1. Introduction

This project originally started as a real-time physics engine designed for computer games, supporting rigid-body simulation, collision detection, and collision response. As the project evolved, the primary goal shifted from interactive game physics toward building a large-scale distributed physics world simulator capable of running efficiently on multi-core and multi-node systems. The final system is no longer focused on game-engine integration, but instead on parallel simulation performance and correctness across distributed subdomains.

The simulator executes several major phases during each simulation step: Broadphase, Narrowphase, Island Construction, and Constraint Solving. Each phase contains components that are naturally data-parallel as well as non-parallelizable substeps. Understanding the dependency patterns and developing non-trivial parallel techniques within these phases are the central research focus of this project.

The project applies two layers of parallelization:

- intra-node OpenMP parallelization within a single physics world
- inter-node MPI domain decomposition, where the world is divided into independent uniform spatial grids, each simulated by a different process

The two layers involve different theoretical assumptions, bottlenecks, and correctness challenges, and therefore require different parallelization strategies.

---

2. Simulation Pipeline

Each timestep consists of the following phases:

Broadphase

The broadphase assigns an Axis Aligned Bounding Box (AABB) to each body and determines potentially overlapping pairs. My implementation uses a quadtree spatial acceleration structure. While most of the pair-generation work can be parallelized, the phase begins by flushing “dirty proxies,” which requires serial re-insertion into the tree. These serial steps were measured separately during benchmarking.

Narrowphase

For each candidate body pair produced by broadphase, the narrowphase performs exact collision testing using geometric methods. In most cases this is trivially parallelizable. However, narrowphase also maintains persistent contact manifolds and must build per-body adjacency information required by the constraint solver. These steps are inherently serial.

Island Construction

Bodies connected through collision contact form a connected component known as an *island*. In classical game engines, the island subsystem exists primarily for sleeping logic—objects that have stopped moving for a sufficiently long period are deactivated to avoid unnecessary solver work. My simulator implements this sleeping infrastructure, but sleeping itself is disabled in my distributed experiments in order to avoid inconsistent activation between MPI ranks. Nevertheless, island decomposition is still essential, because it provides a natural, semantically correct unit of decomposition for solver parallelization and for determining independence during inter-island execution.

Island construction uses a depth-first search and remains serial. However, this serial step is computationally cheap compared to solver iterations and enables effective inter-island parallelization later.

Constraint Solver

Collision resolution is performed through a sequential-impulse formulation, specifically a Projected Gauss–Seidel (PGS) method. Conceptually, each contact constraint defines one row of a large linear complementarity problem. Each PGS iteration processes constraints sequentially and immediately applies velocity corrections, creating inherent read-after-write dependencies that complicate parallelization.

---

3. OpenMP Parallelization

OpenMP is used to parallelize broadphase, narrowphase, and especially the constraint solver. The solver is the most challenging part because PGS is inherently order-dependent: if two constraints involving the same body attempt to update velocities simultaneously, they will observe stale or conflicting values. Fortunately, rigid-body LCPs usually admit multiple feasible solutions, and physical correctness does not require bit-identical deterministic results. Any result that satisfies non-penetration and friction bounds is physically plausible. This observation enables parallelization that trades strict determinism for speed.

I implemented and evaluated two strategies:

3.1 Inter-Island Parallelization

Since islands do not interact, each island can be solved independently on separate threads. This approach is widely used in industrial engines (e.g., Box2D, Bullet). Inter-island parallelization gives the largest speedup and is relatively simple to implement.

However, the approach is sensitive to load imbalance. Realistic scenes often contain one dominant island plus many tiny isolated ones. Overall parallel scalability is bounded by the size of the largest island.

3.2 Intra-Island Parallelization via Graph Coloring

For each island, the constraint graph treats bodies as nodes and constraints as edges. Two constraints conflict if they affect the same body. A greedy graph coloring assigns non-conflicting constraints to color groups. All constraints within one color can be processed in parallel, while color groups are executed sequentially.

My implementation packs constraints across islands by color (instead of per island) so that each color step launches large parallel batches.

However, this method introduces additional overhead. Coloring itself typically consumes around 3% of serial frame time. I attempted parallel execution within a single island using a greedy graph coloring algorithm. My coloring produces very few colors (typically 5–7 in my dense scenes). After parallel experiments, the performance results were disappointing:

- Inter-island only: ~5× speedup in typical medium scenes
- Intra+Inter: ~3.9× (average)
- In sparse scenes: Inter ≈ 5.2×, while Inter+Intra falls to 3.8×
- Only in highly aggregated single island cases did intra-island help: 3.2× vs 2.7×

Therefore, intra-island parallelization only improves performance in artificially extreme cases where a single enormous island dominates the simulation domain. In every realistic scenario, coloring overhead dominates, and performance is strictly worse. My final solver disables coloring entirely and uses inter-island parallelization exclusively.

To validate correctness, I visually inspected physical behavior and monitored constraint violation after solving. Neither method produced incorrect results under test conditions.

---

4. MPI Parallelization

The outer parallel layer uses fixed spatial domain decomposition. The global simulation region is divided into uniform grid cells, and each cell is permanently assigned to one MPI rank. Every rank runs a full local physics step (broadphase, narrowphase, solver) on its own bodies. Bodies are assigned to cells using their center-of-mass location.

The main challenge arises when bodies lie near cell boundaries or move across them. These cases are handled by two communication mechanisms: ghost exchange (temporary overlapping bodies) and migration (permanent ownership transfer).

4.1 Ghost Regions

Each cell maintains a ghost region that slightly exceeds its physical boundary. When a body's AABB overlaps a neighboring cell, the owning rank sends a compact serialized copy of that body to the

neighbor. The neighbor either creates a new ghost body or updates its existing copy.

Ghost bodies are treated as read-only—they participate in broadphase and narrowphase, but do not modify velocities or integration results. Only the owner rank updates real body state during the solver. After each timestep, ghosts that were not refreshed are removed. This guarantees that each body has exactly one authoritative owner, while other ranks only hold temporary views.

4.2 Body Migration

If a body's center of mass moves into another grid, the original owner sends a final serialized state and destroys its local instance. The receiving rank deserializes the body and becomes the new owner. This ensures each real body lives exactly once in the distributed domain and that global IDs remain consistent.

4.3 Communication Pattern

Both ghost exchange and migration are performed once per timestep using MPI collective calls (Alltoallv). All body data is first serialized into byte buffers on the sender side and fully reconstructed on the receiver side. Exchanging only once per step significantly reduces communication overhead, at the cost of slightly slower convergence for collisions that span multiple grids. In practice, cross-grid collisions were rare (<3% in my experiments), so a single communication step was an effective and simple design choice.

4.4 Correctness

To validate correctness, I implemented a single-process version that emulates the same communication flow internally and provides full rendering. I compared body trajectories from the MPI run against this emulation by logging body states each step and replaying them offline. Both simulations produced matching results. Near very dense multi-grid collisions, some jitter is visible due to delayed synchronization, which is expected. More frequent communication (e.g., every solver iteration) would reduce this but would also increase runtime.

---

5. Experimental Setup

The final experiments use a single large 2-D world with coordinates from (−10, −10) to (1200, 1200). The world is split into a 2 × 2 grid to create regions of higher contact density near the internal boundaries. Each subregion is surrounded by four static walls of thickness 2, so bodies cannot leave the global box. In total the scene contains 700,000 dynamic bodies and 16 static walls.

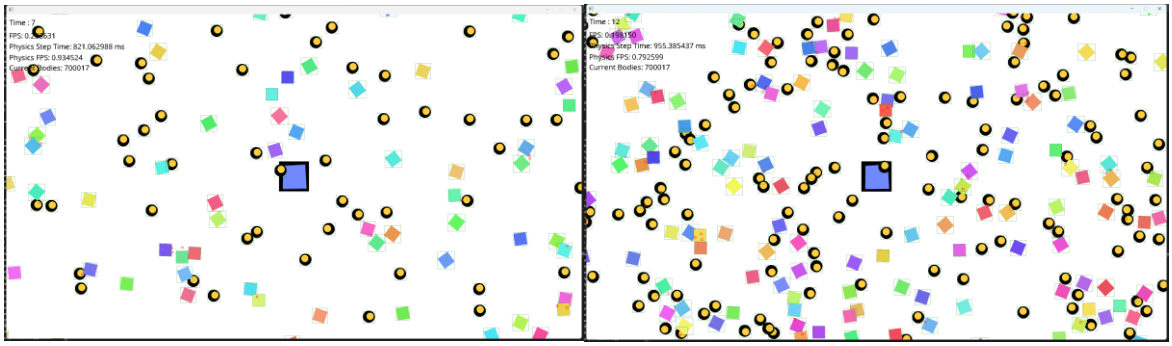
Each dynamic body is either a box (width and height 0.2) or a circle (radius 0.2), chosen with equal probability. Initial linear velocities are sampled uniformly from 0 to 20, and initial angular velocities from 0 to 5 radians per second. The simulation runs for 600 fixed-time steps at 60 Hz, corresponding

to 10 seconds of physical time. These parameter choices were driven by the practical constraint that the single-core baseline (1 MPI rank, 1 OpenMP thread) must complete within the 30-minute job limit on Great Lakes.

I evaluate two classes of scenes:

- Balanced scene ("NORMAL") – bodies are initially placed evenly across the domain, with roughly uniform local density.
- Skewed scene ("SKEWED") – starting from the balanced placement, I remap each body's relative x and y coordinates by raising them to the power 1.2. This transformation pushes about 30% of the bodies into the upper-left quadrant and 20% into the bottom-right. Visually this produces much denser clusters and large contact islands near the top left, significantly increasing the computational cost.

Below figures shows a qualitative comparison of the balanced and skewed scenes taken at the same world position (around (50, 50)). Temporary penetrations and red contact markers are expected in these screenshots; they arise because the physics is stepping at 60 Hz while rendering runs at a much lower frame rate.



All timing numbers below are measured over the entire 600-step run and reported as averages per step.

---

6. Overall Scaling Results

Below figures summarize the end-to-end performance across a grid of MPI ranks and OpenMP thread counts, for both balanced and skewed scenes, these tables are collected as csv and plotted by ChatGpt.

Below images are: (1)simulation time of NORMAL(in seconds), (2)speedup of NORMAL, (3)simulation time of SKEWED(in seconds), (4)speedup of SKEWED. I collected all running time of each substep of the simulation in each rank, due to the length issue I cannot show it in this report, just ask me if you want.

num_ranks \ omp_threads	1	2	4	9	18	36
1	1628.848	886.097	539.706	341.564	267.704	276.697
2	849.436	482.106	285.840	179.871	129.628	-
4	545.041	233.005	147.878	85.827	63.350	-
8	200.999	168.881	96.772	54.089	38.605	-
9	154.536	92.853	55.386	-	-	-
16	91.211	54.183	34.031	26.528	-	-
18	123.721	68.682	39.083	-	-	-
36	29.786	17.064	12.460	-	-	-
72	12.836	7.289	-	-	-	-
144	6.293	-	-	-	-	-

num_ranks \ omp_threads	1	2	4	9	18	36
1	1.000	1.838	3.018	4.769	6.085	5.887
2	1.918	3.379	5.698	9.056	12.566	-
4	2.988	6.991	11.015	18.978	25.712	-
8	8.104	9.645	16.832	30.114	42.193	-
9	10.540	17.542	29.409	-	-	-
16	17.858	30.062	47.863	61.400	-	-
18	13.165	23.716	41.677	-	-	-
36	54.685	95.454	130.727	-	-	-
72	126.898	223.482	-	-	-	-
144	258.838	-	-	-	-	-

num_ranks \ omp_threads	1	2	4	9	18	36
1	1634.275	925.670	582.236	370.567	268.174	274.246
2	855.674	480.728	289.421	180.381	171.947	-
4	577.459	234.736	196.812	121.592	88.919	-
8	337.043	169.046	107.368	67.404	49.174	-
9	272.927	122.640	83.390	-	-	-
16	91.350	76.235	59.416	38.448	-	-
18	115.876	73.472	44.212	-	-	-
36	66.433	36.224	25.472	-	-	-
72	35.563	19.552	-	-	-	-
144	18.852	-	-	-	-	-

num_ranks \ omp_threads	1	2	4	9	18	36
1	1.000	1.766	2.807	4.410	6.094	5.959
2	1.910	3.400	5.647	9.060	9.505	-
4	2.830	6.962	8.304	13.441	18.379	-
8	4.849	9.668	15.221	24.246	33.235	-
9	5.988	13.325	19.594	-	-	-
16	17.890	21.437	27.506	42.507	-	-
18	14.104	22.244	36.964	-	-	-
36	24.600	45.116	64.160	-	-	-
72	45.955	83.584	-	-	-	-
144	86.689	-	-	-	-	-

6.1 Baseline and Superlinear Speedup

The slowest configuration is a single MPI rank with a single OpenMP thread. In the balanced scene this run takes about 1629 seconds for 600 steps (roughly 27 minutes).

By contrast, the fastest balanced configuration tested (144 ranks × 1 thread) finishes in about 6.3 seconds, for a speedup of roughly 259× relative to the baseline. The skewed scene shows a similar pattern: 1 rank × 1 thread needs about 1634 seconds while 144 ranks × 1 thread completes in 18.8 seconds, a speedup of about 86×.

These speedups are superlinear with respect to the number of ranks. For example, in the balanced case the ideal linear scaling from 1 to 72 ranks would predict about a 72× speedup, but the measured speedup (~127×) is significantly higher. This behavior is a key indicator that domain decomposition is not just distributing work but actually reducing the intrinsic difficulty of the broadphase and solver in each subdomain.

The physical reason is that broadphase and contact generation do not scale linearly with body count. In a monolithic tree, some "fat" AABBs end up near the top of the tree and must be tested against many query volumes; as the body count grows, more such problematic nodes appear, and the average query cost grows faster than linearly. Splitting the world into many smaller trees dramatically reduces the probability of these worst-case interactions and shrinks the typical tree depth, so the per-body cost of broadphase and contact generation drops in addition to the parallel speedup.

6.2 MPI vs OpenMP: Where the Parallelism Comes From

The tables clearly show that most of the scalability comes from MPI domain decomposition, rather than from OpenMP alone.

- In the balanced scene, the total time for 1 rank improves from 1629 s (1 thread) to 539 s (4 threads) and then flattens around 270–280 s for 18–36 threads. That is roughly a 3× speedup from 1 to 4 threads and only about 6× at 18 threads, with diminishing returns beyond that.
- In contrast, moving from 1 rank, 1 thread (1629 s) to 16 ranks, 1 thread (91 s) yields an 18× speedup purely from domain decomposition.

Once each rank receives a smaller subdomain, even a modest number of threads (2–4) is enough to saturate the available parallelism inside each local world. Pushing the thread count higher mostly increases overhead and exacerbates the order-dependence of the PGS solver, which leads to weaker scaling (and this is probably also because of parallelization hyperparameter tune, as I didn't spend too much time on this, and some other guesses are mentioned in more discussion part later).

The skewed scene stresses the system with heavier clustering and larger islands near some grid boundaries. Here, OpenMP scaling is slightly weaker and high-thread configurations experience more load imbalance. However, the dominant trend remains: increasing the number of ranks consistently

reduces runtime even under skewed load, demonstrating that both the domain decomposition and the ghost/migration protocol are robust.

6.3 Effect of Skewed Distribution

Comparing the "NORMAL" and "SKEWED" tables, the skewed configuration always runs slower at the same rank/thread combination, which is expected because:

- The upper-left region contains a much larger, more tightly packed island, leading to more contacts and more PGS iterations.
- Grid boundaries in that region see many cross-cell collisions, which stress the ghost exchange mechanism and increase the cost of broadphase and narrowphase near the boundary.

However, the overall scaling trends remain similar:

- The 1-rank runtimes increase by only a few percent (1629 s vs 1634 s).
- At 16 ranks × 1 thread, total times are essentially unchanged (about 91 s in both normal and skewed scenes).
- The MPI + OpenMP configurations still achieve large speedups; for example, 36 ranks × 1 thread yields about 29.8 s in the balanced case and 66.4 s in the skewed case, while 36 ranks × 2 threads drops to around 17.1 s and 36.2 s respectively.

This indicates that the domain decomposition is handling realistic load imbalance reasonably well; MPI remains effective even when one region has much heavier collision density than others.

But when MPI rank num becomes larger the speedup is significantly lower than balanced scene. At small rank counts (e.g., 2, 4, or 8 ranks), the dense cluster in the upper-left region is mostly contained inside one or two cells; they are heavier than others, but still have relatively large volumes. As I increase the number of ranks, that same dense cluster is cut across many small cells. Several things happen:

- The heavy region now spans more ranks, so more ghosts are required around those borders.
- Many of these cells have large partial islands lying exactly on the boundaries, so they depend more on ghost data and thus suffer more from delayed synchronization.
- The time per step is increasingly dictated by the slowest few cells that own the densest pieces of the cluster, while other ranks may be underutilized.

Consequently, the skewed scene benefits from MPI at low and moderate rank counts, but its speedup curve flattens earlier and becomes noticeably worse than the balanced case as the rank count grows. This behavior matches the intuition that domain decomposition works best when the workload is spatially uniform; when most of the work lives inside one corner of the grid, slicing the grid more finely eventually just spreads that hot area across more ranks and increases communication.

7. Substep Analysis: Broadphase and Narrowphase

To better understand which components limit scalability, I break down the per-step timing into broadphase, narrowphase, solver, and other components. Here I focus specifically on broadphase pair generation and narrowphase collision detection, for the simpler case of 1 MPI rank with varying OpenMP threads. All numbers below are averages per step over the 600-step run.

With 1 MPI rank and 1 OpenMP thread, the running time proportion is :

```
Total simulation time = 1628.848364 seconds
===== Rank 0 timing stats =====
  IntegrateVelocities  : avg = 30.023369 ms over 600 samples
  Solver Initialize    : avg = 65.460990 ms over 600 samples
  Narrowphase Generate Manifold : avg = 32.720710 ms over 600 samples
  Solver solve         : avg = 357.508905 ms over 600 samples
  NarrowPhase          : avg = 81.828097 ms over 600 samples
  NarrowPhase Collision Detection : avg = 48.544293 ms over 600 samples
  BroadPhase           : avg = 2034.696994 ms over 600 samples
  BuildIslands         : avg = 37.682704 ms over 600 samples
  Broadphase pair generation : avg = 1955.482863 ms over 600 samples
  IntegrateForces      : avg = 25.068719 ms over 600 samples
```

With 16 MPI rank and 9 OpenMP thread, the running time proportion is:

```
Total simulation time = 26.528495 seconds
===== Rank 0 timing stats =====
  Narrowphase Generate Manifold : avg = 1.139455 ms over 598 samples
  IntegrateVelocities  : avg = 1.944410 ms over 600 samples
  Solver Initialize    : avg = 1.562273 ms over 600 samples
  NarrowPhase Collision Detection : avg = 0.315998 ms over 598 samples
  Solver solve         : avg = 1.464434 ms over 600 samples
  NarrowPhase          : avg = 1.480978 ms over 600 samples
  BroadPhase           : avg = 14.012700 ms over 600 samples
  BuildIslands         : avg = 1.689743 ms over 600 samples
  Broadphase pair generation : avg = 10.977189 ms over 600 samples
  IntegrateForces      : avg = 1.313764 ms over 600 samples
```

With 144 MPI rank and 1 OpenMP thread, the running time proportion is:

```
Total simulation time = 6.292930 seconds
===== Rank 0 timing stats =====
  Narrowphase Generate Manifold : avg = 0.115706 ms over 598 samples
  IntegrateVelocities  : avg = 0.183496 ms over 600 samples
  Solver Initialize    : avg = 0.272058 ms over 600 samples
  NarrowPhase Collision Detection : avg = 0.318811 ms over 598 samples
  Solver solve         : avg = 0.709277 ms over 600 samples
  NarrowPhase          : avg = 0.439641 ms over 600 samples
  BroadPhase           : avg = 3.945658 ms over 600 samples
  BuildIslands         : avg = 0.118904 ms over 600 samples
  Broadphase pair generation : avg = 3.357612 ms over 600 samples
  IntegrateForces      : avg = 0.138772 ms over 600 samples
Rank 1: OpenMP enabled   OPENMP = 201511
```

With 1 MPI rank and 18 OpenMP thread, the running time proportion is:

```
Rank 0: OpenMP threads = 18
Total simulation time = 267.703681 seconds
===== Rank 0 timing stats =====
  IntegrateVelocities  : avg = 30.643848 ms over 600 samples
  Solver Initialize    : avg = 33.476830 ms over 600 samples
  Narrowphase Generate Manifold : avg = 33.402197 ms over 600 samples
  Solver solve         : avg = 34.709363 ms over 600 samples
  NarrowPhase          : avg = 37.258721 ms over 600 samples
  NarrowPhase Collision Detection : avg = 3.305698 ms over 600 samples
  BroadPhase           : avg = 164.016838 ms over 600 samples
  BuildIslands         : avg = 36.944831 ms over 600 samples
  Broadphase pair generation : avg = 115.204879 ms over 600 samples
  IntegrateForces      : avg = 25.410776 ms over 600 samples
```

With 1 MPI rank and 36 OpenMP thread, the running time proportion is :

```
Total simulation time = 276.697425 seconds
===== Rank 0 timing stats =====
  IntegrateVelocities  : avg = 41.783324 ms over 600 samples
  Solver Initialize    : avg = 36.336824 ms over 600 samples
  Narrowphase Generate Manifold : avg = 41.565806 ms over 600 samples
  Solver solve         : avg = 30.355357 ms over 600 samples
  NarrowPhase          : avg = 45.107727 ms over 600 samples
  NarrowPhase Collision Detection : avg = 2.824054 ms over 600 samples
  BroadPhase           : avg = 124.417267 ms over 600 samples
  BuildIslands         : avg = 48.184567 ms over 600 samples
  Broadphase pair generation : avg = 61.160954 ms over 600 samples
  IntegrateForces      : avg = 32.871062 ms over 600 samples
```

Where Broadphase pair generation and Narrowphase Collsion Detection and Solver solve are parallelized parts.

7.1 Broadphase Pair Generation

In the balanced scene, the monolithic single-thread configuration (1 rank, 1 thread) spends about 2.03 seconds per step in the overall broadphase and about 1.96 seconds specifically in "broadphase pair generation". With more threads:

- 1 thread: broadphase ≈ 2035 ms; pair generation ≈ 1955 ms
- 2 threads: broadphase ≈ 1002 ms; pair generation ≈ 940 ms
- 4 threads: broadphase ≈ 539 ms; pair generation ≈ 488 ms
- 9 threads: broadphase ≈ 264 ms; pair generation ≈ 216 ms
- 18 threads: broadphase ≈ 164 ms; pair generation ≈ 115 ms
- 36 threads: broadphase ≈ 124 ms; pair generation ≈ 61 ms

Comparing to the 1-thread baseline, broadphase pair generation achieves approximately:

- 2× speedup at 2 threads,
- 4× at 4 threads,
- 9× at 9 threads
- about 17× at 18 threads,
- and over 32× at 36 threads.

This is significantly better than the speedup of the total program for the same configurations. It confirms that broadphase is highly parallelizable and that the quadtree traversal work distributes nicely across threads. The remaining limitations at high thread counts are mostly due to:

- shared writes when inserting/updating proxies at the beginning of the broadphase step (which are serial in the current implementation),
- and to the fact that broadphase is only one component of the full step; solver and other phases do not scale as well and thus dominate beyond a certain number of threads.

For the skewed scene, broadphase timings are very similar at the 1-rank level: the single-thread broadphase is around 1995 ms per step and drops to about 124 ms at 36 threads. The speedup pattern is nearly identical, which implies that within a single rank the quadtree broadphase handles density variations gracefully; the major skew-related slowdown appears only once we introduce multiple ranks and communication.

7.2 Narrowphase Collision Detection

The narrowphase stage contains two main parts: collision detection between candidate pairs and manifold generation/maintenance. Here I focus on the "NarrowPhase Collision Detection" timing, which measures the pure geometric tests between body shapes.

In the balanced scene at 1 rank, collision detection scales as follows:

- 1 thread: ≈ 48.5 ms per step
- 2 threads: ≈ 24.9 ms
- 4 threads: ≈ 12.4 ms
- 9 threads: ≈ 5.9 ms
- 18 threads: ≈ 3.3 ms
- 36 threads: ≈ 2.8 ms

Relative to the 1-thread baseline, this corresponds to roughly:

- 1.95× speedup at 2 threads,
- 3.9× at 4 threads,
- 8.2× at 9 threads,
- about 14.7× at 18 threads,
- and over 17× at 36 threads.

This is already very good: collision tests are embarrassingly parallel over contact pairs, and there is very little shared state other than read-only body transforms and shapes. The residual cost at high thread counts is mostly scheduling overhead and cache contention.

The skewed scene behaves similarly. Single-thread detection is slightly more expensive (around 53 ms per step), but drops to approximately 3.7 ms at 36 threads, again giving near-linear scaling. This is consistent with the fact that the number of potential pairs is higher in dense regions, but each pair test itself is independent and parallel-friendly. (Again, bad speedup at high thread num is still probably because of hyperparameter tuning)

7.3 Comparison: Broadphase/Narrowphase vs Solver

When contrasted with the solver timings, the broadphase and narrowphase components clearly scale better:

- In the balanced scene, "Solver solve" decreases from about 358 ms at 1 thread to about 35 ms at 18 threads and ~30 ms at 36 threads, which is roughly an 11× speedup at best – good, but not as strong as the 32× speedups seen in broadphase pair generation and collision detection.
- The weaker scaling is expected because PGS updates shared body velocities in place and is fundamentally order-dependent; conflicts between threads introduce both contention and algorithmic noise.

As a result, at low thread counts the solver dominates the step time, while at high thread counts the total runtime is increasingly limited by the less scalable components such as solver initialization, island construction, and MPI communication (in multi-rank runs). This also explains why adding more than 18 threads per rank provides only marginal improvement in overall runtime even though broadphase and collision detection themselves continue to scale. Another reason is that each GreatLake CPU has only 18 cores, so using more than 18 threads will increase switching time and

even increase serial part running time as shown above.

---

8. More Discussion

The results highlight that domain decomposition, not just intra-node threading, is the main source of scalability in this simulator. Splitting the world into many MPI ranks changes both the amount of work per core and the structure of the contact graph.

First, the decomposition naturally splits large contact islands near grid boundaries into several smaller “partial islands”. In the monolithic solver, a huge island with thousands of bodies must be processed as one connected constraint graph; with decomposition, the island is cut where it crosses grid boundaries, and each rank only solves the subset of constraints involving its owned bodies and their ghosts. This significantly reduces solver cost per rank, and is one of the reasons for the superlinear speedups: the problem itself becomes easier after the split, in addition to running on more cores.

This design has a trade-off. Cutting islands across cell boundaries reduces convergence quality near the borders, because each rank only sees slightly outdated ghost states from the previous time step. In principle, more and more splits could lead to worse behavior as the grid gets finer. In practice, however, the fraction of collisions that actually touch cell boundaries remains extremely small. Even with 144 ranks (12 × 12 grid), less than about 3% of all contacts occur in the border regions. As a result, the slower convergence is localized and visually acceptable in my experiments; most of the world behaves as if it were solved monolithically.

As the number of ranks continues to grow, the communication pattern imposes a different limitation. Each new split increases the surface-to-volume ratio of a cell: there are more boundaries per body, which means more ghosts. That has two consequences:

- Ghost count grows, so serialization and deserialization cost increases.
- Communication volume increases, because larger fractions of the local bodies need to be sent as ghosts every step.

At some point, these effects will dominate and the speedup will start to drop. In the current experiments the system is clearly not at that point yet. Even at 144 ranks × 1 thread (cells of size roughly 100 × 100 units with only about 5,000 bodies each), the simulator still achieves around 259× speedup in the balanced scene relative to the 1-rank, 1-thread baseline. The plateau is instead imposed by the available Great Lakes resources: 144 cores is the largest configuration I was allowed to allocate.

On the OpenMP side, the scaling limits are easier to interpret in terms of Amdahl’s Law. Some parts of the pipeline are fundamentally serial or only weakly parallelizable: flushing dirty proxies in the broadphase, island construction, solver setup, and various bookkeeping tasks. However, even if I

consider *only* the explicitly parallelized regions, the speedups are still sublinear. For example, broadphase pair generation and narrowphase collision detection scale very well, but the PGS solver saturates around 10–12× speedup from 1 to 18 threads and improves little beyond that. This is likely due to a mixture of factors:

- Memory bandwidth and cache effects, since each constraint update touches multiple bodies and their velocity data, these data are not contiguous in memory.
- Synchronization and false sharing, as many constraints in dense regions repeatedly update the same bodies.
- Algorithmic order dependence of PGS, which makes random or highly parallel update orders less efficient than carefully ordered serial sweeps.

These issues are inherent to the sequential-impulse formulation and are harder to remove than the more obvious outer-loop parallelism.

Overall, the experiments show that:

- Hybrid MPI + OpenMP parallelization is effective, but the dominant gains come from MPI domain decomposition and the resulting structural simplifications of the broadphase and solver.
- Ghost-based Schwarz iteration is sufficient for correctness in this application, because cross-border collisions are rare and the reduced convergence near boundaries affects only a tiny portion of the world.
- The current configuration is still on the "good" side of the communication/ghost-overhead curve; the bottlenecks are largely resource limits and solver scalability, not the domain-decomposition design itself.

These observations suggest several natural directions for future work, such as adaptive partitions that follow dense regions, alternative solvers with better parallel convergence properties, or multi-iteration communication patterns that selectively improve boundary convergence without incurring the cost of global synchronization after every iteration. Some research tried to use union find to parallelize island construction part, which is also worth exploring.

9. Reference

[1] Erin Catto. "Iterative Dynamics with Temporal Coherence". GDC 2009.

[2] P. L. Lions. "On the Schwarz alternating method". *III International Conference*, 1990

[3] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2004