $O(\log n)$ in best cases.

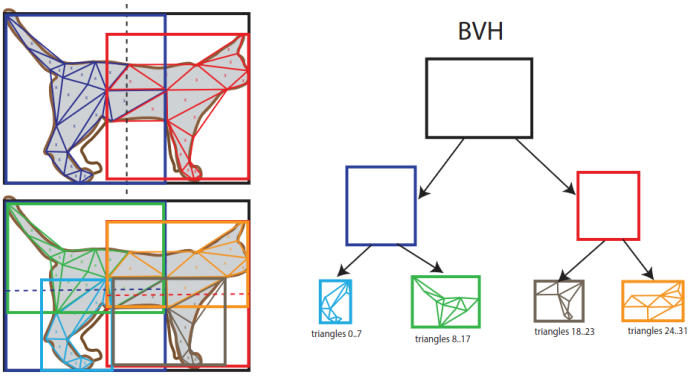



Figure 3: Recursively computing bounding boxes of BVH nodes. The black bounding box is split into 2 groups of primitives (blue and red) and with their own bounding boxes (which may overlap). Then each node is again divided into 2 new groups (cyan and green, and orange and grey), and keeps going, until the leaves, which contain consecutive indices of primitives. Source: BX CSE306 Computer Graphics

One weakness of centroid splitting is that it does not directly optimize traversal cost. A bad split may still occur if primitives overlap heavily or if the geometry distribution is uneven. This can lead to larger bounding volumes and more ray intersection tests during traversal. In

practice, however, this implementation already provides a significant improvement over brute force intersection.

### 3.1 Surface Area Heuristic

While centroid splitting produces BVHs quickly, it does not explicitly optimize traversal cost. A more sophisticated method is the Surface Area Heuristic (SAH), which estimates how expensive a traversal cost will be for a split based on the surface areas of the child nodes and the number of primitives they contain.

The SAH cost can be approximated as

$$C_{SAH} = \frac{A_L}{A_P} N_L + \frac{A_R}{A_P} N_R$$

where $A_P$ is the surface area of the parent node, $A_L$ and $A_R$ are the surface areas of the child nodes (we used their bounding box's area), and $N_L$ and $N_R$ are the number of primitives inside them. Lower SAH costs generally correspond to better traversal performance due to reduced node overlap and fewer intersection tests.

Evaluating every possible split directly is computationally expensive, so we implemented a binned approximation of SAH. Primitives are grouped into a fixed number of spatial bins along each axis, and candidate splits are evaluated only at bin boundaries. In our implementation we used 16 bins, which provided a good balance between construction time and traversal quality without significant memory overhead. With this, the time complexity only adds up by a constant amount of bin number, in this case 16.

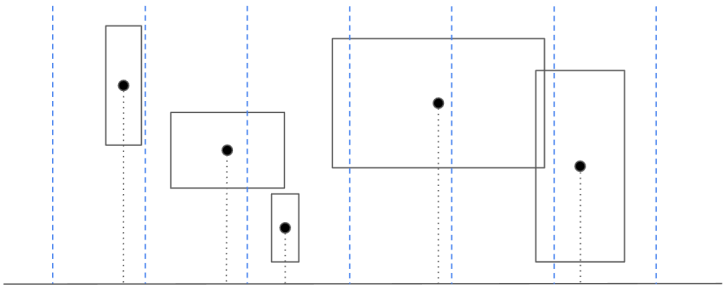


Figure 4: SAH bins and choosing a Splitting Plane for BVH. Each primitive are placed in a bin along the axis based on the centroid of its bounds. The algorithm then estimates the cost for splitting the primitives using the planes at each of the boundaries, and select the one giving the minimum cost. Source

## 4 Top-down Parallel BVH construction

The recursive structure of top-down BVH construction naturally exposes task parallelism, since after a node is partitioned, the left and right subtrees can be constructed independently. However, naively spawning tasks at every recursive level introduces significant scheduling overhead and poor balancing for uneven partitions.

To address this, we implemented a hierarchical thread budget model. The root node begins with a fixed thread budget specified by the user. After each split, the available threads are distributed between the child nodes proportionally to the number of primitives they contain. For example, a balanced split receive approximately equal thread allocations, while uneven splits allocate more threads to larger subtrees. As recursion continues deeper into the tree, thread budgets gradually shrink until construction becomes fully sequential.

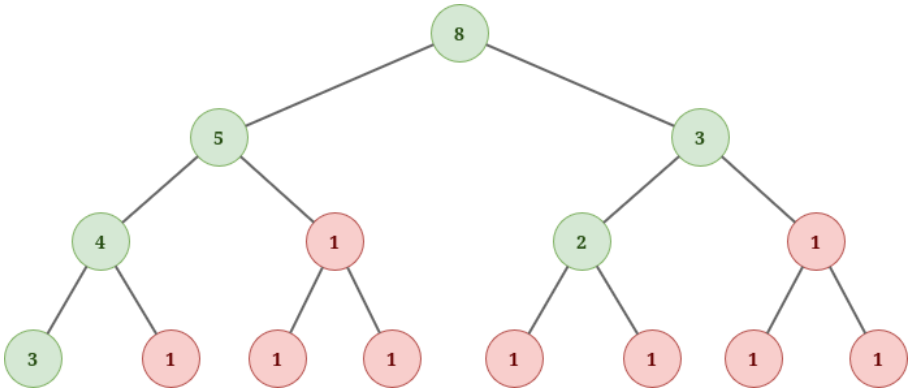



Figure 5: Thread allocation for a BVH build process. Green = parallelizable, red = pure sequential. Root starts with 8 thread (user-defined), then recursively allocate threads to its children based on their sizes. Once a node reaches one allocated thread, the full subtree is run sequentially

An important observation is that different BVH levels benefit from different forms of parallelism. Upper levels contain few nodes but many primitives, so we parallelize internal operations such as bounding box computation, SAH binning, and primitive partitioning using reductions, map-reduce accumulation, and prefix-sum-based scattering. Deeper levels contain smaller primitive sets but many independent subtrees, making recursive subtree construction the more effective source of parallelism. The algorithm at the same time adapts a flexible strategy in which upper levels primarily exploit data parallelism within node construction, while lower levels exploit task parallelism across independent subtree builds, allowing even a simple thread budget allocation scheme to achieve efficient parallel execution throughout the build process.

However, one drawback of this strategy is that threads assigned to a subtree are never reintegrated into the construction of other subtrees once that work completes. As a result, the effective thread count can gradually diminish throughout the build process, especially for large scenes with highly unbalanced, scattered model workloads. A possible improvement would be to use a task queue or work-stealing mechanism to redistribute threads and better utilize available parallelism. This section will consider only the parallelization of a pure top-down BVH construction algorithm.

**Technical details** This subsystem was implemented in C++23, which served as the default language baseline for the top down component of the project. Most of the parallelism was built using standard library concurrency primitives: `std::thread` for worker management, `std::atomic<int>` for tracking global allocations, and thread joining via `.join()` to handle execution barriers.

To eliminate global heap allocation bottlenecks, we pre-allocate a single, contiguous flat array before construction begins. This array is sized to hold the maximum theoretical node count ($2N - 1$, where $N$ is the number of primitives). When a thread splits a node, it reserves slots for the child nodes by safely incrementing a global `std::atomic<int>` index counter, keeping node allocation lock-free. Some other large arrays were also preallocated to prevent large array allocations and contentions between multiple threads.

The technical details of data-parallel algorithms implemented to handle the internal operations at these upper levels are detailed below.

## 4.1 Parallel SAH Binning via Map-Reduce

Evaluating the Surface Area Heuristic (SAH) at upper tree levels requires binning primitives without introducing synchronization bottlenecks. A naive approach, where multiple threads concurrently update a single global array of bins, would cause severe thrashing and heavy thread

contention. To avoid this, we use a Map-Reduce strategy.

During the **Map Phase**, the primitive list of the current node is divided into equal chunks and distributed among the threads. Each thread maintains its own private, isolated bin array in thread-local storage. As a thread loops through its assigned primitives, it computes the target bin index based on the current primitive and updates its local bin bounding boxes and counts. Because this memory is entirely isolated to each thread, this heavy processing phase runs with zero synchronization overhead and minimal memory overhead. The complexity here is $O$(chunk size * axis count).

At the **Reduce Phase**, these independent thread-local arrays are merged into a single global bin array to find the optimal split plane. Because the number of bins is typically quite small (16 bins) a single-threaded sequential reduction across the thread pool's local arrays is fast enough. This clean separation eliminates lock contention entirely and lets the threads maximize CPU usage during the sweep. Complexity $O$(thread count * axis count * bin count).

### 4.2 Parallel Partitioning via Prefix Sum and Scatter

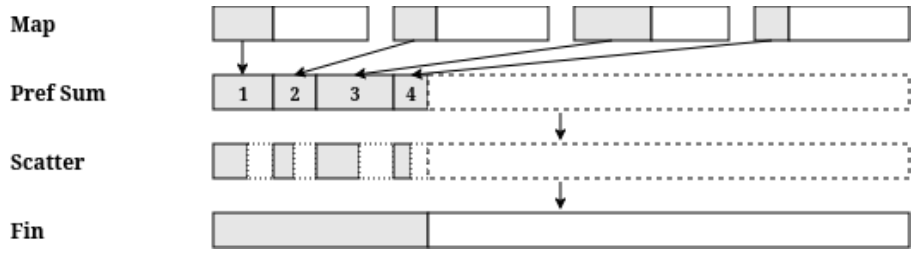



Figure 6: Parallel Partitioning algorithm in 3 steps: Mapping and flagging left elements (grey); Calculating start offset of each left blocks with prefix sum; Scattering the elements to their respective blocks. Algorithm should also calculate offset and scatter for right (white) chunks

Once the SAH evaluation determines the optimal split, the primitives of the node must be reordered into left and right of the split index, and thus the left and right child node. Doing this concurrently is challenging because individual threads do not know where their primitives belong in the final global array. We solve this with a multi-pass, lock-free partitioning pipeline that combines a parallel prefix sum (scan) and a scatter phase to put them into their respective index. This implementation actually closely resembles how the C++ standard library policy std::execution::par handles parallel partitioning.

The pipeline executes in three distinct steps:

1. **The Count Phase:** Worker threads are spawned to process their local chunks of primitives against the chosen split axis. Each thread counts exactly how many of its primitives belong to the left child and how many belong to the right, flagging these elements in a small shared array. The complexity is $O$(chunk size).

2. **The Parallel Prefix Sum Phase:** The main thread waits for all counting to finish, then run prefix sum calculation over the shared count array. This operation calculates the unique, global memory offsets, mapping out the exact, non-overlapping destination index ranges for each thread's chunk of primitives. The complexity is $O$(thread count).

3. **The Parallel Scatter Phase:** After finishing the prefix sum, the parallel worker threads are spawned to read their input chunks. Using with their guaranteed unique destination offsets, the threads "scatter" (write) the primitives directly into the destination buffer. A parallel array copy operation at the end is performed to modify the original primitive array. Complexity $O$(chunk size).

Because the prefix sum guarantees that every single thread writes to a completely disjoint segment of memory, this final scatter phase is 100% lock-free and takes full advantage of CPU bandwidth.

### 4.3 Tree flattening

We store the nodes of our tree in a contiguous vector, which is much better for cache locality compared to a tree with linked nodes, since neighbouring nodes are likely to reside in the same cache line and can be fetched together. When building sequentially we do so in a depth first manner, resulting in a nodes vector that is already flattened. Consequently, the left child of an internal node is located immediately after its parent (at index $i + 1$), while the right child is located at index $i + s$, where $s$ is the size of the left subtree.

We could further improve cache locality by instead using an ordered depth first traversal, where we place the node that has the highest surface area (thus a higher probability that a ray traverses it) next to the parent node. This prevents cache misses and memory accesses to a distant part of the tree. We did not implement this feature.

However, this approach is not possible in a parallel top-down construction, since the final ordering of nodes in memory is not known until all subtrees have been constructed. When using an atomic counter to insert into the nodes, the resulting tree layout will not be flattened, and parent and child nodes could be separated. As we increased the number of threads, we saw that this scattering increased, resulting in worse cache-locality and increased traversal times. Note that traversal itself is still done sequentially. The number of build threads only affects the memory layout of the resulting BVH. More concurrent node allocations lead to more fragmentation, causing additional cache misses during traversal.

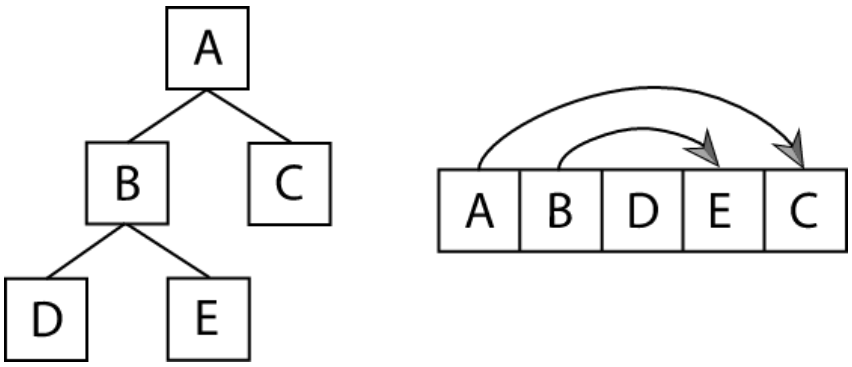



Figure 7: A tree flattening process that takes in a non-contigous tree and flattens in to a vector using depth first traversal. Source

One could also assume that each subtree has at most $2 * N - 1$ size, where $N$ is the number of triangles in that subtree. This allows child nodes to be placed at predetermined offsets relative to their parent, preserving a logical depth-first layout even during parallel construction. In practice, many subtrees will have much less than the maximum possible number of children, causing many gaps in the tree. These gaps increase memory consumption, reduce cache efficiency, and negate some of the benefits of storing the BVH in a compact contiguous representation.

What we instead do in our parallel building is constructing an intermediate tree first, and executing a flattening pass afterwards. During this pass, we do a depth-first traversal, writing the nodes to a now gapless and contiguous vector. This separated flattening pass allows for a natural entry point into different optimization algorithms, such as the ordered depth-first traversal we mentioned before. This flattening leads to better traversal times, although at the cost of flattening the tree.

# 5 Linear BVH construction

A major shortcoming of the sequential and parallel BVH builds is that they construct levels of the tree sequentially, which limits the amount of parallelism that they can achieve. Even with our parallel approach, we do not know the final locations of all the nodes, forcing us to travel through the tree top-down. In his 2012 paper, Tero Karras introduced an approach that allows constructing the entire tree in parallel. This process first encodes the centroids of primitives to a single integer using Morton encoding, sorts them, and then constructs the tree topology bottom up.

## 5.1 Morton encoding

Morton codes, also known as Z-order curves, map multidimensional data to one dimension while preserving spatial locality. One can use Morton codes to encode a three-dimensional position to an integer by interleaving the bits of its normalized coordinates. Points that are close in space tend to receive similar Morton codes, which naturally leads to a useful grouping of nearby primitives into the same subtree.

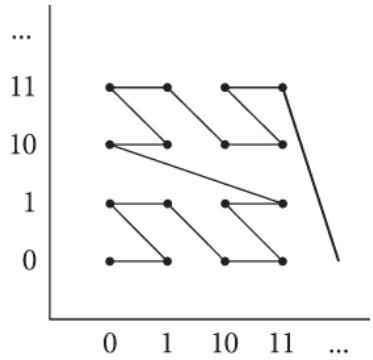



Figure 8: Morton encoding creates a natural traversal, mapping nearby codes close. Source

For a point in space, we have the $x$, $y$, and $z$ axes, so we will encode three integers into one Morton code. Our implementation provides an encoder for 3x10-bit into a 32-bit integer, and an encoder for 3x21-bit into 64-bit integer. The 3x10-bit variant allocates 10 bits to each spatial dimension, while the 3x21-bit variant allocates 21 bits per dimension. In our LBVH implementation, we only use the 3x10-bit variant, since they allow for huge speed-ups in Radix sorting whilst not sacrificing too much precision.

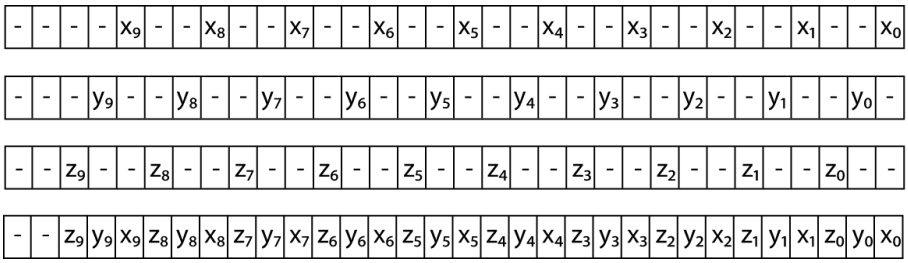



Figure 9: Three integers are encoded into a final Morton code by interleaving their bits. Source

Since we only care about the relative positions of every primitive with respect to each other and not their global positioning, we compute the bounding box that encompasses all the primitives, and normalize with respect to these coordinates. Afterwards, for each primitive, we compute its axis-aligned bounding box and use its centroid as the representative point. We normalize the center to lie in the interval $[0, 1]$ for each of its axes.

## 5.2 Sorting the morton codes

Once each primitive has been assigned a Morton code, the primitives are sorted by that key. The sorting step is central to LBVH construction because the resulting order determines the tree topology. A comparison-based sort is not ideal here, since the keys are fixed-width integers and the total number of primitives is large. For this reason, we use radix sort instead of a comparison sort.

Radix sort is particularly well suited to Morton codes because the keys are already integers with a known bit width. Rather than comparing full keys against each other, radix sort processes the keys in small groups of bits. In our implementation we use an LSD radix sort with 6-bit buckets, which means that each pass handles 64 buckets. A 30-bit Morton code therefore requires 5 passes. Since each pass is linear in the number of elements, the overall complexity is effectively $O(5 * n)$, which is linear in practice.

This choice is also cache-friendly. Each pass first builds histograms of bucket counts, then computes prefix sums, and finally scatters the elements into their new positions. The algorithm performs predictable linear scans over memory, which is usually much faster than the irregular access patterns of comparison sorts for large data sets.

One downside of the radix sort implementation we used is its high memory-bandwidth requirement. As a result, comparison-based sorting algorithms can outperform radix sort when sorting large records or when using larger key types such as $u64$. This was a primary reason for choosing 30-bit Morton codes. Initially, we stored each primitive's bounding box together with its Morton code and primitive index, resulting in records of 64 bytes. However, the increased memory traffic again caused comparison-based sorting algorithms to outperform radix sort. To reduce memory-bandwidth pressure, the bounding boxes are stored separately, while the Morton array contains only the Morton code and primitive index.

## 5.3 Constructing the tree topology

After sorting the Morton codes, all primitives are stored in a linear order that approximately preserves their spatial locality. Normally, BVH construction algorithms would recursively partition this ordered sequence and create nodes in a top-down fashion, like we did earlier. Such an approach introduces dependencies between levels of the tree, since the children of a node cannot be created until the node itself has been processed.

However, the complete tree topology can be derived directly from the sorted Morton codes. Rather than recursively partitioning the primitives, each internal node independently determines the range of primitives that it covers and the location at which that range should be split. As a result, all internal nodes can be constructed simultaneously.

Before constructing the internal nodes, we first allocate storage for the entire tree. In this LBVH representation, each primitive is stored in its own leaf node. Therefore, an LBVH containing $n$ primitives contains $n$ leaf nodes and $n1$ internal nodes, resulting in a total of $2n - 1$ nodes. Since this number is known beforehand, the entire node array can be allocated once. The layout we used follows the convention proposed by Karras. Internal nodes occupy the first $n - 1$ entries of the array, while leaf nodes occupy the remaining $n$ entries. Consequently, the position of every leaf node can be determined immediately after sorting.

```
let leaf_offset = primitive_count - 1;

for leaf_idx in 0..primitive_count {
    let node_idx = leaf_offset + leaf_idx;

    let prim_idx =
        morton_primitives[leaf_idx].primitive_index;

    nodes[node_idx] = LinearBvhNode::new_leaf(
        cached_bboxes[prim_idx].clone(),
```

```
            prim_idx,
        );
}
```

At this point, all leaf nodes have been placed into their final locations. The remaining task is to determine the connectivity of the internal nodes.

For each internal node $i$, the algorithm first determines the range of Morton codes that belongs to that node. This is achieved by comparing the common prefixes of neighboring Morton codes. The length of the common prefix between two Morton codes is measured using the number of leading equal bits. Intuitively, a long common prefix indicates that two primitives are spatially close, while a short common prefix indicates that they are spatially distant.

The algorithm begins by examining the common prefixes to the left and right of the current node. This determines the direction in which the node's range expands. An exponential search is then used to find the extent of the range, followed by a binary search to identify the exact boundary. Once the range has been found, a second binary search determines the split location that separates the left and right child ranges.

The implementation of this procedure is contained in the function `find_range_and_split`. Given the sorted Morton array and an internal node index $i$, it returns `first, split, last` where `first` and `last` define the primitive range covered by the node and `split` determines the boundary between its two children.

After the range and split position have been determined, the child indices can be computed directly. A child corresponds to a leaf node whenever its range contains exactly one primitive. Otherwise, the child corresponds to another internal node. Since every internal node can determine its children independently, no synchronization between nodes is required during topology construction.

An important consequence of this design is that topology construction has no dependencies between internal nodes. Every node performs the same computation using only the sorted Morton array and writes its result to a unique location in memory. This makes the topology construction phase embarrassingly parallel and allows all $n - 1$ internal nodes to be generated simultaneously.

This is especially suited for GPUs, since they have enough threads to generate large portions of the tree simultaneously.

### 5.4 Bounding Box propagation

While we have constructed the internal topology of the tree, we have not computed the bounding boxes of every internal node. This cannot be done at the same time as the internal topology construction, and is the last step to our LBVH. An internal node should enclose all primitives contained within its subtree. Since each child node already represents a region of space, the bounding box of an internal node can be obtained by taking the union of the bounding boxes of its children. Repeating this process throughout the hierarchy eventually yields a bounding box for every node, including the root, which encloses the entire scene. Doing this efficiently in parallel is not trivial, so in our implementation we only do this sequentially with a top-down recursion.

### 5.5 Parallelisation

Several stages of the LBVH build benefit from parallel execution. The computation of primitive bounding boxes and Morton codes can be performed independently for every primitive, making it a straightforward data-parallel operation. Similarly, the radix sort implementation uses parallel histogram construction and scattering phases to process large arrays efficiently. After sorting, the construction of leaf nodes is completely independent, since the final location of each leaf is

already known. The same is true for internal topology construction, where each internal node determines its covered range and child indices solely from the sorted Morton code array.

To exploit this parallelism, we use the `rayon` crate, which provides a work-stealing thread pool implementation for Rust. Rayon automatically manages a global pool of worker threads and distributes work among them dynamically. This approach avoids the overhead of manually creating and synchronising threads while also providing effective load balancing when individual tasks have varying execution times.

An important advantage of this design is that parallelism is exposed at the finest granularity possible. Rather than parallelising over large subtrees, which can lead to load imbalance as the tree becomes deeper, work is distributed over individual primitives and nodes. This allows all available processor cores to remain occupied throughout the build process.

# 6 Implementation Details

## 6.1 C++

Overall, the experience writing the modular pathtracer with C++ has been smooth. The build's performance and modular uses two C++ language features. Firstly, the acceleration structure is used as a templated argument: `TriangleMesh<AccelStructure>` is templated over the BVH type, so the sequential and parallel, share one interface and are swapped via a single type parameter. Secondly, build parameters (thread count, parallel threshold, adaptive splitting, flattening) are read at runtime from a string-keyed config file. Another idea is to templating the construction algorithms over the primitive type, rather than fixing them to triangle meshes, which is left as future work.

For the thread model, the build manages threads by hand: `std::thread` workers spawned per split, a thread budget distributed manually, and synchronisation through `.join()` barriers and a single `std::atomic<int>` node counter. This gives fine-grained control but pushes all correctness onto the programmer. The pure spawn-join scheme also suffers the thread-decay weakness noted above, where threads pinned to a finished subtree are never reused. This can be mitigated using OpenMP tasks (`#pragma omp task` / `taskwait`) with the cost of synchronization, which is also an idea for future work.

## 6.2 Rust

The Linear BVH code implementation and benchmarks were implmented in the Rust 2024 language. The crates (external code) used the implementation were fastrand (for random number generation), image (for writing to image files and storing textures), rayon (for parallelising), and tobj (to load OBJ files).

The implementation process overall was smooth, with some minor problems encountered. Firstly, Rust has no support for specilization, which made writing generic code for the MortonEncoder a bit difficult, although that was overcome with the use of intermediary traits implemented for each type. Furthermore, the Rust compiler did not implement the Sync trait automatically for the MortonEncoder, since it contained usage of the Vector class, which has internal mutability. The Sync trait is needed to use an object across threads. Although we do not use internal mutability inside of the MortonEncoder structure, the Rust compiler is not smart enough to deduce that, so we have to implement the `unsafe` trait ourselves.

The other problem encountered during implementation was the insufficient strength of the template meta-programming capabilities of the Rust compiler, which are currently light years behind C++. This proved an annoying problem with RadixSort, which is a really simple algorithm, and we had to use some constant as non-const instead.

We used the Rayon crate to implement the parallelisation aspect of the LinearBVH, since the parallelisation needed is trivial, as it only operates on a contiguous memory location (namely,

just a Vector). It works as a work-stealing thread pool, and incorporating rayon into the code base is trivial.

### 6.3 Code Optimizations

Whilst it is good to have human-readable and maintainable code, in performance critical applications such as rendering, we need to consider many more optimization aspects, such as heap versus stack allocation, cache misses, structure sizes and alignments. A simple memory bandwidth reduction may lead to halved computation time, for example.

1. **Minimising allocations inside hot paths** Heap allocations are expensive and should be avoided inside inner loops whenever possible. For example, we reserved space for BVH nodes before construction in order to prevent repeated reallocations, which reduced memory management overhead during the build.

2. **Keeping data layout compact and cache-friendly** The size and alignment of structures have a large impact on performance. Smaller structures fit better in cache, and well-aligned data is easier for the CPU to process. In our code, we tried to keep node sizes to 64 bytes, which allows for better memory access patterns in a 64-bit CPU.

3. **Caching and precomputing whenever possible** We tried to cache computations that would be required in the hot loops later on. For example, instead of passing a primitive array around in our recursive build function, we computed and stored the bounding box and centroids of every primitive. This avoided needless computation inside of the hot loops, passing a huge primitives array thus saturating the memory bandwidth, and redundant bounding box calculations, improving cache locality. This allowed for a huge speedup of $\sim\%50$.

## 7 Benchmarks and Discussion

All the benchmarks were either run with school computers or on an Intel i9-13900H 32GB RAM machine. The model we used, `Maria_C`, is taken from sketchfab. The model has 816,000 triangles, and 408,500 vertices.

We benchmarked both sequential and parallel building for BVH and LinearBVH. We also benchmarked sequential traversal times for the sequentially and parallelly built BVHes and LinearBVHes.

### 7.1 Parallel BVH Benchmarks

#### 7.1.1 Build Time Scaling

We tested the build time of our parallel BVH on the `Maria_C` model with different thread counts. Figure 10 shows how the build time drops as we add more threads.

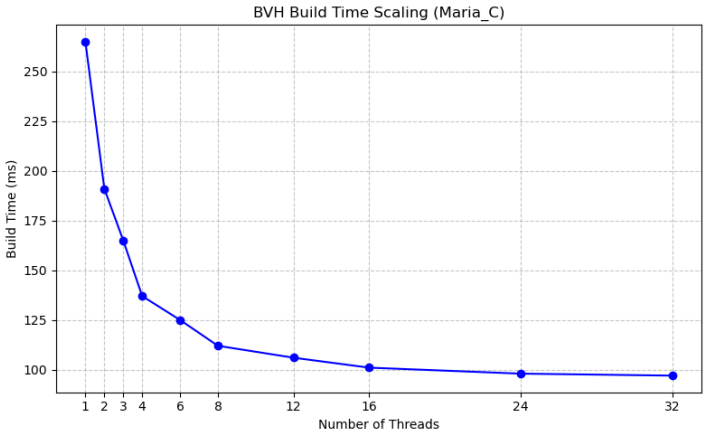



Figure 10: BVH build time scaling on the `Maria_C` mesh.

The build time improves a lot up to 8 threads, dropping from 265ms to 112ms. After 8 threads, the improvements slow down, reaching around 97ms at 32 threads. This happens because some parts of the tree building are sequential (like processing the root node), and there is some overhead when managing many threads for the smaller nodes at the bottom of the tree.

### 7.1.2 Flattening Overhead and Traversal Performance

We also added a step to flatten the tree into a 1D array. This puts the nodes closer together in memory, making ray traversal faster due to better CPU caching.

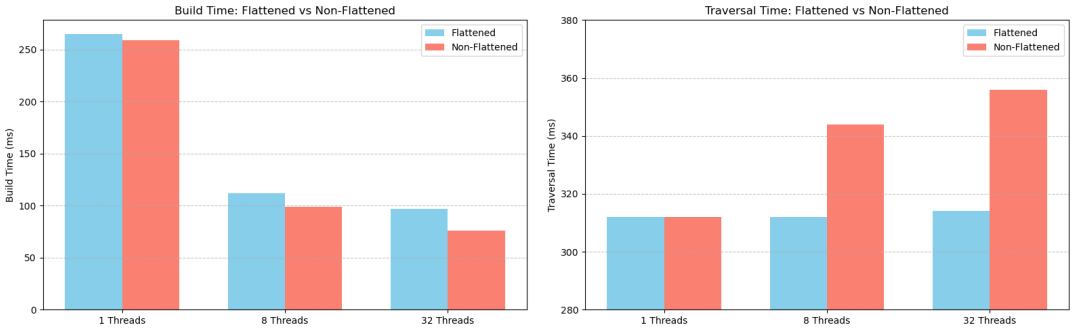



Figure 11: Build Time and Traversal Time differences between flattened and non-flattened BVH of Maria, on 1, 8 and 32 threads.

As shown in Figure 11, flattening adds a small extra cost to the build time (around 6-20ms), meanwhile it makes traversal much faster on higher thread counts. Note that traversal is sequential in these measurements. The slower traversal times come from the more fragmented memory layout made by higher build thread counts.

An interesting finding is that the traversal time of the flattened tree stays consistently fast ($\sim$ 312ms) no matter how many threads were used to build it. However, the non-flattened tree gets slower to traverse (going from 312ms to 356ms) when more threads are used. This happens because when multiple threads create nodes at the same time, the memory becomes fragmented. A ray moving through the tree has to jump around in memory, which is slow. Flattening the tree fixes this problem completely.

## 7.2 Linear BVH Benchmarks

### 7.2.1 Morton Code Radix sort Benchmark

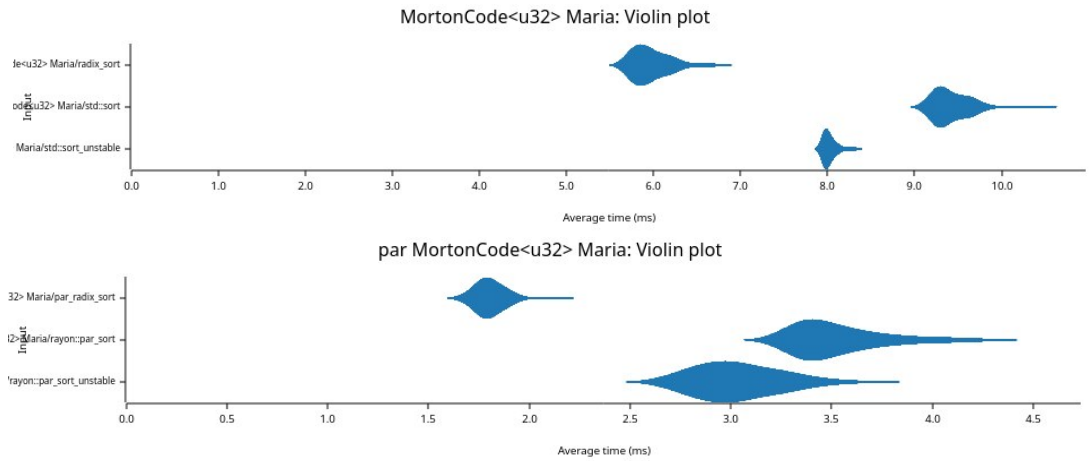



Figure 12: Morton code of Maria Radix sort vs Standard sort vs Unstable sort, sequential and parallel

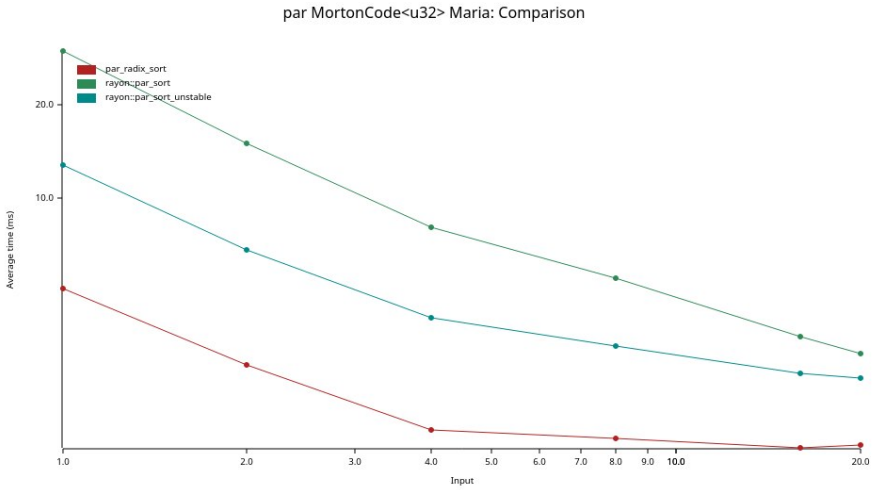



Figure 13: LBVH build time scaling with threads on the `Maria_C` mesh

As expected, radix sort performs best for Morton code reordering in both sequential and parallel settings, scaling greatly with thread count up to 20, significantly outperforming both standard sort and unstable sort. This is largely because radix sort takes advantage of the fixed-size integer structure of Morton codes, allowing it to avoid comparison-based overhead and instead process keys in linear time relative to their bit width. Overall, the results confirm that radix sort is a more suitable choice for this type of data representation.

### 7.2.2 LBVH Build and Traversal Benchmark

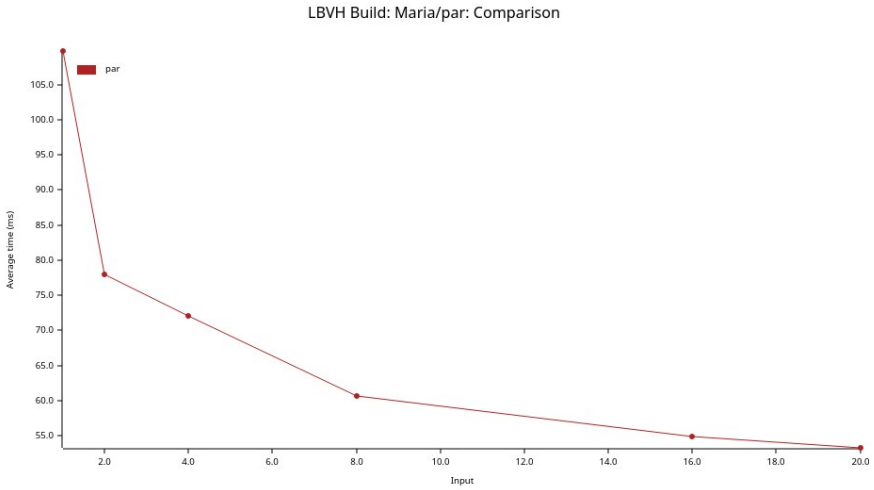



Figure 14: LBVH build time scaling on `Maria_C` mesh

Similar to parallel BVH, build time improves up to 8 threads, but it starts at a much better point: going from 110ms to 62ms at 8 threads, and reaches lowest at 53ms. Once again, this can be explained with sequential sections and thread management overhead.

## 7.3 Parallel BVH vs LBVH

In the following section, we compare the performance between Parallel BVH (top-down) and Linear BVH (bottom-up), both sequential and parallel (max thread count). We then discuss their comparison and tradeoffs between build and traversal time.

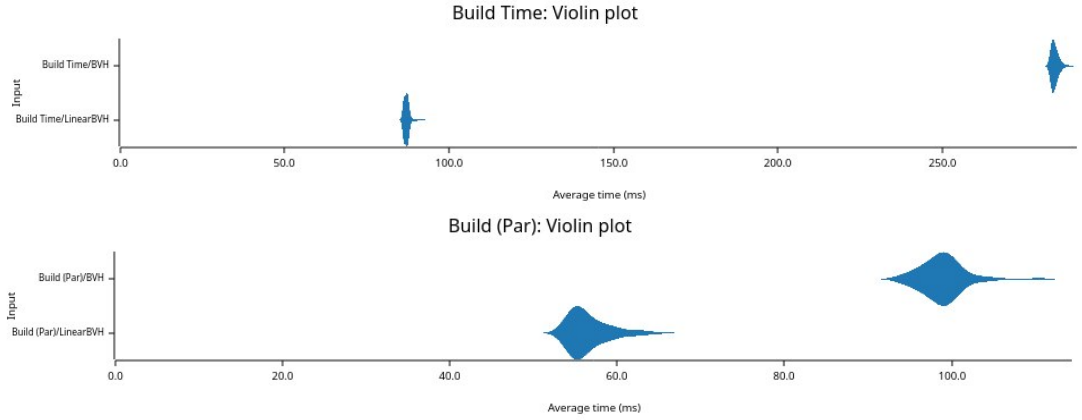



Figure 15: BVH vs LBVH of Maria Build time, sequential and max parallel

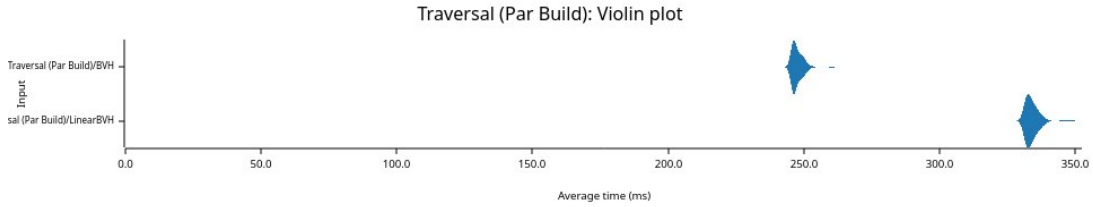



Figure 16: BVH vs LBVH of Maria Traversal time, sequential and maxparallel

In this setting, LBVH is expected to outperform parallel BVH in tree construction time at maximum thread usage, with over 200ms and 50ms win in sequential and max core parallel settings, because it uses a simpler algorithm and does not prioritize tree quality as much. On the other hand, LBVH has a small disadvantage during traversal, specifically by about 60ms, since its lower-quality tree leads to less efficient queries. However, this cost is minor compared to the large improvement in build time. Overall, this shows a tradeoff between build speed and traversal performance, and the better choice depends on the specific use case.

**Implementation and Threading Models Difference (C++ vs Rust)** As mentioned earlier, the parallel top down BVH and linear BVH were implemented in C++ and Rust, which are later compared by performance. However, beyond just run time comparison, the codebases also differ in implementation experience and threading model.

# 8 Conclusion

In this project, we explored the design and implementation of both traditional SAH-based BVHs and Linear BVHs, highlighting the trade-offs between construction quality and build performance. SAH-based construction produces high-quality hierarchies that generally lead to efficient ray traversal, but its recursive nature limits the amount of parallelism that can be exploited during construction. In contrast, LBVHs use Morton encoding and sorting to derive the hierarchy directly from the spatial ordering of primitives, exposing a much greater degree of parallelism and making them particularly well suited to modern multi-core processors and GPUs. We also examined several practical implementation optimizations, including caching frequently used computations, storing nodes in contiguous memory, and flattening tree structures to improve cache locality and traversal performance. Ultimately, neither approach is universally superior: SAH-based BVHs typically offer better traversal efficiency, while LBVHs provide substantially faster construction times. The most appropriate choice therefore depends on the requirements of the application, with static scenes favouring high-quality hierarchies and dynamic or real-time workloads benefiting from rapid, highly parallel construction.

# References


[1] Tero Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, page 33–37, Goslar, DEU, 2012. Eurographics Association.

[2] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[3] Wenzel Jakob Matt Pharr and Greg Humphreys. Physically based rendering: From theory to implementation. `https://www.pbr-book.org/`, 2024.

[4] Chris Zhang. Rendering notes: Bvh. `https://blog.zcy.moe/en/blog/bvh/`, 2022.

[5] Ingo Wald, Thiago Ize, and Steven G. Parker. Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Computers  Graphics*, 32(1):3–13, 2008.

[6] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[7] John D. Owens Mark Harris, Shubhabrata Sengupta. Chapter 39. parallel prefix sum (scan) with cuda. `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda`.