ParustDCD: A Rust Implementation of Parallel Double-Phase Discrete Collision Detection

Dylan Liang, Qiang Fu

1 Introduction

1.1 Double-Phase Discrete Collision Detection

Double-Phase Discrete Collision Detection (DPDCD) lies at the heart of many computer graphics and robotics systems[8]. In its essence, DPDCD splits collision checking into two stages: a broad phase that rapidly culls object pairs unlikely to intersect, and a narrow phase that performs exact intersection or distance queries on the survivors. This two-phase paradigm balances efficiency and accuracy: cheap, approximate tests eliminate most pairs, while expensive but accurate geometric algorithms handle the rest. When dealing with dynamic scene configurations, the algorithm checks collisions at discrete time steps, without modeling continuous motion between frames, thus having the word "discrete" in its name.

1.2 A Brief History of Collision Detection Algorithms

Early collision checking for simple 3D shapes relied on the Separating Axis Theorem (SAT) for accurate collision detection of convex polygons, and Axis-Aligned Bounding Box (AABB) for broad-phase overlapping tests[4]. As scene complexity grew, spatial partitioning techniques, such as uniform grids, octrees, kd trees, Bounding Volume Hierarchy (BVH), and Sweep-and-Prune (SAP), enabled fast lookup of potential contacts by dividing space into cells or hierarchical regions[12].

The advent of computing geometry algorithms for convex shapes—Gilbert–Johnson–Keerthi Algorithm (GJK) and its extension Expanding Polytope Algorithm (EPA)—revolutionized narrow-phase queries[5][17]. By reducing intersection and penetration-depth computations to sequences of support mappings and simplex expansions, these methods enabled robust checks among convex shapes. Libraries such as Parry3D[2] (from the Rapier[3] project) and PhysX's[14] narrow-phase module incorporate GJK/EPA to deliver high-throughput distance queries for rigid bodies, making them useful for modern robotics and computer graphics applications where a single time step in simulation contains millions of queries.

In recent years, DPDCD pipelines have become the de facto standard for both real-time graphics and robot motion planning. The broad-phase filters millions of bodies per frame using BVH or SAP, while the narrow-phase employs GJK/EPA for convex shapes or triangle–triangle intersection for meshes. Continuous collision detection (CCD)[13]—which handles fast-moving objects by considering swept volumes—extends this model, but is orthogonal to most DPDCD pipelines and thus only briefly noted here.

Parallelism has emerged as a critical enabler for scaling DPDCD to large scenes and high update rates. On multi-core CPUs, task-based frameworks distribute BVH traversal or SAP interval tests across threads, approaching linear speedups for moderate workloads[6]. GPU-driven approaches[9][15] launch thousands of threads for uniform triangle tests or grid-based broad-phase culling, but irregular tree walks and dynamic BVH updates pose load-balancing challenges. Hybrid CPU–GPU methods balance host scheduling overhead against device memory bandwidth, yet often require tightly coupled graphics APIs or bespoke kernels.

1.3 Related Implementations in Rust

Rust's[11] emphasis on memory safety and performance makes it attractive for collision checking, but comprehensive and portable DPDCD libraries remain limited.

Parry3D[2] implements GJK and EPA in pure Rust, offering very efficient narrow-phase queries. It serves as the foundation for Rapier's full physics engine, but provides no out-of-the-box DPDCD pipeline, either serial or parallel. Users must manually integrate Parry3D primitives with their own BVH structures and threading model. Assembling the building primitives into a complete parallel DPDCD pipeline requires significant glue code and careful parallelization design to avoid contention.

The Rapier[3] crate itself includes a physics engine paradigm with integrated collision, constraint, and dynamics systems, making it heavyweight for scenarios that require only kinetic collision checks without full dynamics simulation. While Rapier3D leverages Rayon internally, its API is geared toward full-body dynamics. Extracting only the collision subsystem requires disentangling dependencies and absorbing substantial engine overhead.

Experimental Rust-GPU crates explore collision primitives on GPU shaders, but lack a general DPDCD pipeline and are constrained by the current nascent Rust-GPU ecosystem[7].

For the crates mentioned above, developers either take a considerable amount of engineering work, or introduce unnecessary performance overhead, when using them to build a customized parallel DPDCD pipeline for specific applications. To date, to our knowledge, no public rust crate offers a portable parallel DPDCD module in Rust that cleanly unites BVH-based broad-phase, GJK/EPA narrow-phase, and adaptive parallel scheduling.

1.4 Originality and Research Potential of Our Implementation

Our implementation fills this gap. ParustDCD focuses on a portable CPU-parallel DPDCD pipeline in Rust, targeting robotics and graphics scenarios where safety, determinism, and flexibility for secondary development are paramount.

Our implementation can be easily ported to robotics libraries in rust, such as apollo-rust[1], for further customized development. This portability is offered by our light-weighted implementation with minimal dependency on other crates. Our implementation is flexible for developers to add additional traits for specific applications. For example, the computational process of collision detection can be made differentiable easily by adding ad-trait[10] to our implementation, accumulating derivatives along with computations.

By confining heavy computation to CPU cores, our approach avoids GPU-API dependencies and delivers low-latency responses, thus suitable for modern robotics and computer graphics applications that call for high computational performance.

We provide open-source code for our implementation. [1]

2 Methods

We implemented a two-stage collision-detection pipeline in Rust, combining a non-trivially parallelized BVH broad phase with an embarrassingly parallelized GJK narrow phase, all scheduled via Rayon's[16] work-stealing runtime. At the top level, AABB construction, BVH building, broad-phase traversal, and narrow-phase tests are each expressed as parallel operations. These four parallel blocks build the resulting parallel double-phase collision detection algorithm (Alg. 5). We take advantage of zero-cost parallelization abstractions and thread-local buffers so that no explicit synchronizations are programmed and no global locks are used.

[1] https://github.com/chen-dylan-liang/Parallel_Collision_Check

2.1 Parallel BVH Construction

The BVH is built over the AABBs of all objects as in Alg. 1. Given an array of object indices and their AABBs, we first check whether the number of indices n falls below a small leaf cutoff; if so, we emit a leaf node containing those indices. Otherwise, we choose whether to parallelize further based on a build-parallelism threshold.

Algorithm 1 Parallel BVH Construction

procedure PARALLEL_BUILD_BVH(aabb_indices, all_aabbs, cut_off_size)
  $n \leftarrow$ length(aabb_indices) ▷ Number of AABBs to process
  if $n \leq$ cut_off_size **then**
    return BVHLeafNode(aabb_indices, all_aabbs) ▷ Create a leaf node if below cutoff
  end if
  $do\_parallel \leftarrow (n >$ BUILD_PARALLEL_THRESHOLD$)$ ▷ Decide if parallelism is needed
  if $do\_parallel$ **then**
    (axis, midpoint) $\leftarrow$ PARALLEL_LONGEST_EXTENT_AXIS(aabb_indices, all_aabbs)
  ▷ Find the split axis and the midpoint using divide-and-conquer in parallel
    (left, right) $\leftarrow$ PARALLEL_SPLIT_AT_AXIS(aabb_indices, all_aabbs, axis, midpoint)
  ▷ Apply a parallel filter to split at midpoint
    if left is empty **or** right is empty **then**
      return BVHLeafNode(aabb_indices, all_aabbs) ▷ Fallback to leaf node if degeneracy happens
    end if
    $(l, r) \leftarrow$ parallel_join (PARALLEL_BUILD_BVH(left), PARALLEL_BUILD_BVH(right)) ▷ Recursively build left and right in parallel
  else
    (axis, midpoint) $\leftarrow$ SERIAL_LONGEST_EXTENT_AXIS(aabb_indices, all_aabbs) ▷ Compute split axis and midpoint serially
    (left, right) $\leftarrow$ SERIAL_SPLIT_AT_AXIS(aabb_indices, all_aabbs, axis, midpoint)
    if left is empty **or** right is empty **then**
      return BVHLeafNode(aabb_indices, all_aabbs) ▷ Fallback to leaf node if degeneracy happens
    end if
    $l \leftarrow$ PARALLEL_BUILD_BVH(left, all_aabbs, cut_off_size) ▷ Recursively build left subtree
    $r \leftarrow$ PARALLEL_BUILD_BVH(right, all_aabbs, cut_off_size) ▷ Recursively build right subtree
  end if
  node_aabb $\leftarrow l$.union_aabb(r) ▷ Merge bounding boxes
  return BVHInternalNode(node_aabb, l, r) ▷ Return internal node
end procedure

Longest-Extent Axis Computation. Both serial and parallel versions examine the centroids of the selected AABBs to determine which of the x, y, or z dimensions has the largest spread. In the serial case, we scan once over all centroids to track minimum and maximum in each coordinate. The parallel variant performs a divide-and-conquer reduction: splitting the index range among tasks, each computes local min/max per axis, and Rayon merges these in a logarithmic-depth tree, yielding overall work $O(n)$ and span

$O(\log n)$.

Split-at-Axis. Once the longest axis and its midpoint (the average of min and max centroid coordinates) are known, we partition the index array into "left" and "right" children. The serial splitter performs an in-place stable partition in $O(n)$ work. The parallel splitter uses two filters in parallel—each scanning a disjoint slice—collecting indices whose centroid lies below or above the midpoint, then concatenating results. Work remains $O(n)$ but span falls to $O(\log n)$.

Recursive Subtree Assembly. With the split sets in hand, we guard against degeneracy—if one side is empty, we emit a leaf. Otherwise, we spawn two child-build tasks via Rayon's `join`. After both subtrees return, we merge their bounding boxes in constant time to form the parent's AABB and return an internal node. Overall, suppose there are N primitives in the scene, this build has

$$W_{\text{build}}(N) = O(N \log N), \qquad S_{\text{build}}(N) = O(\log^2 N).$$

2.2 Parallel Broad-Phase Collision Checking

Once the BVH is built, we perform a pairwise traversal of the tree against itself to cull non-intersecting node pairs as in Alg. 2.

Algorithm 2 Parallel Broad-Phase Collision Check

```
1: procedure PARALLEL_BROAD_PHASE_CHECK(s1, s2)
2:     Initialize a thread-local storage tl to store buffers   ▷ Each thread has its own local buffer
3:     GATHER(s1, s2, 0, tl) recursively                       ▷ Recursively populate local buffers
4:     Initialize empty list out                               ▷ Final output vector
5:     for all cells in tl do
6:         Extract local vector and extend out                 ▷ Flatten thread-local buffers into one vector
7:     end for
8:     return out                                              ▷ Return collected collision pairs
9: end procedure
```

Thread-Local Gather. We allocate a thread-local vector buffer on each worker to accumulate leaf-leaf collisions without synchronization. The gather (Alg. 3) procedure tests the current pair of nodes for AABB overlap; if they do not overlap, it returns immediately. If both are leaves, it enumerates all index-pairs (i, j) with $i < j$ and appends them to the local buffer. If exactly one is a leaf, it recurses on the leaf versus each child of the internal node. If both are internal and the recursion depth is below a fixed `MAX_DEPTH`, it spawns another two parallel tasks for the four child-pair combinations, with two combinations in each task, to effectively reduce the number of tasks and thus the overhead of spawning; beyond that depth it falls back to serial recursion to avoid excessive overhead.

Algorithm 3 Parallel Broad-Phase Gather

```
1: procedure GATHER(s1, s2, depth, tl)
2:                                                          ▷ Return if no overlap
3:     if ¬ Intersects(s1, s2) then
4:         return
5:     end if
6:     if IsLeaf(s1) and IsLeaf(s2) then
7:         local ← tl.get_or_init()
8:         for all i ∈ LeafIndices(s1) do
9:             for all j ∈ LeafIndices(s2) do
10:                if i < j then
11:                    Push(local,(i, j))
12:                end if
13:            end for
14:        end for
15:    else if IsLeaf(s1) then
16:        (ℓ, r) ← Children(s2)
17:        GATHER(s1, l, depth+1, tl)
18:        GATHER(s1, r, depth+1, tl)
19:    else if IsLeaf(s2) then
20:        (ℓ, r) ← Children(s1)
21:        GATHER(l, s2, depth+1, tl)
22:        GATHER(r, s2, depth+1, tl)
23:    else
24:        (s1l, s1r) ← Children(s1)
25:        (s2l, s2r) ← Children(s2)
26:        if depth < MAX_DEPTH then
27:                                                          ▷ parallel split into four calls
28:            parallel_join(
29:            (GATHER(s1l, s2l, depth+1, tl),
30:            GATHER(s1l, s2r, depth+1, tl))
31:            (GATHER(s1r, s2l, depth+1, tl),
32:            GATHER(s1r, s2r, depth+1, tl)) )
33:        else
34:            GATHER(s1l, s2l, depth+1, tl)
35:            GATHER(s1l, s2r, depth+1, tl)
36:            GATHER(s1r, s2l, depth+1, tl)
37:            GATHER(s1r, s2r, depth+1, tl)
38:        end if
39:    end if
40: end procedure
```

Flattening Results. After the top-level gather returns, each thread's buffer holds a disjoint set of candidate pairs. We then concatenate all thread-local buffers into a single output vector. Suppose P is the number of candidate pairs, this broad-phase traversal does average work and span:

$$W_{\text{traverse}}(N) = O(N \log N + P), \qquad S_{\text{traverse}}(N) = O(\log N)$$

2.3 Parallel Narrow-Phase Collision Detection

The candidate index-pairs are refined via the GJK algorithm to detect actual contacts between the convex shapes as in Alg. 4.

- We `par_iter` over all candidate pairs (i, j), invoking a `gjk_contact` routine on each pair.
- If the returned distance d is zero, we record a contact $\{i, j\}$.

Algorithm 4 Parallel Narrow-Phase Collision Check

procedure PARALLEL_NARROW_PHASE_CHECK(`pairs`, `shapes`, `poses`)
  Assert that `shapes` and `poses` have the same length
  Initialize an empty list `contacts`
  for all $(i, j) \in$ `pairs` **in parallel do**
    Compute $(p, d) \leftarrow$ GJK_CONTACT(`shapes`$[i]$, `poses`$[i]$, `shapes`$[j]$, `poses`$[j]$)
    if $d = 0.0$ **then**
      Add `Contact`$\{i, j\}$ to `contacts`
    end if
  end for
  return `contacts` ▷ Return all detected contacts
end procedure

For each pair is independent, this stage is embarrassingly parallel. The work is $W(P) = O(P\,K)$, where $O(K)$ is the time complexity of running GJK on one pair, and the span is $S(P) = O(K)$.

2.4 Double-Phase Pipeline Orchestration

Putting them all together:

1. **AABB Construction:** Compute each object's AABB from its shape and pose in parallel. ($W = O(N)$, $S = O(1)$)
2. **BVH Build:** Organize AABBs into a tree in parallel. ($W = O(N \log N)$, $S = O(\log^2 N)$)
3. **Broad Phase:** Traverse the BVH against itself, gathering candidate pairs in parallel. ($W = O(N \log N + P)$, $S = O(\log N)$)
4. **Narrow Phase:** Run GJK on each candidate in parallel. ($W = O(P\,K)$, $S = O(K)$)

Algorithm 5 Parallel Double-Phase Collision Detection

procedure PARALLEL_DOUBLE_PHASE_COLLISION_CHECK(`shapes`, `poses`, `cut_off`)
  `aabbs` $\leftarrow$ For each (`shape`, `pose`) in parallel, compute AABB ▷ Construct AABBs
  `indices` $\leftarrow [0, \ldots,$ `aabbs.len()` $- 1]$
  `bvh` $\leftarrow$ PARALLEL_BUILD_BVH(`indices`, `aabbs`, `cut_off`) ▷ Broad Phase: Build BVH
  `pairs` $\leftarrow$ PARALLEL_BROAD_PHASE_CHECK(`bvh`, `bvh`) ▷ Broad Phase: Collision Check
  `contacts` $\leftarrow$ PARALLEL_NARROW_PHASE_CHECK(`pairs`, `shapes`, `poses`) ▷ Narrow Phase: Refine Collision Pairs
  return `contacts`
end procedure

Therefore, for the whole pipeline, we have work and span as follows:

$$W_{\text{total}} = O\big(N \log N + P\,K\big), \quad S_{\text{total}} = O\big(\log^2 N + \max\{\log N, K\}\big).$$

3 Results

3.1 Experimental Setup and Baselines

All experiments were run on a desktop with an Intel i7 CPU (5.4 GHz, 8 performance-cores + 12 efficient-cores, 28 threads total), and 32 GB RAM. We first validated correctness by comparing our double-phase pipeline outputs against `parry`'s narrow-phase implementation for up to $N = 10{,}000$ randomly generated primitives. The primitives are all convex hulls, with randomly picked sizes ranging from 50 vertices to 100 vertices, which are usually seen in robotics applications. The distributions for drawing the samples are all uniform. For baselines we measured:

- **Parry's serial narrow-phase:** `parry`'s GJK implementation run in a single thread.
- **Our serial narrow-phase:** our GJK implementation run in a single thread.
- **Parry's parallel narrow-phase:** `parry`'s GJK implementation with `par_iter`.
- **Our parallel narrow-phase:** our GJK implementation with `par_iter`.
- **Serial double-phase:** serial BVH build + seril BVH traversal + serial GJK, all implemented by ourselves.
- **Parallel double-phase:** full pipeline (parallel BVH build, parallel BVH traversal, parallel GJK) implemented by ourselves.

3.2 Scaling Performance

Figure 1 shows the end-to-end runtimes of the six variants as the number of primitives N increases, plotted on a linear y–axis. We observe:

- The two serial narrow-phase curves (`serial_our_narrow` and `serial_parry_narrow`) grow roughly like $O(N^2)$, diverging against others noticeably once $N \approx 1000$.
- Among the other four methods, three (serial double, parallel our narrow, parallel parry narrow) also exhibit noticeable superlinear growth with $n > 5000$, whereas our parallel double-phase method remains approximately constant runtime in comparison.

Re-plotting on a logarithmic y–axis in Figure 2 makes these separations more pronounced:

- The two serial narrow-phase curves form a tight group at the top.
- The serial double, parallel our narrow, and parallel parry narrow form a middle tier.
- The parallel double-phase curve remains the lowest, demonstrating its superior scalability.

Figure 3 plots the survival ratio $P/\binom{N}{2}$ of candidate pairs after the broad phase. This ratio declines rapidly with N, highlighting the effectiveness of BVH culling. Note also that the serial double-phase method is slightly slower than the parallel narrow-phase variants, since building and traversing the BVH in one thread adds nontrivial overhead, plus the serial narrow phase is slow even the survival pairs are minial.

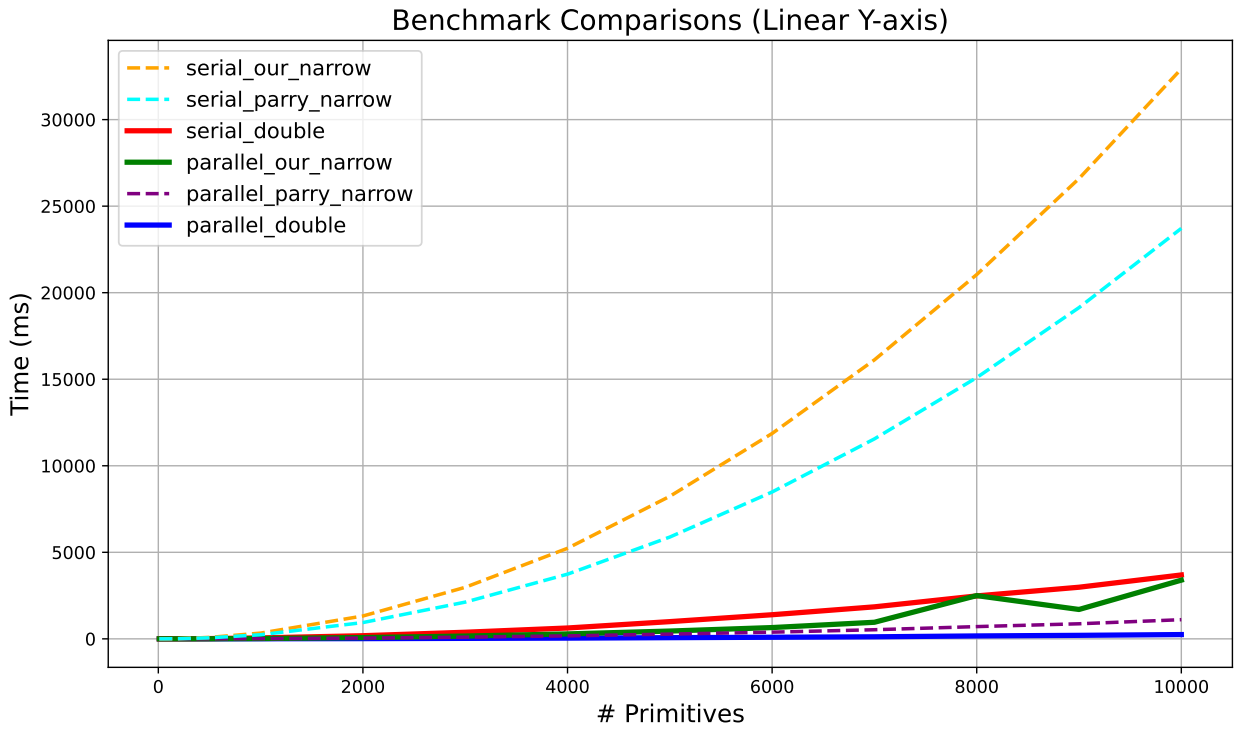

Figure 1: End-to-end runtime vs. number of primitives (y linear).

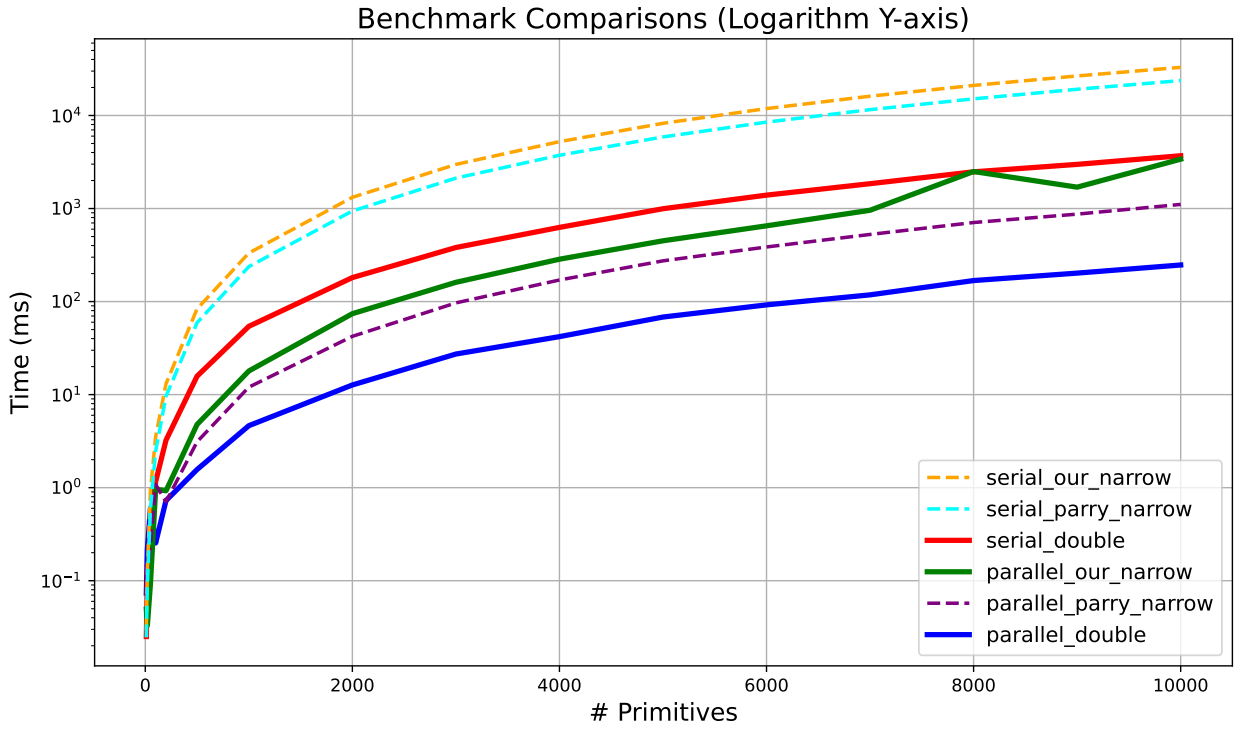

Figure 2: End-to-end runtime vs. number of primitives (y logarithmic).

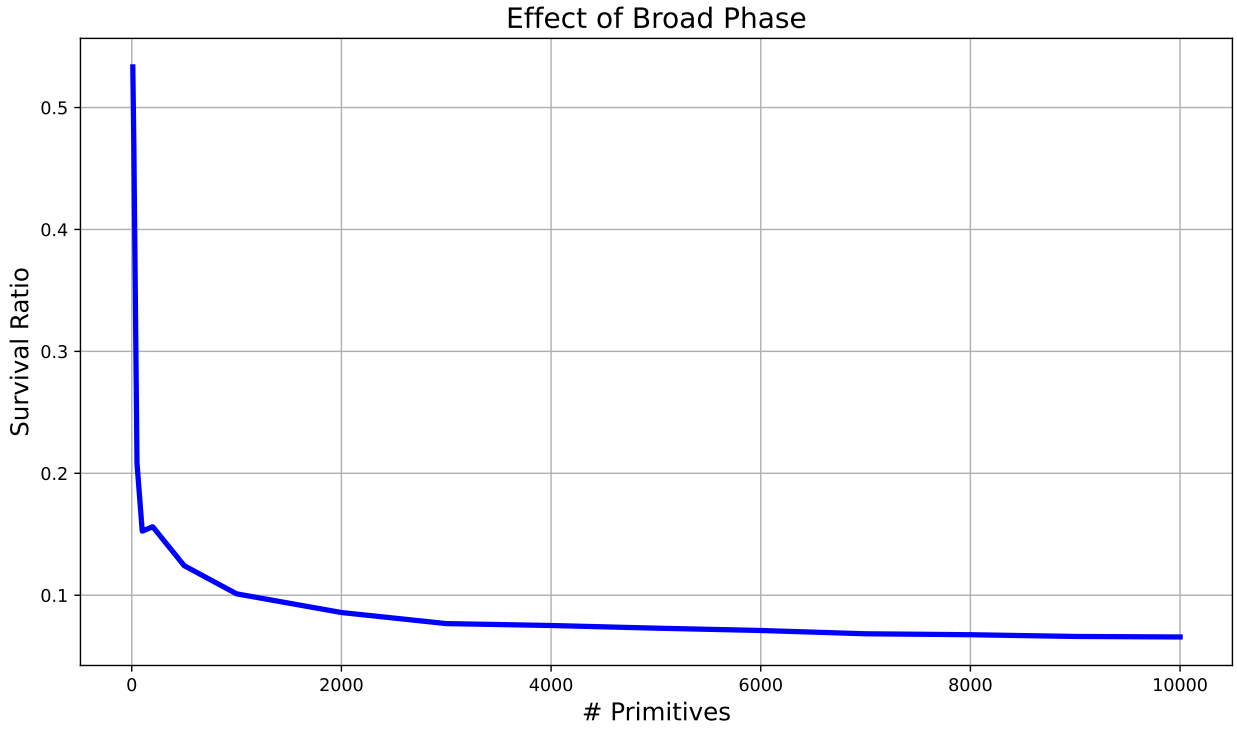

Figure 3: Effect of broad phase: fraction of surviving leaf-leaf pairs $P/\binom{N}{2}$.

3.3 Parallel Speedup

Table 1 summarizes component-wise timings for $N = 10{,}000$ and the corresponding speedup ratios:

Stage	Serial (ms)	Parallel (ms)	Speedup
Build AABBs	1.974	0.546	3.62×
Build BVH	1.928	1.426	1.35×
Broad-Phase Traversal	188.925	46.219	4.09×
Narrow-Phase (GJK)	3549.824	266.638	13.31×

Table 1: Component-wise timings and speedup ratios for $N = 10{,}000$ randomly generated primitives.

The narrow-phase GJK achieves the highest speedup, while BVH building shows more modest gains due to task-spawn overhead on shallow recursions.

3.4 Parameter Tuning

We tuned key broad-phase parameters to our hardware:

- `do_parallel` threshold in BVH build set to $\lceil N/3 \rceil$ primitives.
- `MAX_DEPTH` in `Gather` set to 16.
- Leaf cutoff in BVH build had negligible impact on the parallel double-phase pipeline as shown in Table 2 and was left at default (4).

Leaf Cutoff	1	4	16	64	256	1024
Parallel Double-Phase Algorithm Runtime (ms)	247.11	247.27	247.96	261.28	245.47	245.19

Table 2: Parallel double-phase algorithm runtime on different leaf cutoffs for $N = 10{,}000$ primitives.

These choices balance spawn overhead against available parallelism to maximize throughput on our 28-thread system.

4 Conclusion

In this work we have presented ParustDCD, a lightweight, CPU-parallel double-phase collision-detection pipeline implemented entirely in Rust. By combining a non-trivially parallel BVH-based broad phase with an embarrassingly parallel GJK narrow phase under Rayon's work-stealing scheduler, our implementation achieves outstanding scalability across all stages: AABB construction, BVH build and traversal, and narrow-phase contact queries. Benchmarks on up to 10 000 primitives demonstrate quadratic behavior only in the serial narrow-phase variants, while our fully parallel double-phase method effectively remains constant runtime in comparison. Component-wise speedups top 13× for GJK and exceed 4× for broad-phase traversal on a 28-thread Intel i7 system.

Beyond raw performance, ParustDCD fills a gap in the Rust ecosystem by offering a portable, dependency-minimal library that can be seamlessly integrated into specific systems and applications. We highlight our implementation's potential for future research work in real-time robotics and computer graphics.

References

[1] Apollo Lab, Yale University. *apollo-rust: A Robotics Library in Rust*. GitHub repository. 2025. URL: https://github.com/Apollo-Lab-Yale/apollo-rust.

[2] Dimforge. *Parry3d: 3D collision-detection library for Rust*. Version 0.20.0. Repository: https://github.com/dimforge/parry. 2025. URL: https://docs.rs/parry3d/0.20.0.

[3] Dimforge. *Rapier3d: 3D real-time physics engine for Rust*. Version 0.25.0. Homepage: https://rapier.rs, Repository: https://github.com/dimforge/rapier. 2025. URL: https://docs.rs/rapier3d/0.25.0.

[4] David Eberly. *3D game engine design: a practical approach to real-time computer graphics*. CRC Press, 2006.

[5] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.

[6] Ilan Grinberg and Yair Wiseman. "Scalable parallel collision detection simulation." In: *SIP*. Vol. 7. Citeseer. 2007, pp. 380–385.

[7] Eric Holk et al. "GPU programming in rust: Implementing high-level abstractions in a systems-level language". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 315–324.

[8] S. Kockara et al. "Collision detection: A survey". In: *2007 IEEE International Conference on Systems, Man and Cybernetics*. 2007, pp. 4046–4051. DOI: 10.1109/ICSMC.2007.4414258.

[9] Christian Lauterbach et al. "Fast BVH construction on GPUs". In: *Computer Graphics Forum*. Vol. 28. 2. Wiley Online Library. 2009, pp. 375–384.

[10] Chen Liang et al. *ad-trait: A Fast and Flexible Automatic Differentiation Library in Rust*. 2025. arXiv: 2504.15976 [cs.RO]. URL: https://arxiv.org/abs/2504.15976.

[11] Nicholas D Matsakis and Felix S Klock. "The rust language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 2014, pp. 103–104.

[12] Daniel Meister et al. "A survey on bounding volume hierarchies for ray tracing". In: *Computer Graphics Forum*. Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712.

[13] Quan Nie et al. "A survey of continuous collision detection". In: *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*. IEEE. 2020, pp. 252–257.

[14] NVIDIA Corporation. *PhysX SDK: Real-time multi-physics simulation library*. Version 5.0. Documentation: https://nvidia-omniverse.github.io/PhysX/physx/. 2025. URL: https://github.com/NVIDIA-Omniverse/PhysX.

[15] Jia Pan and Dinesh Manocha. "GPU-based parallel collision detection for fast motion planning". In: *The International Journal of Robotics Research* 31.2 (2012), pp. 187–200.

[16] The Rayon Developers. *Rayon: Data-parallelism library for Rust*. Version 1.10.0. Crate homepage: https://github.com/rayon-rs/rayon. 2025. URL: https://docs.rs/rayon/1.10.0.

[17] Gino Van Den Bergen. "Proximity queries and penetration depth computation on 3d game objects". In: *Game developers conference*. Vol. 170. 2001, p. 209.