Efficient Tree-based Parallel Algorithms for N-Body Simulations Using C++ Standard Parallelism

Thomas Lane Cassell and Tom Deakin
School of Computer Science
University of Bristol
Bristol, UK, BS8 1UB
Email: tom.deakin@bristol.ac.uk

Aksel Alpay and Vincent Heuveline
Engineering Mathematics and Computing Lab
Interdisciplinary Center for Scientific Computing
Heidelberg University
Heidelberg, Germany
Email: aksel.alpay@uni-heidelberg.de
Email: vincent.heuveline@uni-heidelberg.de

Gonzalo Brito Gadeschi
GPU Architecture
NVIDIA
Munich, Germany
Email: gonzalob@nvidia.com

***Abstract*—The Barnes-Hut approximation for N-body simulations reduces the time complexity of the naive all-pairs approach from $O(N^2)$ to $O(N \log N)$ by hierarchically aggregating nearby particles into single entities using a tree data structure. This inherently irregular algorithm poses substantial challenges for performance portable implementations on multi-core CPUs and GPUs. We introduce two portable fully-parallel Barnes-Hut implementation strategies that trade-off different levels of GPU support for performance: an unbalanced concurrent octree, and a balanced bounding volume hierarchy sorted by a Hilbert space-filling curve. We implement these algorithms in portable ISO C++ using parallel algorithms and concurrency primitives like atomics. The results demonstrate competitive performance on a range of CPUs and GPUs. Additionally, they highlight the effectiveness of the `par` execution policy for highly concurrent irregular algorithms, outperforming `par_unseq` on CPUs and GPUs with *Independent Thread Scheduling*.**

I. Introduction

N-Body simulations are often used in cosmology to understand the gravitational interactions between bodies (planets, asteroids, etc), and more recently for high-dimensional data visualisation in machine learning. The naive $O(N^2)$ all-pairs algorithm is simple enough to be used as a canonical example for exploring new parallel programming models and GPU accelerators as it is trivially parallelizable. In practice however, approximations, such as Barnes-Hut [1], are used to reduce the work required at the expense of increased complexity through the introduction of a tree data structure. Real contemporary scientific applications may use more complex algorithms than Barnes-Hut, such as solving the problem with a multipole method or a Fourier transform, the study of accelerating Barnes-Hut on GPUs is still interesting in its own right as the tree-based data structures it uses are transferable to other domains and algorithms. The tree represents a hierarchical model of the spatial dimension based on the proximity of the bodies locations to each other. This can then be used to approximate long distance interactions between the bodies by considering some aggregated value representing several particles far away in space, rather than each pair-wise interaction. The primary challenge therefore is for there to be an efficient parallel scheme to construct the tree, especially for GPU accelerators where a large degree of independent concurrent work is required for good performance.

Expressing parallel algorithms requires a parallel programming model. Further, we require a model which has the support necessary to run on a range of devices, including GPUs. ISO C++ introduced parallel programming features to write complex parallel and concurrent algorithms directly in portable C++, eliminating the need for auxiliary models like OpenMP. Major HPC vendors - AMD, Intel, and NVIDIA - support it on both CPUs and GPUs. Our previous work shows that C++ delivers high performance across various compute- and memory-bound tasks on diverse hardware [2]. This study extends that exploration to irregular algorithms, specifically the Barnes-Hut algorithm, achieving good performance on CPUs and GPUs for all stages of the algorithm.

The main contributions of this work are:

1) Developed two novel parallel schemes for Barnes-Hut on multi-core CPUs and GPUs: a) A *Concurrent Octree* scheme with parallel tree construction using an starvation-free algorithm (Section IV-A); b) A balanced Hilbert-sorted Bounding Volume Hierarchy (BVH) built from bodies sorted along a Hilbert space-filling curve (Section IV-B).
2) Characterize the performance of portable heterogeneous single-source implementations of these algorithms implemented using ISO C++ parallel algorithms and concurrency primitives across a wide range of CPUs and GPUs architectures (Section V) and compare them against a state of the art SYCL implementation.

We have made our open source code base available online.[1]

II. Standard Parallelism in C++

This section introduces the main C++ implementations and parallel programming features used in this study: the C++ parallel algorithms for *Parallel For* (`for_each`), *Parallel Reduce* (`transform_reduce`), and *Parallel Sort* (`sort`), C++ primitives for finer grained concurrency (`atomic` and

See Acknowledgement for funding sources.

[1] https://github.com/UoB-HPC/stdpar-nbody

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00099

Algorithm 1 Parallel Vector Addition (ISO C++)

```
auto r = views::iota(0, n);
for_each(par_unseq, begin(r), end(r),
  [=](int i) {  x[i] = x[i] + y[i]; });
```

Algorithm 2 Barnes-Hut Time Integration Loop

1: **for** $t \in$ Timesteps **do**
2: CALCULATEBOUNDINGBOX
3: BUILDTREE
4: CALCULATEMULTIPOLES
5: CALCULATEFORCE
6: UPDATEPOSITION
7: **end for**

`memory_order`), and C++ execution policies for specifying forward progress requirements (`par` and `par_unseq`).

C++ parallel algorithms are used to offload parallel work to multi-core CPUs and GPUs. They extend the more than hundred sequential C++ standard library algorithms with execution policies [3] that express whether it is safe to parallelize (`par`) and vectorize (`par_unseq`) their invocation. Applications are then responsible to ensure algorithm invocations do not introduce data-races or deadlocks when run concurrently on multiple threads. Algorithm 1 illustrates an implementation of parallel vector addition with this programming model. Most implementations provide a `-stdpar=<CPU or GPU>` compiler flag to control in which processor parallel work runs. Memory migration across devices is implicit and relies on system technologies like Unified Virtual Memory (UVM), Unified Shared Memory (USM), or Linux's Heterogeneous Memory Management [4]. There is a variety of C++ implementations that offload parallel algorithms to either multi-core CPUs or GPUs, both vendor specific (AMD's ROCm stdpar, Intel oneAPI, and NVIDIA HPC SDK), and open-source (AdaptiveCpp [5]), as well as many other implementations that only support CPUs (Clang, GCC, MSVC).

Atomic Operations make concurrent memory accesses from different threads *indivisible*. For example, incrementing a memory location using non-atomic operations is split into a load into a temporary, an increment operation on the temporary, and a store, whereby these partial updates by concurrent threads may cause loss. Beyond atomic stores and loads, this study uses `std::atomic<int>::fetch_add(N)` to increment memory locations by N, and `compare_exchange(old, new)` operations replace the value in a memory with `new` only if it compares equal to `old`. Atomics are not *vectorization safe*.

Memory ordering weakens the default sequentially-consistent ordering of atomic operations. This study performs reductions that do not need to order any other memory operations by specifying `memory_order_relaxed`. This study implements locks and critical sections by using `memory_order_acquire` to acquire a lock or enter a critical section, and `memory_order_release` to release the lock or exit the critical section. Atomic memory operations are *vectorization-unsafe* (see [algorithms.parallel.defns] [6]) and require the use of the parallel execution policy `par`. Readers unfamiliar with the C++ memory model, atomic operations, and memory ordering, are referred to Williams [7].

Forward progress requirements are specified via execution policies. The `par` policy provides *parallel forward progress*, i.e., it guarantees that if a thread starts running it will be eventually scheduled again. This enables applications to use *starvation-free* algorithms [3], in which threads enter critical sections and take locks. If a thread is suspended while holding a lock, `par` guarantees that it will eventually be scheduled again, allowing it to release the lock. The `par_unseq` policy does not provide this guarantee: it provides *weakly parallel forward progress* [3], requiring all threads to be able to make progress independently of each other, and only allowing *vectorization safe* operations to be used. This enables SIMD-unit like implementations that schedule threads in lock step, but prevents threads from entering critical sections or using atomic operations. The parallel algorithms in the *Concurrent Octree* strategy make extensive use of critical sections and require `par`, while the parallel algorithms in the *Hilbert BVH* strategy only require `par_unseq`. Readers unfamiliar with forward-progress and execution policies are further referred to Pennycook et al. [8]. While most GPUs only provide *weakly parallel forward progress* and can only offload parallel algorithms using `par_unseq`, the survey from Sorensen et al. [9] shows that NVIDIA GPUs with *Independent Thread Scheduling* (ITS) [10], [11], i.e., all NVIDIA GPU architectures since Volta, provide *parallel forward progress* and can offload starvation-free parallel algorithms that use `par`.

III. N-BODY PROBLEMS

An n-body problem describes the evolution of N distinct bodies of mass m_i at positions $\mathbf{x}_i$ under the action of external forces $\mathbf{F}_i$, such as those generated by the gravitational potential in Equation 1 where G is the gravitational constant. To simulate the motion of these particles, the system of ordinary differential equations (ODEs) is discretized using Störmer–Verlet time integration [12]. The force $\mathbf{F}_i$ depends on the distance between body i and all other bodies, coupling all ODEs in the system. These forces must be recalculated at every time step as the bodies move. The algorithm that evaluates the gravitational potential by computing all pairwise force interactions between the bodies in a brute-force manner is known as the *particle-particle method* or the *all-pairs algorithm*, and has $O(N^2)$ time complexity.

$$\mathbf{F}_i = G m_i \sum_{j=1, j\neq i}^{n} \frac{m_j(\mathbf{x}_j - \mathbf{x}_i)}{||\mathbf{x}_j - \mathbf{x}_i||_2^3} \tag{1}$$

IV. PORTABLE PARALLEL ALGORITHMS FOR BARNES-HUT

The *Barnes-Hut algorithm* [1] is a widely used method to compute these forces with $O(N \log N)$ time complexity in

Algorithm 3 Parallel Bounding Box Reduction (ISO C++)

```
auto [x_min, x_max] // bbox coords
 = transform_reduce(par_unseq,
  // Range of body indices:
  body_ids.begin(), body_ids.end(),
  // Initial value for the reduction:
  std::make_tuple(vec::max(), vec::min()),
  // Reduces two boxes into one
  [](auto a, auto b) {
    return {min(a, b), max(a, b)};
  },
  // Map body index to box
  [s = system.state()](auto i) {
    return {min(s.x[i]), max(s.x[i])};
  }
);
```

theory, though not always in practice [13]. It employs a hierarchical spatial decomposition to group particles in regions of space, approximating many long-range interactions by aggregating the contributions of multiple particles into a single interaction. As particles move, the spatial decomposition must be updated to reflect their new positions. Popular data-structures for the hierarchical spatial decomposition include trees, such as quadtrees, octrees, kd-trees, and BVH. However, these spatial decompositions result in irregular tree traversals for force computation.

The Barnes-Hut algorithm has five steps during time integration shown in Algorithm 2:

1) CALCULATEBOUNDINGBOX: performs a parallel reduction over all body positions as illustrated by Algorithm 3 to determine the bounding box of the root node of the tree, which is the smallest box containing all bodies.

2) BUILDTREE: constructs the BVH or octree ensuring that each leaf node contains at most one body. The specific algorithms vary for the two strategies considered and are detailed in Sections IV-A and IV-B.

3) CALCULATEMULTIPOLES: the multipole expansion is analogous to a Taylor-Series expansion, and the multipole moments to its coefficients. This step computes the moments at each tree node, which for the problem considered are just the center of mass and momentum of all bodies covered by the node. For a leaf node, these are simply the mass and momentum of the body it contains, if any. For any other node, these can be computed from the center of mass and momentum of its children, and are computed recursively from the leaf nodes up to the root node, which contains the total mass of the system. While this study uses monopoles for exposition, the algorithms described here extend to multipoles.

4) CALCULATEFORCE: computes forces by approximating long-range interactions with a multipole expansion. For each body, the tree is traversed from the root in depth-first search (DFS) order. If the node's centre of mass is further from the body than some threshold distance, the force interactions with bodies covered by that node are approximated by that node's multipole moments, and its children are not traversed. At leaf nodes, exact pair-wise interactions are computed.

5) UPDATEPOSITION: advances each body position in time by integrating the force with the Störmer-Verlet method.

The following two sub-sections expand on the details of BUILDTREE, CALCULATEMULTIPOLES, and CALCULATEFORCE, for the two strategies evaluated by this study.

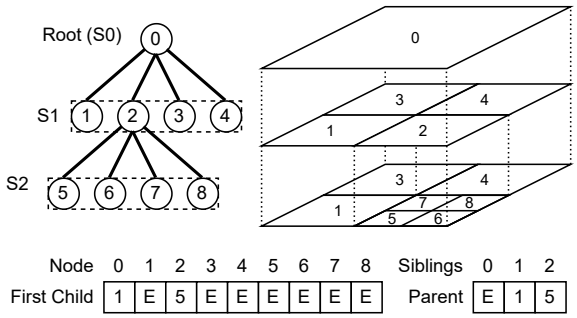

Fig. 1. Quadtree data-structure and memory layout: one offset to first child per node (E: Empty), one parent offset per siblings.

A. Concurrent Octree

This section provides an overview of the Concurrent Octree strategy, its data structure, and its three parallel algorithms.

Figure 1 shows the graph and in-memory representation of the quadtree data structure; the octree uses a similar representation. Each node stores an offset to its first child (4 bytes), enabling root-to-leaf traversals during the BUILDTREE and CALCULATEFORCE steps. Nodes are isotropically sub-divided, meaning each node has either four children in Morton order or is a leaf node. Additionally, each group of sibling nodes stores an offset to its parent (4 bytes per 4 nodes: 1 byte/node), enabling leaf-to-root traversals during the CALCULATEMULTIPOLES step. A memory pool for a concurrent bump allocator is reserved beforehand, allowing threads to dynamically allocate memory concurrently with relaxed atomic add operations. Since the tree size is unknown, it is estimated from the number of nodes required to fit all bodies at an isotropically sub-divided tree level.

All of the Concurrent Octree steps are massively parallel, i.e., the parallelism available in each algorithm is $O(N)$, i.e., proportional to the input size.

1) BUILDTREE: The tree is initialized with a single root node from the bounding box computed in the prior CALCULATEBOUNDINGBOX step. The BUILDTREE step, illustrated in Algorithm 4, inserts all bodies in parallel into the tree with a *Parallel For* that performs a root-to-leaf traversal for each body until the leaf node covering the body is found. Threads either insert the body into an *Empty* leaf node or sub-divide a *Body*-containing leaf node by allocating new children, moving the node's body into the appropriate child, and recursing into the children. Fine-grained inter-thread synchronization avoids races from two threads attempting to sub-divide the same

Algorithm 4 Parallel Octree BUILDTREE step

```
1: for b ∈ Bodies do                          ▷ Parallel loop with par
2:     index ← root node
3:     while True do                          ▷ Root to Leaf traversal
4:         next ← atomic load acquire child[index]
5:         if next not (empty | locked | leaf node) then
6:             index ← next sibling covering b
7:             continue                       ▷ Traverse to child
8:         else if next is empty & try lock acquire next then
9:             insert b at next
10:            child[index] ← store release body
11:            break                  ▷ Body b inserted successfully
12:        else if next has body & try lock acquire next then
13:            c ← allocate children
14:            move next's body to next's child covering it
15:            child[index] ← store release c
16:            continue               ▷ Next try traverses to child
17:        end if                         ▷ Failed to lock: try again
18:    end while
19: end for
```

Algorithm 5 Node sub-division critical section

```
atomic_ref<int> l{lock};
if (l.compare_exchange_weak(Body, Locked,
                            acquire)) {
    // If node contains body:
    // - only one thread acquires lock
    //   & enters critical section
    // ..sub-divides the node..
    // Children update releases lock:
    l.store(children, release);
} else {
   // Threads that fail wait
   while(l.load(acquire) == Locked);
}
```

node concurrently. We extend the token values Empty, Body to include a Locked state that implements the following locking mechanism illustrated in Algorithm 5. Threads use `std::atomic_ref::compare_exchange_weak` operations to attempt to lock *Empty* or *Body*-containing leaf nodes. The thread that successfully acquires the lock performs the subdivision within a critical section while the remaining threads wait. The likelihood of waiting decreases as the tree grows and threads spread throughout it. This algorithm is *starvation-free*, completing if every thread that enters a critical section is eventually re-scheduled to exit it, avoiding deadlock. Therefore, it requires *Parallel Forward Progress* guarantees, and the *Parallel For* must use the `par` execution policy [3]. To enhance performance beyond atomics' default sequentially consistent memory ordering, acquire/release operations are used: acquire for taking the lock and waiting, and release for releasing the lock.

2) CALCULATEMULTIPOLES: The multipole moments are computed using a *Parallel Tree Reduction* via a leaf-to-root traversal. Since leaf nodes are scattered, the algorithm schedules one thread per node, which exit if a node is not a leaf. Thus, the available parallelism remains proportional to the number of bodies ($O(N)$). Each thread computes the multipole moments of its node, which for the body-containing leaf nodes are just the mass and momentum of the body. Threads then accumulate these onto their parent node with a relaxed atomic add (`std::atomic_ref::fetch_add`) and signal completion with an acquire+release integer atomic increment on a per node counter, which returns the old value. The last thread to update a node recurses into its parent, while its sibling threads simply exit (see Figure 2). This algorithm has no critical-sections and is *wait-free*, but since atomic acquire-release operations synchronize, they are C++ vectorization-unsafe, requiring the `par` execution policy.

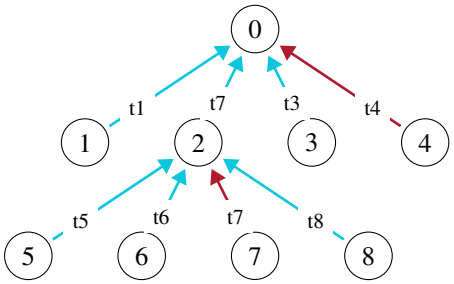

Fig. 2. Parallel Tree reduction in CALCULATEMULTIPOLES. Nine threads - one per node - are launched. Two of those - nodes 0 and 2 - exit immediately as their nodes are not leafs. The others update their parent moments and counter, and exit when they are not the last threads to do so. The remaining threads (shown in red) recurse to their parents.

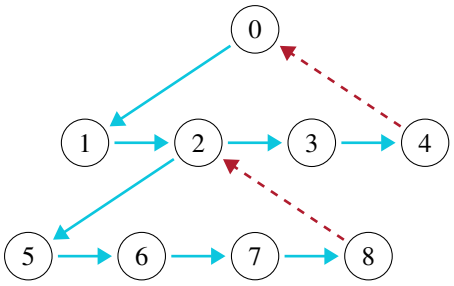

Fig. 3. The arrows show the order in which nodes are traversed during the Depth First Search in CALCULATEFORCE. A solid blue arrow represent a *forwards step* and a dashed red arrow represents a *backwards step*. In backward steps, the node offset of the children is always greater than its parent, enabling a stackless implementation, that traverses to a sibling or to the parent depending only on the prior node offset.

3) CALCULATEFORCE: The force on each body is approximated using the multipole expansion by performing a DFS tree traversal that only explores children within a distance threshold from the node. A stackless DFS leverages the offset of a node's children always being larger than their parent, allowing the implementation to run well with a bounded stack size on GPUs (Figure 3). It is implemented with a *Parallel For* and the `par_unseq` execution policy, since the computations for each node are independent.

Algorithm 6 BVH along Hilbert Ordering

1: **for** $t \in$ Timesteps **do**
2: CALCULATEBOUNDINGBOX
3: HILBERTSORT
4: BUILDTREEACCUMULATEMASS
5: CALCULATEFORCE
6: UPDATEPOSITION
7: **end for**

Algorithm 7 HILBERTSORT masses & positions

```
vector<double> m, vector<vec3<double>> x;
auto r = views::zip(m, x);
sort(par, begin(r), end(r),
 [](auto a, auto b) {
    auto [am, ax] = a;
    auto [bm, bx] = b;
    return hilbert(ax) < hilbert(bx);
});
```

B. Hilbert-sorted Bounding Volume Hierarchy (BVH)

This section provides an overview of the Hilbert-sorted BVH strategy, introducing its data structures and describing its three parallel algorithms, which only require *weakly parallel forward progress* [3] and run on GPUs without *Independent Thread Scheduling* [10]. As such, all routines can use the `par_unseq` execution policy. This approach is based on Alpay's quasar microlensing code [14] and the SpatialCL library [15], with similar algorithms found in computer graphics [16]. The strategy, illustrated in Algorithm 6 and Figure 4, sorts all bodies along a Hilbert space-filling curve, then builds both a BVH and reduces the multipole moments in a single leaf-to-root traversal.

The BVH is a balanced binary tree with power-of-two leaf nodes, such that the number of BVH levels, nodes per level, and total number of nodes, are predetermined. The data-structure can be interpreted as a skip list, enabling efficient stackless traversal algorithms without the need for explicit connectivity information in memory.

1) HILBERTSORT first grids the bodies within the coarsest equidistant Cartesian grid capable to hold all bodies. The Hilbert index of each body's grid cell is computed with the Skilling's Grey algorithm [17] and is used to sort the bodies in parallel with `std::sort`, as shown in Algorithm 7. Note the Hilbert index is precomputed to avoid recomputation.

2) BUILDTREEANDMULTIPOLES first constructs the BVH leaf nodes from the bodies. Then, a *Parallel For* over uninitialized nodes of the subsequent coarser level reduces the bounding boxes and multipole moments of their children. This process repeats level by level until the root node is reached, as illustrated in Figure 4. Since the reductions at each node are independent, `par_unseq` execution policy suffices.

3 CALCULATEFORCE: this step is similar to Concurrent Octree's, with two key differences. First, the skip list nature of the balanced BVH allows for jumps from a leaf node to the next node in the DFS traversal across multiple levels without traversing nodes in-between. Second, since bounding boxes of BVH nodes may be elongated and may overlap (e.g., see boxes of 0,1 and 2,3 in Figure 4), the multipole expansion may need to evaluate a larger number of terms than Concurrent Octree for the same distance threshold. Similarly, since the BVH directly jumps to the next node on a higher level instead of unwinding the tree and reevaluating the distance threshold from the top downwards, it might include more computations on lower tree levels for the same value of the distance threshold. Consequently, the interpretation of the distance threshold between the octree and the BVH is different, and the accuracy of computation may vary for the same distance threshold.

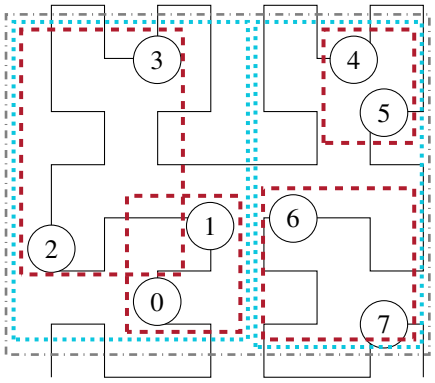

Fig. 4. Hilbert-sorted BVH. Adjacent bodies along the Hilbert curve are grouped into the next level of the BVH (dashed, red bounding boxes). The two left-most red bounding boxes show that these may overlap. Then, adjacent bounding boxes along the Hilbert curve are grouped pairwise (dotted blue bounding boxes) to form the next level of the tree, and so on, until only the root bounding box (shown in gray) remains.

Table I. Hardware (HW), Software (SW) environment versions for AMD ROCm, Intel OneAPI, or NVIDIA HPC SDK, Toolchains: AdaptiveCpp (a), Clang (c), GCC (g), HIPCC (h), NVC++ (n), DPC++ (o), theoretical peak bandwidth, measured BabelStream C++ StdPar TRIAD Bandwidth.

HW	SW	Toolchain	Th. [GB/s]	Exp. [GB/s]
AMD MI100	6.1.3	h,a	1200	1013
AMD MI250 GCD	6.1.3	h,a	1600	1375
AMD MI300X	6.1.3	h,a	5300	4006
AMD 9654 (Genoa)	13,18	g,c	460	287
AWS Graviton4	13,18	g,c	530	413
Intel PVC1550 1/2 Tiles	24.1	o,a	3276	1079 / 2054
Intel 8480C (SPR)	13,18	g,c	307	197
NV Grace-120	13,18	g,c	500	448
NV V100-16	24.7	n,a	900	845
NV A100-80	24.7	n,a	2000	1768
NV H100-80	24.7	n,a	3300	3073
NV GH200-480	24.7	n,a	4000	3683

V. RESULTS AND ANALYSIS

A. Experimental setup

The experimental setup is summarized in Table I. To evaluate the peak performance and portability of the single-source ISO C++ implementations of *Concurrent Octree* and *Hilbert BVH*, we select a mix of GPUs and CPUs from various vendors. The GPU software stack versions (AMD ROCm, Intel OneAPI, and NVIDIA HPC SDK) are listed in the "SW" column.

Toolchains: To evaluate the algorithms on multi-core CPUs and GPUs, we use a range of vendor-specific and open-source ISO C++ implementations. Each experiment is conducted using two toolchains per system. For GPU experiments, we select a vendor-specific toolchain—AMD ROCm (h), Intel OneAPI (o), or NVIDIA HPC SDK NVC++ (n)—alongside the vendor-neutral open-source AdaptiveCpp (a) toolchain, which is the only implementation that runs on all CPUs and GPUs evaluated. CPU experiments use GCC (g) and Clang (c). Each measurement reflects the best-performing implementation.

We encountered the following C++ implementation issues:
1) ROCm stdpar runs all supported algorithms correctly on all AMD GPUs evaluated but exhibits sub-optimal performance when `--hipstdpar-interpose-alloc` is disabled and `XNACK=1`. We believe enabling this option would improve performance, but it caused crashes similar to those reported by Lin et al. [18].
2) C++23 `views::zip` is not supported in the versions of AdaptiveCpp, Clang, and NVC++ evaluated. For NVC++, we replace it with `zip_iterator`, while with AdaptiveCpp and Clang, we sort an auxiliary buffer of Hilbert and body index pairs, applying it as a permutation afterwards.
3) GNU C++ Standard Library, used by GNU and Clang, leverages Intel TBB on both x86 and Arm, but has a bug [19] that causes parallel algorithms to run sequentially when iterators from C++20 ranges like `iota` or `zip` are used. We apply the bug's patch to ensure parallel execution.
4) Non-relaxed GPU atomics exhibited bugs in AdaptiveCpp and NVC++. Updating AdaptiveCpp to commit `41225f3` and using `cuda::atomic` with NVC++ resolved these.
5) OneAPI compilation flow using the experimental `-fsycl-pstl-offload=gpu` flag failed due to missing features within its parallel STL implementation. Replacing `par_unseq` with `oneapi::dpl::execution::par_unseq`, showed low performance indicating other issues requiring further investigation [20], [21].

Validation: We validated the experimental environments by comparing STREAM TRIAD benchmark results, using the BabelStream ISO C++ parallel algorithms implementation, against the hardware's theoretical peak bandwidth (see Table I). All experiments follow the workflow described in the Artifact Description (Appendix A), which employs a container to ensure consistent toolchain and dependency versions. Additionally, we validate the performance and accuracy of our implementations against the state-of-the-art n-body solver from Thüring et al., implemented in SYCL [22], by simulating the evolution of 1,039,551 small solar system bodies from NASA's JPL Small-Body Database [23] for one full day with a timestep of one hour. The L2 error norm of the final body positions among all three implementations is below 10^{-6}. Our Octree algorithm outperforms BVH by 3.3x, and Thüring et al. by 5.2x, on H100.

Simulation workload: The experiments simulate a deterministic collision between two neighboring Galaxies with varying number of bodies, using double precision[2] (FP64) and a theta factor of 0.5. The simulations produce consistent final results across all systems, conserving mass and energy.

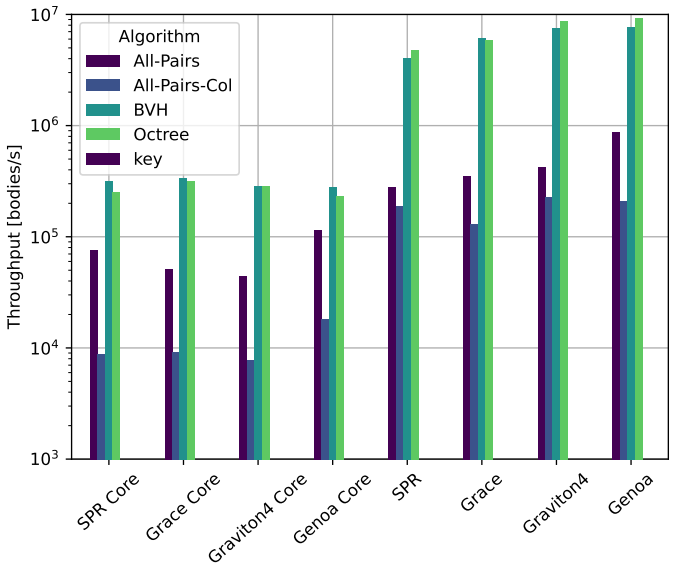

Fig. 5. Single-core sequential throughput vs single-socket parallel throughput for **tiny-size** *galaxy* workload (10^4 bodies). Genoa set to NPS1 mode.

Algorithms: We evaluate the proposed $O(N \log N)$ Concurrent *Octree* and Hilbert-sorted *BVH* algorithms, along with two $O(N^2)$ implementations of the brute-force all-pairs algorithm: the classical *All-Pairs* implementation, parallelized over the bodies using `par_unseq`, and *All-Pairs-Col*, which uses `par` to parallelize over the force-pairs with concurrent accumulation via `atomic::fetch_add`.

B. Measurements & Analysis

Sequential vs Parallel Execution is evaluated by replacing parallel execution policies with the sequential `seq` policy. Figure 5 shows algorithm throughput for tiny inputs (10^4 bodies) on single CPU cores and single socket. We observe up to 40x performance improvements due to parallelization, reflecting the massively-parallel nature of the algorithms. The Octree and BVH algorithms outperform classical brute-force algorithms due to their better algorithmic complexity. On CPUs, the classical *All-Pairs* algorithm outperforms *All-Pairs-Col*, which incurs higher coherency traffic due to all-to-all atomic reductions.

CPU/GPU Performance Across Algorithms: Figure 6 shows algorithm throughput for small problems (10^5 bodies) on CPUs and GPUs. *All-Pairs* exhibits higher throughput than *All-Pairs-Col* on all systems except NVIDIA GPUs. Measuring *All-Pairs-Col* on AMD and Intel GPUs requires replacing `par` with `par_unseq`, but this was slow on PVC. Overall, MI300X delivered the highest throughput for all-pair family algorithms. The *BVH* algorithm runs on all evaluated systems, while the *Octree* algorithm only runs on CPUs and NVIDIA

[2]We use double precision to enable comparisons with Thüring et al. [22].

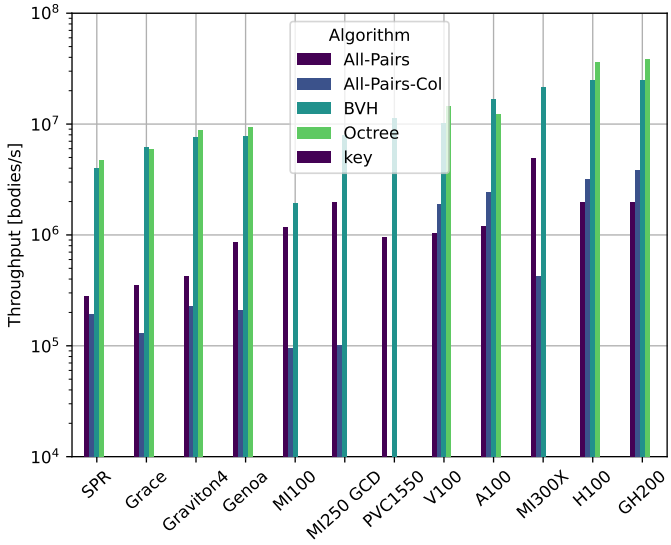

Fig. 6. Algorithm throughput for **small-size** *galaxy* workload (10^5 bodies).

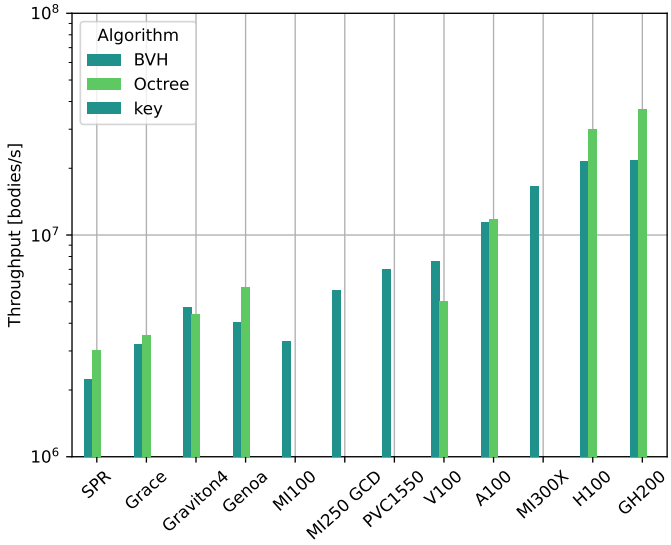

Fig. 7. Algorithm throughput for **mid-size** *galaxy* workload (10^6 bodies).

GPUs due to its forward progress requirements. On GH200, *Octree* delivered the highest overall throughput, outperforming *BVH* by 1.5x for a fixed distance threshold (see the end of Sec. IV-B). These trends extend to mid-sized problems (10^6 bodies), as shown in Figure 7. Attempts to run *Octree* on Intel and AMD GPUs reliably caused them to hang, which is expected since running this experiment requires introducing undefined behavior by replacing `par` with `par_unseq` and they do not provide ITS [20], [24], [25].

You may notice that for the small-size in Figure 6, the BVH algorithm outperformed the Octree one, whereas the reverse occurs for the mid-size in Figure 7. This can be attributed to the partitioning of L2 introduced in the later Ampere architecture to improve bandwidth [26], causing the

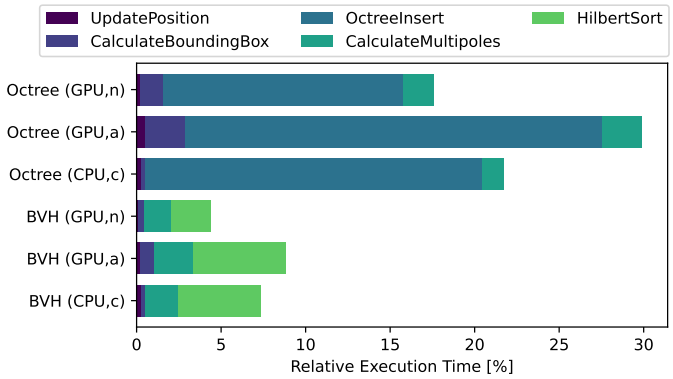

Fig. 8. Relative execution time of algorithm components for small-size *galaxy* workload (10^5 bodies) on GH200 CPUs and GPUs with AdaptiveCpp (a), NVC++ (n), and clang (c). The remaining execution time is spent in CALCULATEFORCE (not shown).

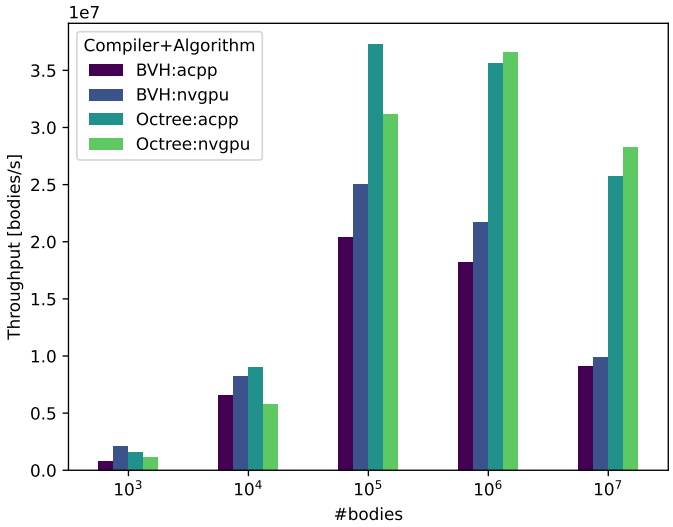

Fig. 9. Comparison of two ISO C++ heterogeneous toolchains on GH200.

inevitable increase of latency for cache coherence for the atomic operations we use. Note that this is improved in the Hopper GPUs where Octree outperforms BVH at scales larger than 10^5 (small-size).

Implementation Impact: Figure 8 shows the relative execution time of each algorithm on CPUs and GPUs of a GH200 system (excluding CALCULATEFORCES) for three compilers. This allows us to measure the impact of the compiler and stdpar implementation. The results show that such variation is relatively small, attributed mainly in the sorting algorithm which is not necessarily optimised in all compilers.

Figure 9 further compares the throughput of two different heterogeneous ISO C++ implementations on GH200: AdaptiveCpp and NVC++, for different numbers of bodies. The two compilers show comparable performance, with the largest absolute difference being 1.25x. As most of the runtime is attributed to CALCULATEFORCES, the runtime differences are mostly attributable there, with other differences being

relatively minor as previously identified in Figure 8.

GPU NUMA effects: On Intel PVC 1550 Max, the best result for small problems is obtained with 2 tiles, while the best result for larger problems is obtained with 1 tile, suggesting that NUMA effects may penalize throughput for larger problems. Our measurements show the best result of either one or two tiles. On MI250, the 2 GCDs cannot be programmed as a single GPU, so all result shown are for a single GCD.

VI. Related Work

The Barnes-Hut algorithm, originally developed for cosmology simulations, has recently gained renewed interest due to its application in the t-SNE technique for high-dimensional data visualization in machine learning [27], [28].

Octree data structure is central to many Barnes-Hut based methods. Thüring et al. developed a Barnes-Hut implementation in SYCL and benchmarked it on GPUs [22]. They enhance parallelism by dividing the top-down approach by Burtscher and Pingali [29] into two stages: first, building a partial tree in a single work-group; and second, in a subsequent kernel, constructing the remaining independent sub-trees in parallel with one work-group per sub-tree. This two-stage approach is necessary due to the synchronization constraints dictated by the memory and execution model of work-items and work-groups. We benchmarked our performance against this.

Iwasawa et al. explore a highly parallel Barnes-Hut implementation on the many-core Sunway TaihuLight supercomputer [30]. They offload part of the algorithm to the accelerator cores, while keeping tree construction on the "simple cores". This study highlights that tree construction can be computationally expensive, and they amortized this cost by reusing the same tree over multiple time steps as an additional approximation. This approach can be applied to any Barnes-Hut implementation.

Hamada et al. propose a parallel algorithm for GPUs where different phases of the Barnes-Hut algorithm are executed on the host CPU, with only the force calculation performed on the GPU [31]. Similarly, Xu et al. developed Redwood to allow tree traversals on CPUs to occur concurrently with tree node computations on (integrated) GPUs [32]. Their approach discounts the tree construction.

More complex N-body approximations, such as the Fast Multipole Method (FMM), also utilize an octree, although with different approaches to mass estimation. Atkinson and McIntosh-Smith investigated the tree traversal and multipole accumulation algorithms of FMM on GPUs and many-core accelerators using OpenMP [33]. However, they did not address the parallel construction of the tree.

In Section IV-A, we take a different novel approach to parallelize Octree construction in CPUs and GPUs via finer-grained synchronization.

Bounding Volume Hierarchies along space filling curves are widely studied, particularly for efficient ray tracing [34]. Walter et al. developed a BVH in a bottom-up manner by calculating a distance metric between bodies stored in a kd-tree, enabling fast searching based on this metric. Lauterbach et al. improve this approach [35] by sorting the bodies along a Morton space filling curve and aggregating bodies based on shared leading bits of the Morton code. This aggregation is not necessarily pairwise, and the resulting BVH tree may not be binary. The approach was further refined in the PLOC algorithms [36]–[38], which aggregate nodes based on a radius along the Morton curve, building the next layer in the BVH by aggregating with the two nodes on either side along the 1-D ordering. However, these methods generally do not construct the BVH tree itself, which is required for Barnes-Hut to build the multipoles and calculate the forces. Keller et al. explored a distributed octree using a similar approach [39].

In Section IV-B, we propose an approach that shares the same fundamental principles as these methods but aggregates pairwise using the Hilbert ordering stored in Gray code [17] to construct a balanced binary tree.

All-Pairs: There is extensive literature on all-pairs algorithm performance for N-body simulations on GPUs, e.g., Nyland et al. improved GPU performance using tiling [40].

VII. Conclusion

This paper presents approaches for writing parallel N-body algorithms using portable ISO C++ on both GPUs and CPUs. These algorithms enable parallel construction and traversal of tree-based irregular data structures on GPU accelerators. The implementations demonstrate strong performance across both CPUs and GPUs without requiring any code modifications, showcasing the performance portability achievable with ISO C++, even for highly irregular algorithms.

Additionally, the implementations are portable across multiple hardware and software stacks, supporting both vendor-specific and vendor-neutral open-source implementations of the C++ standard. This confirms that complex algorithms for irregular data structures can be written in a portable manner, running efficiently on both CPUs and GPUs without sacrificing performance compared to explicit heterogeneous programming models, like SYCL.

The Concurrent Octree algorithm delivered the best performance on many devices. However, on GPUs, it requires *Independent Thread Scheduling* [10], [11] to run in parallel, underscoring the advantage of this hardware feature for irregular algorithms. In scenarios where this is not possible, we have demonstrated an alternative algorithm with weaker forward progress requirements that trades off some portability for lower performance on certain systems.

Acknowledgment

This work is in part funded by the ASiMoV UKRI research project (EP/S005072/1). This project has received funding from the European Union's HE research and innovation programme under grant agreement No 101092877 (SYCLops project). Access to the Intel GPU was thanks to the University of Cambridge and the Cambridge Open Zettascale Lab. Thanks to John Linford for access to the AWS Graviton 4 processor.

References

[1] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, December 1986.

[2] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held in conjunction with Supercomputing (PMBS)*. IEEE, 2022.

[3] "C++ execution Policies." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

[4] J. Hubbard, G. B. Gadeschi, C. Garg, N. Sakharnykh, and F. Oh, "Simplifying GPU Application Development with Heterogeneous Memory Management." [Online]. Available: https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management

[5] A. Alpay and V. Heuveline, "AdaptiveCpp Stdpar: C++ Standard Parallelism Integrated Into a SYCL Compiler," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3648115.3648117

[6] ISO, *ISO/IEC 14882:2020 Information technology — Programming languages — C++*. International Organization for Standardization, 2020.

[7] A. Williams, *C++ Concurrency in Action*. Manning, 2019.

[8] S. J. Pennycook, B. Ashbaugh, J. Brodman, M. Kinsner, S. Larsen, G. Lueck, R. Schulz, and M. Voss, "Towards Alignment of Parallelism in SYCL and ISO C++," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCL '23. New York, NY, USA: Association for Computing Machinery, 2023.

[9] T. Sorensen, L. F. Salvador, H. Raval, H. Evrard, J. Wickerson, M. Martonosi, and A. F. Donaldson, "Specifying and testing GPU workgroup progress models," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021.

[10] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[11] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside Volta: NVIDIA's most advanced data-center GPU," 2017. [Online]. Available: https://developer.nvidia.com/blog/inside-volta/

[12] L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Phys. Rev.*, vol. 159, pp. 98–103, Jul 1967. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.159.98

[13] G. Blelloch and G. Narlikar, *Parallel Algorithms*, ser. Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997, vol. 30, ch. A Practical Comparison of N-Body Algorithms.

[14] A. Alpay, "Teralens." [Online]. Available: https://github.com/illuhad/teralens

[15] ——, "SpatialCL." [Online]. Available: https://github.com/illuhad/SpatialCL

[16] B. Walter, K. Bala, M. Kulkarni, and K. Pingali, "Fast agglomerative clustering for rendering," in *IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 81–86.

[17] J. Skilling, "Programming the Hilbert curve," in *American Institute of Physics*, vol. 707, 2004, pp. 381–387.

[18] W.-C. Lin, S. McIntosh-Smith, and T. Deakin, "Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC," 2024. [Online]. Available: https://arxiv.org/abs/2401.02680

[19] G. B. Gadeschi, "GCC bug 110512: Sequential execution of parallel algorithms when C++20 iterators are used." [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=110512

[20] INTEL, "DPC++ -fsycl-pstl-offload offloads `par_unseq` to SYCL." [Online]. Available: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-1/fsycl-pstl-offload.html

[21] ——, "DPC++ -fsycl-pstl-offload uses TBB or OpenMP on host for par." [Online]. Available: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-1/fsycl-pstl-offload.html

[22] T. Thüring, M. Breyer, and D. Pflüger, "Comparing a Naive and a Tree-Based N-Body Algorithm using Different Standard SYCL Implementations on Various Hardware," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1906—-1917.

[23] NASA's Jet Propulsion Laboratory, "Small-body Database." [Online]. Available: https://ssd.jpl.nasa.gov/tools/

[24] AMD, "HIP does not support Independent Thread Scheduling." [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/latest/reference/cpp_language_extensions.html#independent-thread-scheduling

[25] Khronos, "Each work-item in SYCL is a separate thread of execution, providing at least weakly parallel forward progress guarantees." [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#_forward_progress

[26] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, "3.2 the a100 datacenter gpu and ampere architecture," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 48–50.

[27] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[28] L. van der Maaten, "Barnes-Hut-SNE," 2013. [Online]. Available: https://arxiv.org/abs/1301.3342

[29] M. Burtscher and K. Pingali, "Chapter 6 - An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm," in *GPU Computing Gems Emerald Edition*, 2011, pp. 75–92.

[30] M. Iwasawa, D. Namekata, R. Sakamoto, T. Nakamura, Y. Kimura, K. Nitadori, L. Wang, M. Tsubouchi, J. Makino, Z. Liu, H. Fu, and G. Yang, "Implementation and performance of Barnes-hut n-body algorithm on extreme-scale heterogeneous many-core architectures," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 615–628, 2020.

[31] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji, "A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs—towards cost effective, high performance N-body simulation," *Computer Science–Research and Development*, vol. 24, pp. 21–31, 2009.

[32] Y. Xu, A. Li, and T. Sorensen, "Redwood: Flexible and Portable Heterogeneous Tree Traversal Workloads," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 201–213.

[33] P. Atkinson and S. McIntosh-Smith, "On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 92–106.

[34] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A Survey on Bounding Volume Hierarchies for Ray Tracing," *Computer Graphics Forum*, vol. 40, pp. 683–712, May 2021.

[35] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[36] D. Meister and J. Bittner, "Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 3, pp. 1345–1353, 2018.

[37] C. Benthin, R. Drabinski, L. Tessari, and A. Dittebrandt, "PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited," in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 5, July 2022.

[38] C. Benthin, D. Meister, J. Barczak, R. Mehalwal, J. Tsakok, and A. Kensler, "H-PLOC: Hierarchical Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction," in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, July 2024.

[39] S. Keller, A. Cavelan, R. Cabezon, L. Mayer, and F. Ciorba, "Cornerstone: Octree construction algorithms for scalable particle simulations," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3592979.3593417

[40] L. Nyland, M. Harris, and J. Prins, *Fast N-Body Simulation with CUDA*. Addison Wesley, 2007, ch. 31. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda

APPENDIX
ARTIFACT DESCRIPTION (AD)

A. Overview of contributions & artifacts

1) Paper's Main Contributions: Computational artifacts are created for the two main contributions of this paper:

- The portable ISO C++ implementation of the two novel parallel schemes for Barnes-Hut: *Concurrent Octree* (Section IV-A) and *Hilbert-sorted BVH* (Section IV-B).
- Performance characterization across a wide range of CPUs and GPUs (Section V).

2) Computational Artifacts: The only computational artifact is the git repository at github.com/UoB-HPC/stdpar-nbody which contains:

- The portable ISO C++ implementation of all algorithms evaluated in this paper.
- A container providing the SW environment used when producing the results presented in this paper.
- Scripts to generate the results presented in this paper on a particular system.

That is, this single computational artifact suffices to reproduce the data shown in all Figures of the result's section of this paper (Section V).

B. Artifact Identification

The only computational artifact is the git repository at github.com/UoB-HPC/stdpar-nbody.

1) Relation to Contributions: This artifact reproduces the data shown in the results section of this paper.

2) Expected Results: After executing the artifact and post-processing its output, the data points of the Figures in Section V are shown.

3) Expected Reproduction Time (in minutes):

1) Artifact Setup: 5 minutes.
2) Artifact Execution: 30 minutes.
3) Artifact Analysis: 15 minutes.

4) Artifact Setup:

a) Hardware: The artifact has been tested with the hardware listed in Table I. It may work on other hardware. To reproduce on GPU hardware, the GPU drivers need to be compatible with OneAPI 2024.1 on Intel PVC GPUs, ROCm 6.1.3 on AMD GPUs, or the HPC SDK 24.7 on NVIDIA GPUs.

b) Software: A Linux system with Docker, Python 3, and the Python Package HPCCM is required to build the docker container. The Linux user must be in the docker user group. All other software is embedded within the container.

c) Datasets/Input: The data-sets are automatically generated by the application.

d) Installation & Deployment: After executing the script `./ci/run_docker bench`, an `out_$(hostname)` file is generated containing raw data, which when post-processed with `./ci/data.py out_$(hostname)`, displays a table that contains the single data points of the Figures in Section V.

5) Artifact Evaluation: The workflow for the artifact evaluation consists of two tasks. The first task executed with `./ci/run_docker bench` builds the docker container, compiles the application, and runs the experiments. The second task post-process the produced output file with `./ci/data.py out_$(hostname)`.

6) Artifact Analysis: The second task of the artifact evaluation process the results.