Object-Centric Parallel Rigid Body Simulation With Timewarp

John Koenig, Ioannis Karamouzas, Stephen J. Guy

Department of Computer Science and Engineering

University of Minnesota

{koenig, ioannis, sjguy}@cs.umn.edu

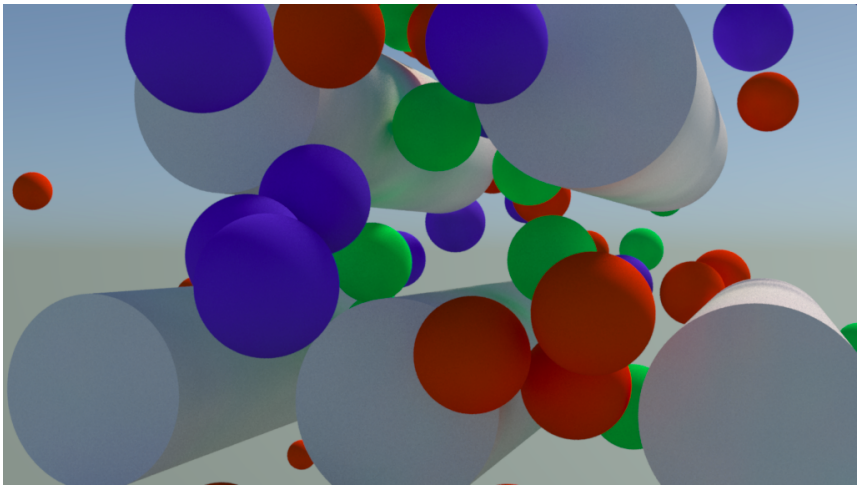


Figure 1: *A dynamic scene with two-hundred spheres falling onto five static cylinders. Our simulation approach is object-centric, with each object modeled as a soft-thread and simulated independently. This results in scalable performance, achieving a 5-6X simulation speedup on eight cores and 9-10X speedup on 16 cores.*

Abstract

We present an object-centric formulation for parallel rigid body simulation that supports variable length integration time steps through rollbacks. We combine our object-centric simulation framework with a novel spatiotemporal data structure to reduce global synchronization and achieve interactive, real-time simulations which scale across many CPU cores. Additionally, we provide proofs that both our proposed data structure and our object-centric formulation are deadlock-free. We implement our approach with the functional programming language Erlang, and test the performance and scalability of our method over several scenarios consisting of hundreds of interacting objects.

CR Categories: I.6.8 [Simulation and Modeling]: Types of Simulation—Parallel

Keywords: interactive real-time simulation, scalability, timewarp

1 Introduction

The number of computational cores available to the average consumer is on the rise. This is partly due to the desire of chip manufacturers to maintain Moore's Law, but also to the recent affordability of CPU accelerators [Tilera 2007; Parallela 2012]. In order for game simulations to make use of modern, potentially heterogeneous platforms, methods for scalable and efficient real-time simulation are required. However, several challenges exist when applying the traditional frame-centric simulation pipeline in a parallel setting. While it is easy to spread integration tasks across several cores, collision detection/resolution becomes a primary challenge, as traditional spatial data structures result in overly restrictive synchronization, ultimately bottlenecking performance.

These factors motivate us to introduce an object-centric simulation model along with a companion data structure which capitalizes on spacetime assumptions common to physical simulations in order to achieve a high-degree of parallelism. Runtime is improved further by permitting objects to take dynamic timesteps that vary based not only on individual state but also circumstances unfolding in the surrounding scene.

Main Results. In this paper, we propose a framework for *parallel* rigid body simulation on multi-core machines. The contributions realized by this work are three-fold. First, we introduce an object-centric model that simulates each scene object independently and only enforces a minimal amount of global synchronization, thus allowing excellent scalability across many cores. Second, we propose a novel spatiotemporal data structure, which is designed to operate in a parallel setting and capitalizes on locality assumptions common to physically-based simulations. Third, we formally provide guarantees about the deadlock-freeness of our proposed data structure and object-centric formulation.

Organization. The rest of this paper is organized as follows. Section 2 highlights previous work in the area of parallel simulation and Section 3 presents a high level overview of our framework. A detailed explanation of our spatiotemporal data structure and object-centric formulation are provided in Sections 4 and 5, respectively, whereas experiments to test the performance of our tech-

nique are presented in Section 6. Finally, some conclusions and plans for further research are discussed in Section 7.

2 Related Work

Previous work in the area of parallel simulation will be described in this section. As our method builds directly on timewarp, this section has been divided into two sections: approaches using timewarp and other methods for parallelism in physical simulations.

2.1 Timewarp

Timewarp was originally presented in [Jefferson 1985] as a method alleviating unnecessary synchronization for distributed systems handling discrete events. Application of timewarp to rigid body simulations was first presented in [Mirtich 2000]. This work placed timewarp within a uniprocessor context and extended it to account for rigid body dynamics in order to prevent unnecessary synchronization between objects during narrowphase. Mirtich lays the groundwork for applying timewarp within a parallel context, however his implementation was restricted to a single processor.

Most recently Ainsley et al. [2012] showed how to apply timewarp to Asynchronous Contact Mechanics (ACM) [Harmon et al. 2009] in order to achieve provably-correct, parallel simulations which are computed significantly faster than ACM. This work, however, focused on parallelism at the frame level and obtains a realism which is not geared towards an interactive setting.

Our work extends Mirtich's original work into a multi-processor, shared-memory context. To accomplish this, we utilize an object-centric simulation model, as compared to [Ainsley et al. 2012]. In addition, we introduce a novel spatiotemporal hash table to achieve scalable real-time simulation.

2.2 Other Approaches

Timewarp has also been applied to domains beyond rigid body simulation. For example, Zheng and James [2011] used timewarp to simulate sound propagation through a scene.

Dequidt et al. [2004] presented a framework in which each simulation object is modeled autonomously. Each object governs its own simulation loop and interactions between objects are driven by zone agents which arbitrate a discrete cell of simulation space. Dequidt et al. remarked that such a framework showed potential for heterogeneous simulations where objects were simulated at different resolutions and with objects of differing types (i.e. rigid, soft, etc).

Allard et al. [2006] described a software engineering framework for distributed real-time simulation which modularized compute nodes into two categories: animators and interactors. Forces on objects are computed in parallel by integrators from states received from animators. Their modular approach allows for many different simulation techniques to exist in the same environment.

Thomaszewski et al. [2008a] developed a simulation technique for physically based simulation which utilized parallel implicit integration and parallelized collision detection by way of fully dynamic task decomposition. They proposed a task splitting approach which estimates work based on analyses of previous simulation steps. Thomaszewski et al. [2008b] demonstrated a similar technique to cloth simulation using a asynchronous variational integrator which realizes asynchronous simulation of cloth with high frequency details such as wrinkles and folds.

Data-flow analysis has also been applied to improve parallelism. For example, Hermann et al. [2009] used task dependency graphs to extract parallelism information at compile time. Hermann et al. [2010] combined this approach with a two-level scheduler and task-stealing to achieve parallel rigid body simulation in heterogeneous CPU/GPU environments. Furthermore, Pingali et al. [2011] incorporated dependency graphs in the Galois framework to run several parallel algorithms on multiprocessors systems.

Highly parallel continuous collision detection developed by Kim et al. [2009] realizes continuous collision detection in hybrid CPU/GPU environments by using CPUs to cull collisions before performing continuous collision on GPU cores. Similarly, Pabst et al. [2010] utilize spatial reasoning to realize efficient real-time simulation in hybrid CPU/GPU environments by using a highly parallel spatial subdivision algorithm to cull simulated objects followed by a GPU-based narrowphase. Focusing only on GPU collision detection, Tang et al. [2011] present a fast steam based method for collision detection between deformable models.

3 Overview

In this section, we describe our framework at a high level. We first discuss how to simulate rigid bodies using an object-centric formulation, and then highlight our proposed spatiotemporal data structure for obtaining highly scalable rigid body simulations.

Our framework adopts an object-centric simulation model, rather than the traditional frame-centric approach. Each simulated rigid-body (entity) within our framework is modeled as an independent *soft-thread* which communicates with other entities only via message passing. Each entity is self-contained, simulating itself locally and maintaining its own local simulation time, referred to as Local Virtual Time (LVT). LVT values are represented as integers and correspond to the intended render time of the frame being simulated. Our object-centric model permits entities to be simulated at varied time steps depending on their individual state or scene conditions. In this way, objects that are sufficiently isolated in spacetime are able to take larger time steps. This saves computation time for more demanding collisions occurring elsewhere and results in an improved scalability.

Events between rigid-bodies (i.e., collisions) are modeled discretely following the approach of Mirtich [2000]. These events occur at a single moment in simulation time and they are realized as messages exchanged between entities. Due to varying step sizes it becomes possible for an entity to receive an event which occurs in its recent past. In order to handle this case, each entity maintains a buffer of recently computed physical states which enables each entity to *rollback* to a previous point in its simulated time-line. Thus, each entity is capable of successfully resolving events which occur in its recent past. Causality is then maintained by use of *antievents* which allows previously reported events to be undone when an entity rollsback and invalidates the event's corresponding physical state.

Entities must undergo a minimal amount of synchronization in order to collectively maintain a global notion of time, referred to as Global Virtual Time (GVT). The value of GVT is defined as the minimum of all entity LVTs. Once an entity computes a new physical state and, any corresponding events have been acknowledged by other involved entities, each entity records its latest physical state into a write parallel buffer for the given LVT value and increments a shared counter. Conversely, during rollback, entities remove their entries from this shared buffer, decrement the shared counter, send corresponding antievents, and wait for acknowledgment. A frame update is triggered when the counter for a given LVT value is equal to the number of entities in the scene. As the last entity at a given LVT value triggers the update, the value of GVT serves to partition those physical states stored by entities into two classes: states occurring at or before GVT are known and those occurring later are

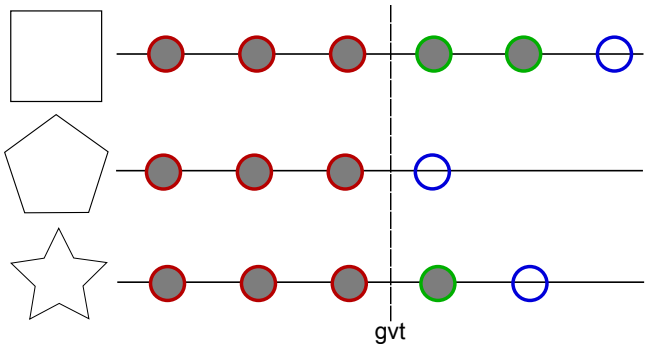


Figure 2: *Simulation of three objects within our framework. Each circle depicts a simulation time step. Filled circles denote completed time steps. GVT is defined as the maximum over all completed time steps. Thus, states prior to GVT (outlined in red) can be safely deleted.*

not. This provides the primary mechanism for garbage collection and deterministic frame generation. Figure 2 illustrates the role of GVT, as well as rollback, within our framework.

To maintain efficient frame aggregation, LVT values are limited to be a fixed step size away from GVT. When an entity integrates forward in time it does so to this maximum upper bound. This larger stride forward is then subdivided into smaller substeps, and the resulting physical states are computed and recorded into a highly-parallel commonly accessible temporal, volumetric hashtable (TVHT). Should writing a physical state for a particular substep result in contacts being reported by the TVHT, integration stops on the given substep in which the contacts occurred and the colliding entity synchronously reports collision events to all entities it made contact with. To achieve a high-degree of scalability, the TVHT does not synchronize entities which are sufficiently separated in space or time (substep). Our framework does not require users to predefine extents of the simulation space, but rather allows entities to exist anywhere within $\mathbb{R}^3$. Details of TVHT are presented in the following section.

4 Parallel TVHT

In the parallel setting, traditional data structures (e.g. kd-trees, hash tables, etc) require a great deal of synchronization to function which ultimately limits the performance gains across many computational cores. Furthermore, a majority of the synchronization in such data structures is unnecessarily pessimistic [Mirtich 2000], as simulation objects sufficiently separated in spacetime are synchronized just as objects that are very close together.

To alleviate these issues, we adopt a spatial reasoning approach inspired by the "zone-agents" of Dequidt et al. [2004]. Our approach combines spatial *buckets* with an efficient temporally-aware volumetric hash table in order to achieve fine-grained synchronization for entities within our scenes. Rather than fixing the size of the simulated world by partitioning it to fixed-size cubes, buckets within our framework moderate many regions of the simulated world.

Similar to entities, buckets are modeled as soft-threads within our framework. A fixed number of buckets are started and enumerated prior to simulation. Buckets support *read*, *write*, and *delete* as atomic operations, as well as mutual exclusion via *lock* and *unlock*. *Read* and *write* operations are defined over the four-tuple $\{P, V, T, B\}$, *delete* is defined over the triple $\{P, T, B\}$, and *lock* and *unlock* are defined over $\{P, B\}$ where P is the process identifier of the entity performing the operation, V is a physical state, $T \in \mathbb{N}$ is an LVT value corresponding to V, and $B \in \mathbb{Z}^3$ is an enumerated value for a particular volume of simulation space.

After each integration step, entities map an Axis-Aligned Bounding Box (AABB), which over-approximates their physical extents, to a set of buckets via volumetric hashing. Each entity then writes its physical state for its current LVT to all selected buckets. In response to a write, each bucket performs narrowphase collision detection against every other physical state recorded within it that happens to exist at the same LVT value and the same region of simulation space. A list of contacts, possibly empty, is generated and returned by the bucket where the authoring entity aggregates contact results from all selected buckets and issues the corresponding collision events. This process is shown in Figure 3.

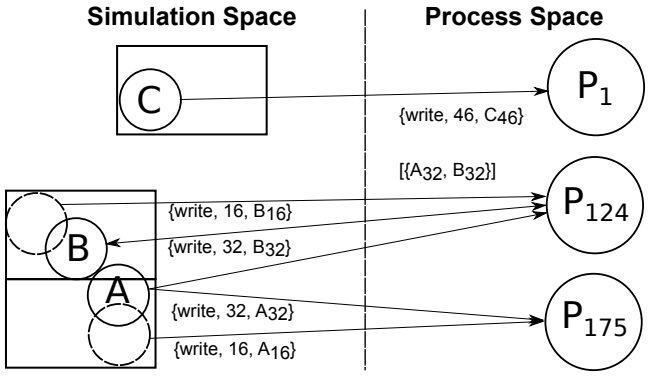


Figure 3: *Three entities: A, B, and C are depicted moving through regions of simulation space at differing values of LVT. C is stationary while A and B approach each other and collide at $LVT = 32$. After each* substep *each entity issues write commands at its current LVT along with its current physical state to processes via volumetric hashing. B writes second to P_{124} which results in the contact $\{A, B\}$ being reported to B.*

Given that it is possible for entities to exist at different LVT values, each bucket maintains a set of previously reported physical contacts. When an entity rolls-back, it deletes previously written physical states for every invalidated LVT value. In response to a delete, buckets return a list of previously reported contacts which have been invalidated as a result of the delete action. Entities aggregate invalidated contacts from each selected bucket, ignoring duplicates, and report antievents corresponding to each such contact to the other affected entities.

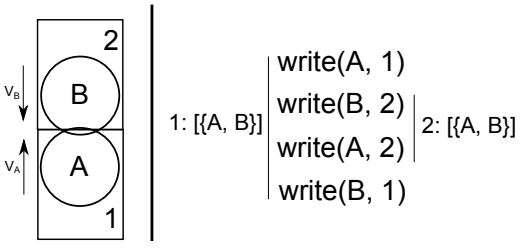


Figure 4: ***Left:*** *Two objects, A and B, collide while each occupying bucket 1 and 2 at a given LVT.* ***Right:*** *Potential ordering of write operations which results in bucket 2 returning $\{A, B\}$ to entity A and bucket 1 also returning $\{A, B\}$ to entity B.*

Maintaining contacts in this fashion allows writes and deletes to occur in any order, which can result in the same contact being detected by separate entities, as shown in Figure 4, as well as stale

contacts reported for physical states which will be soon deleted. While the rollback mechanism is general enough to handle the latter, mutual exclusion must be applied in order to prevent applying forces from multiple contacts more than once. As such, each entity first obtains the lock for every region of simulation space prior to performing write or delete operations during integration or rollback. Algorithm 1 details our approach to locking.

Algorithm 1: Bucket locking procedure.

Input: AABBs, BucketSize
$BucketIds \leftarrow \emptyset$;
for $AABB \in AABBs$ **do**
  $Temp \leftarrow enumerate(AABB, BucketSize)$;
  $BucketIds \leftarrow BucketIds \cup Temp$;
end
$BucketIds \leftarrow sortAscending(BucketIds)$;
for $Id \in BucketIDs$ **do**
  $BucketPID \leftarrow hash(Id)$;
  $lock(BucketPID, Id)$;
end

Obtaining locks in Algorithm 1 is done carefully in order to avoid deadlock. More formally:

Lemma 1. *Algorithm 1 is deadlock-free.*

Proof: A deadlock can occur only if obtaining resources satisfies the circular wait condition [Coffman et al. 1971]. As N^3 is countable, bucket identifiers can be sorted into a lexicographical ordering. By locking in lexicographical order, we are assured that all buckets are locked in an increasing manner. As an entity can hold the lock for a given bucket once, the circular wait condition is eliminated. Thus, the TVHT locking procedure is deadlock-free. ∎

In the following section we present proofs of correctness for our simulation protocol. In doing so, we rely on the following assumptions regarding entity interactions with the TVHT:

1. Contacts are not reported twice to distinct entities. While integrating, entities first obtain locks for all buckets relevant to their motion over the entire *step*.
2. Invalidated contacts are not reported twice to distinct entities. Similar to the previous case. Entities deleting values from the TVHT as when rolling-back first obtain locks for all affected buckets over the entire rollback period.

5 Object-Centric Simulation Protocol

As mentioned in Section 3, entities are represented within our framework as soft-threads. Instead of being scheduled directly by the operating system each entity places tasks on one of several task queues [Mohr et al. 1991], one per core. Each queue has assigned to it a dedicated scheduler, an actual hardware thread, which executes each task in turn starting from the head of their respective queue.

Our object-centric simulation protocol contains three phases: *step*, *rollback*, and *resolve*. As an object integrates through time, it will move through these various phases based on its interactions with other objects. Each of these phases has a different role in maintaining efficient, deadlock-free simulation. In the *step* phase, entities integrate themselves forward in time. If an entity detects a new contact during the *step* phase, it will move to the *resolve* phase where it waits for each contacted entity to acknowledge that they have received and resolved the corresponding collision. At any point in *step* or *resolve* it is possible for an entity to receive an event, or antievent, occurring in its past. When this occurs, the receiving entity abandons its current phase and immediately enters the *rollback* where the entity proceeds to rollback its physical state to the time the event occurred. Every physical state occurring after the event is deleted from the TVHT which may result in synchronization with other entities resulting from antievents. Unlike *step* and *resolve*, *rollback* is un-interruptible. An entity will only leave *rollback* when there are no more pending antievent acknowledgments or events occurring in the past. A finite state machine representing entities within our framework is given in Figure 5.

Notations and Definition: An entity's state is described by the tuple $\{$*LVT, LVT_{ub}, Substep, Phy, Input, Done, Pending, Phys*$\}$, where *LVT* is the entity's current LVT value, LVT_{ub} is the maximum allowed value of LVT given the current value of *GVT*, *Substep* is the rollback granularity in milliseconds, *Phy* is the latest physical state corresponding to *LVT*, *Input* is a queue of pending events in ascending order based on the LVT value when the event originated, $Done$ is a similarly ordered set of events which have already been processed, *Pending* is a set of event identifiers still awaiting acknowledgment, and $Phys$ an ordered key-value store mapping previous LVT values to their corresponding physical states.

Note, we limit our discussion of each phase to only the generation and acknowledgement of events and antievents as these interactions between entities are relevant to correctness of our protocol. Proofs of correctness follow immediately after presentation of the three phases in the following subsections. For brevity, we assume the existence of the following functions in our presentation:

1. *receive* and *receive_tmo*. Both functions receive a message from an entity's message queue with the latter supporting a single argument which allows the user to specify a timeout.
2. *insert*, *delete*, *peek*, *exists*. Common queue operations.
3. *ack*. Sends the acknowledgement for an event or antievent to the other involved entity.

All other functions used in pseudocode are listed in Appendix A.

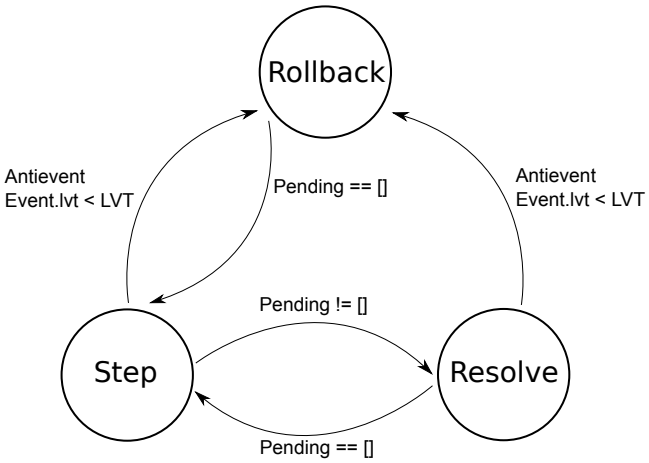


Figure 5: *Formulation of a simulation entity, capable of rollback, as a finite-state-machine.*

5.1 Step

Entities in the $step$ state attempt to integrate themselves forward in time. Events received while in this phase are placed into the sorted *Input* queue. Receiving an antievent immediately interrupts this

process, resulting in the entity entering $rollback$ after removing the corresponding event from *Done*. Once no more messages are able to be received the entity immediately times-out and one of two cases applies, either: the head of the sorted *Input* queue is an event in the entity's recent past (resulting in rollback) or it is not. *Step* is demonstrated in Algorithm 2.

As noted in the previous section, entities which are integrating forward in time must first obtain the locks for all buckets affected by its motion. This is accomplished by first computing all the physical states for each *substep* over the *step* interval. After which, bounds can be computed and used to lock the TVHT prior to the integrating entity writing its physical states for each *substep* into the TVHT. Writing of physical states continues to the earliest *substep* in which contacts are detected, or the last if none are reported.

Algorithm 2: Step phase.

```
Msg ← receive_tmo(0);
switch Msg.type do
    case Event
        Input.insert(Msg);

    case Antievent
        Done.delete(Msg.id);
        rollback(Msg.lvt);
        ack(Msg);

    case NULL
        // Timeout
        Event ← Input.peek();
        if Event.lvt < LVT then
            rollback(Event.lvt);
        else
            stepTowardsLVTub();
            if Pending ≠ ∅ then resolve();
endsw
```

5.2 Resolve

In order to be GVT-invariant, every entity must ensure that each event it generates has been fully applied to the system before proceeding to its next LVT value. To accomplish this, we require every reported event to be acknowledged by the receiving entity before the reporting entity is allowed to resume integration forward in time. Enforcing this policy is the role of the $resolve$ phase. The phase itself operates very simply, blocking-and-waiting for all event acknowledgments corresponding to events stored in the $Pending$ buffer. $Resolve$ is interrupted when either an antievent or event occurring in the past is received. In both cases the receiving entity immediately enters $rollback$. $Resolve$ is described in Algorithm 3.

5.3 Rollback

Similar to events, antievents must also be synchronously acknowledged in order to maintain GVT. The *rollback* phase blocks-and-waits for antievent-acknowledgments for each antievent in its $Pending$ buffer. $rollback$ differs from $step$ and $resolve$ primarily in that: $rollback$ is the only phase which is nondeterministic (i.e. $rollback$ applies itself recursively, adding more antievents to the $Pending$ buffer) and $rollback$ will not change to a different phase before all pending antievents have been acknowledged. $Rollback$ is presented in Algortihm 4.

Algorithm 3: Resolve phase.

```
while Pending ≠ ∅ do
    Msg ← receive();
    switch Msg.type do
        case EventAck
            Pending.delete(Msg.id);

        case Antievent
            if Pending.exists(Msg.id) then
                Pending.delete(Msg.id);
            else
                Pending ← ∅;
                Done.delete(Msg.id);
                rollback(Msg.lvt);
                ack(Msg);

        case Event
            Input.insert(Msg);
            if Msg.lvt < LVT then
                Pending ← ∅;
                rollback(Msg.lvt);
    endsw
end
```

5.4 Protocol Correctness

To demonstrate the correctness of our approach we will show that any two entities in any valid combination of phases are deadlock-free. As two entities can only deadlock when the circular wait condition is met, we restrict our consideration to points of synchronization between entities in all possible cases. We prove our protocol to be deadlock-free in the following combinations of phases: *step-step*, *resolve-resolve*, *resolve-step*, *rollback-step*, *rollback-resolve*, and *rollback-rollback*. While this may not be sufficient to rigorously prove deadlock-freeness for all possible configurations of N entities, in practice our methods scale successfully to include hundreds of objects frequently colliding.

We first present a proof which demonstrates that two entities in *step* will not deadlock, followed by proofs for *step-resolve* and *resolve-resolve*. We conclude this section with a proof demonstrating that entities are deadlock-free when one entity is in *rollback*.

Lemma 2. *Two entities in step are deadlock-free.*

Proof: Only one point of synchronization occurs between two entities in $step$, the locking of TVHT buckets during integration. We have already proven that the TVHT locking procedure is deadlock-free, as shown in lemma 1. Therefore $step$ is also deadlock-free. ■

Lemma 3. *step-resolve and resolve-resolve are deadlock-free.*

Proof: Let A, B be entities s.t. A is in $resolve$ waiting on acknowledgement of an event, E, from B which is in either $step$ or $resolve$. Each unique pairing of these phases are considered below:

1. $resolve$-$step$. B's response to E varies based on the time the event occurred. As we know that $A.LVT = E.LVT$ we consider the following two cases:

   (a) $A.LVT \geq B.LVT$. B places E event onto its $Input$ queue. As $A.LVT \geq B.LVT$ we know that on a fu-

Algorithm 4: Rollback phase.

```
Data: PrevLVT, an LVT value in the past.
Invalid ← [P|{PLVT, P} ∈ Phys, PLVT > PrevLVT];
rollbackState(PrevLVT, Invalid);
while Pending ≠ ∅ do
    Msg ← receive();
    switch Msg.type do
        case AntieventAck
            Pending.delete(Msg.id);

        case EventAck
            // Occurs when resolve enters
            // rollback.  Ack has no effect.

        case Antievent
            if Input.exists(Msg.id) then
                Input.delete(Msg.id);
            else
                Done.delete(Msg.id);
                rollback(Msg.lvt);
                ack(Msg);

        case Event
            Input.insert(Msg);
            if Msg.lvt < LVT then rollback(Msg.lvt);

    endsw
end
```

ture integration step for B that $B.LVT = A.LVT$. At which time, B will process E and send A the acknowledgement.

(b) $A.LVT < B.LVT$. Upon receiving E, B immediately rolls-back to $E.LVT$ and places E onto its input queue. After rollback $A.LVT = B.LVT$ and, as we have seen in the previous case, B will acknowledge E.

2. $resolve$-$resolve$. As B is also in resolve we know that it must be waiting on acknowledgement of an event, E_B, from A. As before, our proof depends on the relation between $A.LVT$ and $B.LVT$:

(a) $A.LVT < B.LVT$ or $B.LVT < A.LVT$. w.l.o.g assume that $A.LVT < B.LVT$. As B generated E_B from a contact reported from the TVHT it must be case that A wrote a physical state to the TVHT corresponding to $B.LVT$. As $A.LVT < B.LVT$ it must be that B received a contact just prior to A undoing said contact as part of $rollback$. However, this cannot be true, as then A would be in $rollback$ awaiting acknowledgement of an antievent for E_B. This is a contradiction as A is in $resolve$. Therefore, our previous assumption is false and this case is impossible.

(b) $A.LVT = B.LVT$. Again, this leads to a contradiction. TVHT enforces the invariant that a collision event between two entities at the same LVT value cannot be detected by two different entities.

■

Lemma 4. *All combinations of two phases where one phase is rollback are deadlock-free.*

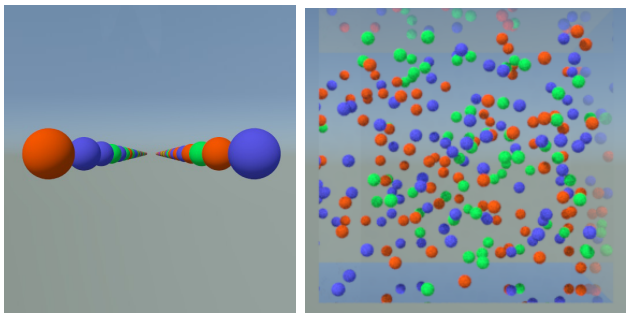
Proof of this lemma is listed in Appendix B. Lemmas 1-4 together demonstrate our protocol is deadlock-free.

6 Results

We implemented our approach using Erlang, a functional programming language with support for concurrent operations [Armstrong et al. 1996]. We tested our framework over 16 cores of an Intel(R) Xeon(R) CPU E5-2670 (Sandy Bridge) operating at 2.60GHz. Tests were done over various scenarios, with different numbers of cores utilized. All times reported are only for simulation, rendering was performed offline. Results were obtained using Erlang's interpreter, having disabled its native code compiler HiPE.

6.1 Scenarios

We analyzed the performance of our framework on the three scenarios described below.



(a) *Parallel Scenario* **(b)** *Bouncing Scenario*

Figure 6: ***Experimental Scenarios*** *(a) Columns of spheres move in parallel motion. (b) Spheres bounce on walls of a translucent cube.*

Parallel: In this scenario, two columns of spheres move past the camera at a constant velocity (Figure 6a). This scenario was tested with both 200 and 500 spheres. Because there are no collisions or rollback, this scene provides a baseline as best-case scenario in terms of scalability.

Cylinders: In this scenario, a stack of spheres falls onto a static array of cylinders (Figure 1). This scenario was tested with both 200 and 500 spheres. It features thousands of collisions, rollbacks and pair-wise interactions.

Bouncing: In this scenario, several balls bounce off a translucent box (and each other) (Figure 6b). This scenario was tested with 200 spheres, randomly initialized in boxes of various sizes (from $100m^3$ to $400m^3$). We also varied the numbers of spheres in a fixed scene size of 250 m^3 to measure performance. The amount of collisions and synchronization increases as the scenario density increases.

6.2 Scalability

We first analyze the scalability of our approach as a function of the number of cores. Figure 7 reports the runtime speedup across 16 cores over all three scenarios for both 200 and 500 objects. All scenarios scale well up to 8 cores, with a speedup of between 5X and 6.5X. On 16 cores, the Cylinders scenario with 200 objects failed to perform significantly faster than with 8 cores. This is likely due, in part, to there not being enough entities in the scene to keep all 16 cores busy. We also note that the Bounce scenario achieves a 7x speedup on 16 cores in both the 200 and 500 object case. The confined state of the spheres in the Bounce scenario limits the number

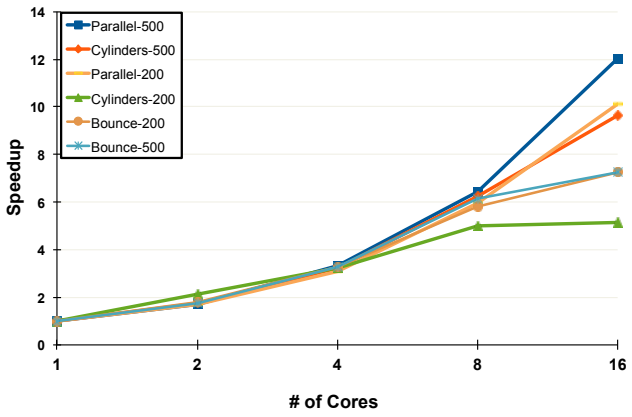


Figure 7: ***Parallel Speedup*** *Scalability of all three scenarios across various number of cores. With only 200 objects, the Cylinders scenario scales well to 8 cores. With larger number of objects both the Cylinders and Parallel scenarios scale well up to 16 cores (9X or more).*

of buckets available which results in a persistently higher degree of synchronization. For the other scenarios, with more objects or less collisions, we see scalability up to 16 cores, with speedups between 9.5X and 11X.

Further insight into scalability can be gained by looking into a breakdown of what percentage of the runtime each activity takes while being run with various numbers of cores. Figure 8 shows the runtime breakdown of various functions for the Cylinders scenario with 500 objects. As it can be seen in this figure, checking for collisions in the TVHT occupies more than half of the runtime, with the rest of the cost primarily being split between responding to a GVT update event and other overhead (e.g., language runtime environment). Neither integration nor processing rollback events contribute significantly to the overall runtime. Figure 8 also demonstrates that collision checking still scales sub-linearly even with our TVHT data structure. This is because two entities that collide must synchronize with each other, reducing the overall parallelism.

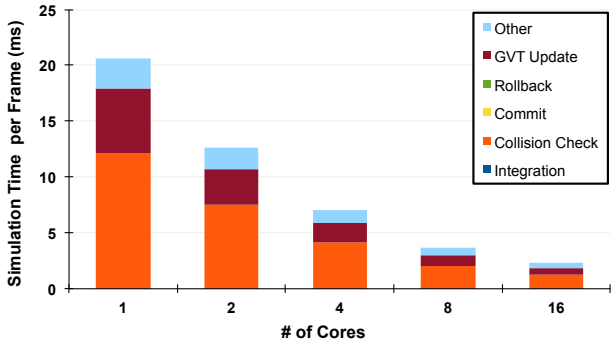


Figure 8: ***Runtime Breakdown*** *Simulation profile of the Cylinders scenario with 500 objects across varying number of cores. Runtime is dominated by collision checking, and to a lesser extent synchronizing GVT between objects, and other runtime costs such as language overhead.*

The density of objects in the scene can have important effects on the scalability of the simulation. As previously discussed, increasing the frequency of interactions tends to reduce the amount of possible parallelism. We explored this effect in the Bouncing scenario, where we varied the volume of the container the balls were bouncing in from $100m^3$ to $400m^3$. The results are reported in Figure 9. As can be seen here, the simulation time decreases (and performance increases) as the simulation density decreases. We note though that in all cases, the performance was realtime and scaled to 9X or larger on 16 cores.

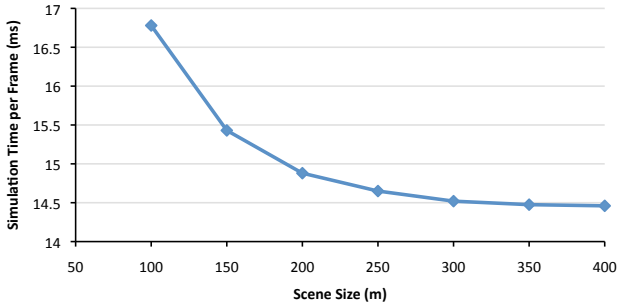


Figure 9: ***Scene Density Analysis*** *The density of the scenario can affect the overall performance. Here the Bouncing scene is simulated with 200 obstacles on 16 cores with a varying size of the container box. As the density decreases the performance increases.*

6.3 Dynamic Step Size

Beyond allowing for good scaling across multiple cores, our proposed object-centric implementation of timewarp allows objects to take large step sizes, refining only as needed when collisions are detected across spacetime in the TVHT. These large timesteps allow us to see large performance gains in scenarios such as Cylinders and Parallel where objects interact infrequently by speculatively simulating into the future and only rolling back when necessary. Figure 10 shows the advantage gained from these large timesteps in terms of runtime performance. As larger values of LVT_{ub} are allowed, faster performance is observed. This increase in performance is related, in part, to a decrease in lock contention resulting from entities taking larger time steps and locking less frequently. In our experiments, the effect of the larger timesteps began to plateau around 256ms, but the overall benefit varies from scene to scene. These gains are diminished for the Bounce scenario where collisions are frequent.

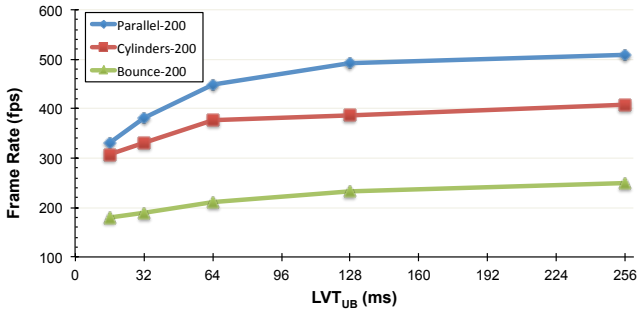


Figure 10: ***Large Timestep Analysis*** *As the per-object timestep size increases, simulations performance also improves. If the timestep is too large for collision-free motion, objects will rollback to the frame on which the collision occurs. These gains are dimished for the Bounce scenario where collisions are frequent. Results collected on 16 cores.*

7 Conclusion and Future Work

We have presented a framework for parallel simulation of rigid bodies across many CPU cores. Our framework combines an object-centric simulation model with a novel highly-parallel, spatiotemporal data structure. We have demonstrated its scalability and runtime performance through various scenarios. In all of our experiments, simulations scale well over many cores. Furthermore, the runtime performance of our simulations improve as objects increase their simulation timestep.

Limitations: The proposed framework has some limitations. As noted previously, the degrees of freedom of each entity plays a large role in our framework's ability to effectively make use of all available computational cores. Scenes where all entities exist in persistent contact with one another will currently not see any substantial runtime speedup as entities in the scene synchronize at every substep with their immediate neighbors. In order to handle these scenarios as effectively as possible our method needs to be extended to explicitly model this form of persistent contact. Adapting the mass splitting approach proposed by Tonge et al. [2012] to our object-centric formulation shows promise.

Future Work: There are many avenues for future research. For one, we would like to investigate how are framework can be extended over many nodes. This is an important step towards implementing a scalable peer-to-peer network that will allow us to efficiently distribute virtual worlds over many machines and spread network and simulation costs across all computers actively participating in the simulation. Connecting entities into Voronoi-based peer-to-peer networks [Hu et al. 2006] provides some interesting ideas in this direction.

Prevalence of GPU-based simulation methods prompts us to also investigate extending our framework to include heterogeneous CPU/GPU environments. Introducing a GPU will create another point of synchronization between entities sharing the GPU. As such, additional techniques for batch scheduling GPU tasks need to be investigated.

We would also like to explore methods to improve the overall runtime performance of our framework. For example, our current implementation uses interpreted Erlang modules, which may slow down the runtime. Exploiting natively compiled modules will likely improve performance at the cost of portability.

Acknowledgements

This work was supported in part by the University of Minnesota Supercomputing Institute.

References

AINSLEY, S., VOUGA, E., GRINSPUN, E., AND TAMSTORF, R. 2012. Speculative parallel asynchronous contact mechanics. *ACM Transactions on Graphics 31*, 6, 151.

ALLARD, J., AND RAFFIN, B. 2006. Distributed physical based simulations for large VR applications. In *Virtual Reality Conference, 2006*, IEEE, 89–96.

ARMSTRONG, J., VIRDING, R., WIKSTR, C., WILLIAMS, M., ET AL. 1996. *Concurrent programming in ERLANG*. Prentice Hall.

COFFMAN, E. G., ELPHICK, M., AND SHOSHANI, A. 1971. System deadlocks. *ACM Computing Surveys 3*, 2, 67–78.

DEQUIDT, J., GRISONI, L., AND CHAILLOU, C. 2004. Asynchronous interactive physical simulation. Rapport de recherche, INRIA.

HARMON, D., VOUGA, E., SMITH, B., TAMSTORF, R., AND GRINSPUN, E. 2009. Asynchronous contact mechanics. *ACM Transactions on Graphics 28*, 3, 87.

HERMANN, E., RAFFIN, B., FAURE, F., ET AL. 2009. Interactive physical simulation on multicore architectures. In *Eurographics Workhop on Parallel Graphics and Visualization*, 1–8.

HERMANN, E., RAFFIN, B., FAURE, F., GAUTIER, T., AND ALLARD, J. 2010. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In *Euro-Par 2010-Parallel Processing*. Springer, 235–246.

HU, S.-Y., CHEN, J.-F., AND CHEN, T.-H. 2006. Von: a scalable peer-to-peer network for virtual environments. *Network, IEEE 20*, 4, 22–31.

JEFFERSON, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3, 404–425.

KIM, D., HEO, J.-P., HUH, J., KIM, J., AND YOON, S.-E. 2009. Hpccd: Hybrid parallel continuous collision detection using cpus and gpus. In *Computer Graphics Forum*, vol. 28, Wiley Online Library, 1791–1800.

MIRTICH, B. 2000. Timewarp rigid body simulation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., SIGGRAPH '00, 193–200.

MOHR, E., KRANZ, D. A., AND HALSTEAD JR, R. H. 1991. Lazy task creation: A technique for increasing the granularity of parallel programs. *Parallel and Distributed Systems, IEEE Transactions on 2*, 3, 264–280.

PABST, S., KOCH, A., AND STRASSER, W. 2010. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In *Computer Graphics Forum*, vol. 29, Wiley Online Library, 1605–1612.

PARALLELA, 2012. http://www.parallella.org/.

PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., ET AL. 2011. The tao of parallelism in algorithms. In *ACM SIGPLAN Notices*, ACM, 12–25.

TANG, M., MANOCHA, D., LIN, J., AND TONG, R. 2011. Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games*, ACM, 63–70.

THOMASZEWSKI, B., PABST, S., AND BLOCHINGER, W. 2008. Parallel techniques for physically based simulation on multi-core processor architectures. *Computers & Graphics 32*, 1, 25–40.

THOMASZEWSKI, B., PABST, S., AND STRASSER, W. 2008. Asynchronous cloth simulation. In *Computer Graphics International*.

TILERA, 2007. http://www.tilera.com/.

TONGE, R., BENEVOLENSKI, F., AND VOROSHILOV, A. 2012. Mass splitting for jitter-free parallel rigid body simulation. *ACM Transactions on Graphics 31*, 4, 105.

ZHENG, C., AND JAMES, D. L. 2011. Toward high-quality modal contact sound. *ACM Transactions on Graphics 30*, 4, 38.

A Supplemental Pseudocode

The following algorithms detail helper functions used in Section 5.

Algorithm 5: stepToLVTub

$Substeps \leftarrow [LVT \ldots LVTUB \text{ by } Substep]$;
$Nphys \leftarrow \emptyset$;
$Nphy \leftarrow Phy$;
foreach $S \in Substeps$ **do**
  $Events \leftarrow [E \in Input | E.lvt = S]$;
  foreach $E \in Events$ **do**
    $Nphy \leftarrow$ applyEvent(E, $Nphy$);
  end
  $Nphy \leftarrow$ integrate($Substep$, $Nphy$);
  $Nphys.insert(Nphy)$;
end

Algorithm 6: stepState

Input: Ordered set of LVT values, $Substeps$, and a parallel set of physical states, $Nphys$.
$Bounds \leftarrow [bound(P) | P \in Nphys]$;
tvht_lock($Bounds$);
foreach $\{S, B, P\} \in zip(Substeps, Bounds, Nphys)$ **do**
  $Events \leftarrow [E \in Input | E.lvt = S]$;
  foreach $E \in Events$ **do**
    ack(E);
  end
  $LVT \leftarrow S$;
  $Phys$.insert({S, P});
  $Input \leftarrow Input \setminus Events$;
  $Done \leftarrow Done \cup Events$;
  $Contacts \leftarrow tvht\_write(Bound, P)$;
  if $Contacts \neq \emptyset$ **then**
    $Pending \leftarrow$ generateCollisionEvents($Contacts$);
    sendEvents($Pending$);
    break;
end
tvht_unlock($Bounds$);

Algorithm 7: rollbackState

Input: A previous LVT value, $PrevLVT$, and $Invalid$, a set of invalid physical states.
$Bounds \leftarrow [bound(Phy) | Phy \in Invalid]$;
tvht_lock($Bounds$);
foreach $I \in Invalid$ **do**
  $BadContacts \leftarrow$ tvht_delete(I);
  $Antievents \leftarrow$ generateAntievents($BadContacts$);
  $Pending \leftarrow Pending \cup Antievents$;
end
tvht_unlock($Bounds$);
$LVT \leftarrow PrevLVT$;
$Phys \leftarrow [\{T, P\} | \{T, P\} \in Phys, T < PrevLVT]$;
$Replay \leftarrow [E \in Done | E.lvt > PrevLVT]$;
$Replay \leftarrow [E \in Replay | \forall A \in Antievents, A.id \neq E.id]$;
$Input \leftarrow [E \in Input | \forall A \in Antievents, A.id \neq E.id]$;
$Input \leftarrow Input \cup Replay$;
$Done \leftarrow [E \in Done | E.lvt \leq PrevLVT]$;

B Additional Proofs

Lemma 4. *All combinations of two phases where one phase is $rollback$ are deadlock-free.*

Proof: Let A, B be entities such that A is in $rollback$ waiting on an antievent, $Antievent$, acknowledgement from B. As $rollback$ is recursive, we first demonstrate that this process will terminate. This is trivial, however, given that $rollback$ always results in an entity existing at an LVT value less than its LVT value prior to $rollback$ and that the value of LVT is bounded below by GVT. Therefore, we are assured that A, B will only $rollback$ finitely many times.

We continue our proof by showing all points of synchronization for all possible phases of B are deadlock-free.

1. B is in $step$.

   In this case, the only point of synchronization between A, B is the locking of $buckets$ during $stepState$ and $rollbackState$. We have already demonstrated the locking of $buckets$ to be deadlock-free in lemma 1. Thus, B is able to process $Antievent$ and send an acknowledgement to A.

2. B is in $rollback$. As B is in $rollback$ and synchronized with A it must be awaiting acknowledgement of an antievent, $Antievent_B$ from A. As locking over the TVHT is atomic for the entire rollback period we can be assured that the antievent of A does not correspond to the same event as B's. Furthermore, as events are defined as occurring at a given time between two entities, therefore $Antievent.lvt \neq Antievent_B.lvt$. w.l.o.g. we examine two cases from the perspective of B:

   (a) $Antievent.LVT < B.LVT$. B then rolls-back as previously described and breaks the deadlock. As A will await acknowledgment from B, we know that B will not run into conflict attempting to process multiple antievents from A.

   (b) $Antievent.LVT > B.LVT$. We know, then, that A and B have rolled back over different, but overlapping regions of spacetime. Also, it must have been the case that A obtained all of its locks for the TVHT first, as B was not the entity which received the invalidated contact between itself and A. As B has already rolled back to a previous point in time, it is safe for B to simply acknowledge $Antievent$ as it has, or will, synchronize with all other entities affected by this later rollback.

3. B is in $resolve$. Thus, B must be awaiting acknowledgement of an event, E from A. Again, we consider the relation between $B.LVT$ and $Antievent.LVT$:

   (a) $B.LVT \leq Antievent.LVT$. A has undone a collision in B's past. B is preempted and enters $rollback$ acknowledging $Antievent$ after successfully rolling-back to, or before, $Antievent.LVT$.

   (b) $B.LVT > Antievent.LVT$. This case emerges when B rolls-back after A but to an earlier value of LVT. As B would have undone the same contact as part of its $rollback$, B simply acknowledges $Antievent$.

■