Speculative Parallel Asynchronous Contact Mechanics

Samantha Ainsley
Columbia University

Etienne Vouga
Columbia University

Eitan Grinspun
Columbia University

Rasmus Tamstorf
Walt Disney Animation Studios

Abstract

We extend the Asynchronous Contact Mechanics algorithm [Harmon et al. 2009] and improve its performance by two orders of magnitude, using only optimizations that do not compromise ACM's three guarantees of safety, progress, and correctness. The key to this speedup is replacing ACM's timid, forward-looking mechanism for detecting collisions—locating and rescheduling separating plane kinetic data structures—with an optimistic speculative method inspired by Mirtich's rigid body Time Warp algorithm [2000]. Time warp allows us to perform collision detection over a window of time containing many of ACM's asynchronous trajectory changes; in this way we cull away large intervals as being collision free. Moreover, by replacing force processing intermingled with KDS rescheduling by windows of pure processing followed by collision detection, we transform an algorithm that is very difficult to parallelize into one that is embarrassingly parallel.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: contact, collision, simulation, parallelization

Links: DL PDF

1 Introduction

The design of physical simulation algorithms can require difficult decisions weighing slower, more principled approaches against faster shortcuts. In the short term, it can be tempting or necessary to prioritize speed. However, in the words of Sutter and Alexandrescu [2004], "it is far, far easier to make a correct program fast than it is to make a fast program correct." In this work we adopt the longer term perspective that as computers become ever faster, methods designed from the ground up to guarantee correctness will prove to be the longest-lasting.

Moreover, while available processing power continues to increase exponentially, this increase is no longer in the raw speed of CPU cores, but in the number of cores available per die [Borkar and Chien 2011]. Algorithms that cannot be parallelized are less likely to survive in the long term.

We seek algorithms to simulate thin flexible materials subject to complex collisions and contact geometries. *Asynchronous contact mechanics* (ACM) [Harmon et al. 2009] addresses this goal by focusing on three built-in guarantees: 1) the simulation is *safe* and provably stops all interpenetrations; 2) it conserves momentum and energy, *physical invariants*, for physical systems with the appropriate symmetries; and 3) for well-posed problems, ACM is guaranteed to make *progress*, in the sense of terminating in finite time. There is, however, a wide gulf between "finite time" and "fast," and the earlier paper was presented as a foundation for systems-style research into correct and *fast* simulations.

We describe a new system for ACM that dramatically decreases the amount of time spent on bookkeeping and collision detection, by an order of magnitude. While the original ACM was difficult to parallelize, we employ *speculation* to expose easy parallelization, leading to another order of magnitude speedup.

Our implementation yields speedups of more than two orders of magnitudes on a 12-core work station, enabling practical computational cost for simulations with complex contact geometries. The algorithm retains ACM's three aforementioned guarantees, and therefore serves as one step toward realizing Sutter and Alexandrescu's statement in the context of physical simulation.

1.1 Overview

Summary of ACM The ACM algorithm upon which our method is based is described in detail by Harmon et al. [2009] and in follow-up work [Harmon 2010; Vouga et al. 2011; Harmon et al. 2011a]. We only briefly summarize the salient features of the method here.

ACM guarantees safety and correctness of the simulation by activating nested *penalty layers* of decreasing thickness $\eta_1 > \eta_2 > \dots$ and increasing stiffness in anticipation of collisions. Whenever two objects approach each other with distance less than η_1, ACM adds a penalty force to the simulation that opposes the collision. If the objects continue approaching and reach a distance η_2 from each other, a second penalty force is added that's stiffer than the first, and so on; the total force applied to the objects grows unbounded as the distance between them decreases, so that it is guaranteed to stop the collision no matter how hard the impact. Each penalty force is stepped *asynchronously* [Lew et al. 2003] instead of in lockstep, allowing each force to be integrated at its own stable time step. An *event priority queue* keyed by time maintains which force is to be processed next.

To detect when to activate a new penalty force, ACM uses *kinetic data structures* (KDSs) [Guibas 1998]. For each pair of primitives in the simulation, a separating slab of thickness η_1 is found that separates the primitives, guaranteeing that they are farther than η_1 apart. This slab certifies that no collisions can occur between them for some time interval into the future. A *KDS event* is placed on the event queue at this expiration time. Whenever the velocity of either primitive changes, this time must be recomputed, or *rescheduled.* When the KDS event is popped from the queue, either a new separating slab is found and its event pushed onto the queue, or the first penalty layer is activated for the two primitives under the conservative assumption that a collision is imminent. The above process is then repeated to detect when to activate the next deepest penalty layer at distance η_2.

An alternative to KDSs The above approach to activating penalty layers guarantees that collisions cannot be missed. Unfortunately, this guarantee carries a heavy cost: in typical simulations

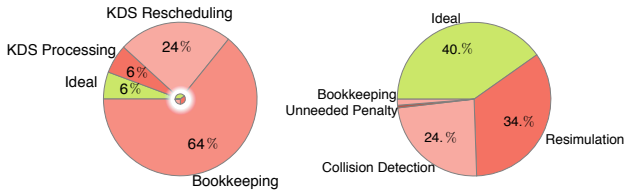

Figure 1: *A breakdown of time spent on useful work* (green)*, and overhead* (orange)*, for the original ACM implementation* (left) *and our approach* (right) *when simulating Harmon et al [2009]'s reef knot example for three simulation seconds. The relative total compute time is illustrated by the total area of the pie charts when the right pie chart is scaled and superimposed on the left. The left plot was generated from data reported by Harmon et al.*

over 90% of the execution time is spent on rescheduling or processing KDS events [Harmon et al. 2009]. Moreover, every time an event is processed, the event queue changes in a way that is unpredictable a priori: a KDS event will either reschedule itself for an unknown time in the future, or will insert a new penalty layer whose first tick could occur an arbitrarily small amount of time later and will itself cause rescheduling of other KDS events. Because of these unpredictable causal dependencies between events, processing and rescheduling of KDS events cannot be easily parallelized.

Instead of using KDSs to guarantee that the simulation never enters an interpenetrating state, we propose taking advantage of the *time warp* paradigm [Jefferson 1985]: we simulate a window of time without attempting to find or resolve any new collisions. At the end of the window we perform retrospective collision detection, and if any collisions were missed, we *roll back* to the beginning of the window, add new penalty layers, and repeat this process until no collisions are detected. We describe this algorithm in detail in §2.

Speculative simulation and rollback is not at first glance an obviously fruitful model for collision detection and response. Consider, as a point of comparison, a hypothetical *ideal* implementation of ACM that wastes no work on collision detection: an omniscient oracle informs this ideal implementation exactly when to activate all penalty layers needed to resolve imminent collisions, and when to deactivate the penalty layer because it will exert zero force the next time it is processed. This ideal implementation gives a lower bound on the amount of computation *any* functionally identical optimization of ACM must perform.

It is useful to compare this ideal implementation to both the original ACM implementation and the proposed rollback scheme: any work done by either method beyond that of the ideal implementation is termed *wasted* work. A rollback scheme performs several types of wasted work:

- **Resimulation**: Every time an event is processed and then rolled back, the time spent processing that event was wasted.
- **Collision detection**: Time spent on collision detection takes away time that could have been spent on integrating forces and advancing the simulation. In the ideal implementation, collision detection is instantaneous.
- **Unneeded penalty forces**: Penalty events that exert zero force (as a result of being added to the event queue by overly-conservative collision detection) do not need to be processed.
- **Bookkeeping**: Rollback has significant miscellaneous overhead: saving and restoring the (large amount of) simulation state after every rollback window, gathering the trajectory data during each window needed by collision detection, and so on.

In the original implementation, processing and rescheduling of KDS events are the main sources wasted work, in addition to event queue maintenance. (Since the event queue contains many KDS events in addition to force events, pushing, popping, and rescheduling events on the queue is more costly.) For the reef knot example by Harmon et al [2009], Figure 1 shows a breakdown of how much wasted work, and of what type, each method exhibits.

Despite the potential overhead of speculative simulation with rollback, this approach has several advantages over original ACM: the collision detection at the end of each simulation window needs only to determine *whether or not* a collision has occurred (the time of impact is unimportant), and can take advantage of four instead of three dimensions of information (see §3). As a result, collision detection and resimulation is substantially cheaper than rescheduling and processing KDS events "along the way." Together these factors significantly reduce the amount of wasted work incurred during a typical simulation. Moreover, both the collision detection and the processing of material/penalty forces within each window can be easily parallelized (§4), unlike rescheduling of KDS events.

Related work The foundation of ACM's time stepping algorithm is the work by Lew et al on asynchronous variational integrators (AVIs) [2003]. Parallel extensions of AVIs have been studied for use in finite element simulations of elastica without contact, using domain decomposition and message passing [Kale and Lew 2007] as well as dependency graphs [Huang et al. 2007]. An implementation of the latter has been incorporated into the Galois framework for running parallel algorithms on multiprocessors [Pingali et al. 2011]; we discuss in Section 5.2 the possibility of using Galois to parallelize our force integration. In graphics, AVIs have been applied to asynchronous integration of cloth internal forces [Thomaszewski et al. 2008b]. Debunne et al [2001] proposed the similar idea of adaptively switching between different levels of resolution and time step when simulating visco-elastic bodies. ACM has recently been extended to handle implicit forces using a ghost mesh [Harmon et al. 2011b].

Mirtich [2000] applied Time Warp to simulations of large numbers of interacting rigid bodies, and our method is largely inspired by this work. Zheng and James [2011] recently used Time Warp at the body level to asynchronously simulate vibrating objects for sound simulation. These methods leverage Time Warp to roll back only those contact groups involved in a detected collision; for simulations with few, large, deformable bodies this approach is less profitable since stiff elastic forces rapidly propagate information away from points of contact. Rather, we use Time Warp primarily because of the substantial savings Retroactive Detection offers asynchronous integration.

Parallel simulation of cloth and thin shells with collisions is a well-studied problem. For instance, Thomaszewski [2006; 2008a] solves for implicit material forces using a data-parallel conjugate gradient algorithm, and uses data from past frames to estimate a good splitting of the collision detection task. Bender and Bayer [2008] simulate inextensible cloth by decomposing it into strips of constraints that can be processed in parallel. Selle et al [2009] efficiently handle very high resolution cloth by parallelizing and extending Bridson's method [2002] to reduce the number of geometric tests needed during collision detection. Our method stands apart in offering ACM's three guarantees of safety, progress, and correctness.

2 Speculative simulation with rollback

In place of forward-looking kinetic data structures, we propose detecting collisions in hindsight and resolving them with a speculative

model in the spirit of Jefferson's *time warp* algorithm [1985]. We tile time into consecutive fixed-sized *rollback windows* of duration R. (We discuss the choice of the parameter R in §5.1.) Within each window, we advance the simulation by processing events as described in the original ACM algorithm, except that we do not add, process, or reschedule any KDS events. That is, we process internal force events, and penalty events for any penalty layers that were active at the start of the window. More collisions may occur during the rollback window but we make no attempt at detecting or responding to them before the end of the rollback window.

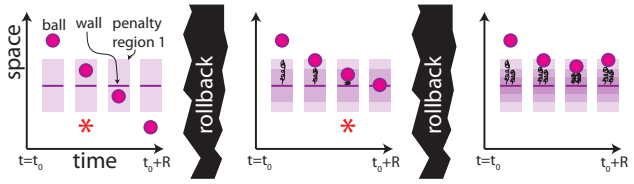

Figure 2: *A cartoon illustrating the rollback process: the simulation steps a ball forward in time, heedless of collisions* (left)*. Interference detection at the end of the rollback window notices that the ball enters the first penalty layer, so the simulation rolls back, activates the first penalty layer, and resimulates the window* (middle)*. During resimulation, the ball enters the second penalty layer; rolling back, activating the second penalty layer, and resimulating, no collisions (i.e., entries into third penalty layer) are detected* (right)*, and the result is finally accepted.*

Algorithm 1 Algorithm to integrate one window

procedure INTEGRATEWINDOW
  $\mathrm{X}_0 \leftarrow \{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots\}$ *// save start-of-window positions*
  $\dot{\mathrm{X}}_0 \leftarrow \{\dot{\mathbf{x}}_0, \dot{\mathbf{x}}_1, \dot{\mathbf{x}}_2, \ldots\}$ *// save start-of-window velocities*
  $Q_0 \leftarrow Q$ *// save start-of-window queue state*
  repeat
    $\mathrm{H} \leftarrow \mathrm{X}_0$ *// reset histories*
    while Q.**head**.$t < t_0 + R$ **do**
      $(E, h, t) \leftarrow Q$.**pop** *// Pop event E with time step h at time t*
      ProcessEvent(E,h,t)
    end while
    $\mathcal{C} \leftarrow$ BroadPhaseCollisionDet(H) *// get missed collisions*
    if $\mathcal{C} \neq \emptyset$ **then**
      $\{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots\} \leftarrow \mathrm{X}_0$ *// restore positions*
      $\{\dot{\mathbf{x}}_0, \dot{\mathbf{x}}_1, \dot{\mathbf{x}}_2, \ldots\} \leftarrow \dot{\mathrm{X}}_0$ *// restore velocities*
      $Q \leftarrow Q_0$ *// restore start-of-window queue state*
      Q.**push**(new penalty events constructed from C)
    end if
  until $\mathcal{C} = \emptyset$
end procedure

We stop processing events when we would process the first event whose time exceeds the end of the rollback window. We then perform interference detection to determine whether any collisions were missed during the window: if so, we restore the entire simulation to its state at the beginning of the rollback window, and activate a penalty layer for each missed collision. We then resimulate the rollback window. The forces from the new penalty layers may have induced additional collisions, or may not have been sufficient to stop the detected collisions, so we repeat this process until collision detection reports no missed collisions during the rollback window. See Figure 2 for an illustration of this process.

Implementation At the start of every rollback window we take a snapshot of the entire simulation state, which we restore in the event of a rollback. This includes positions, velocities, events on the queue and their times, positions of the material reference configuration, etc. We also maintain a *history* of vertex positions: for each vertex, we track its position at the start of the window, the end of the window, and its position and time whenever it changes velocity due to force processing. Since vertices move along piecewise linear trajectories, this history minimally encodes the entire trajectory of the simulation over the rollback window, and is passed to the collision detection algorithm at the end of the window. Note that due to ACM's asynchrony, different vertices have different numbers of history entries and entries do not necessarily align in time.

At the end of every rollback window, if a penalty layer is exerting zero force, we remove it from the list of active penalty layers.

Algorithm 2 Algorithm to process one event

Process an event E with time step h, scheduled time t
procedure PROCESSEVENT(E,h,t)
  $\xi := \mathbf{stencil}(E)$ *// global indices of the local stencil*
  for $i \in \xi$ **do** *// update positions and clocks*
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + (t - t_i)\dot{\mathbf{x}}_i$
    $t_i \leftarrow t$
  end for
  compute $\mathbf{F}_\xi$ *// local impulses $\mathbf{F}_i$ for $i \in \xi$ (embarrassingly parallel)*
  $\dot{\mathbf{x}}_\xi \leftarrow \dot{\mathbf{x}}_\xi - h\mathrm{M}_\xi^{-1}\mathbf{F}_\xi$ *// update velocities (embarrassingly parallel)*
  Q.**push**$(E, h, t + h)$ *// Schedule recurring event*
  for $i \in \xi$ **do** *// save history*
    H_i.**append**$(\mathbf{x}_i, t)$ *// H_i records trajectory of $\mathbf{x}_i$*
  end for
end procedure

3 Collision Detection

At the end of every rollback window, we must examine the trajectory of the system over the course of the window and determine if any collisions occurred. In particular, each pair of material primitives (edge-edge or vertex-face) either has no active penalty layers—in which case we must detect if we need to activate the first, outermost penalty layer for that pair—or they already have some penalty layers on the queue, in which case we must check if the next-deeper layer is needed. To preserve the *safety* guarantee we must perform continuous-time proximity detection on the full trajectory: it is not enough to merely check positions at the end of the window as objects may have tunneled through each other. This proximity detection is the same as collision detection with an offset surface corresponding to the thickness of the penalty layer.

Our particular problem domain has several distinguishing features:

- Each vertex in the simulation moves in a piecewise linear trajectory, and vertices do not change trajectory in lockstep.
- Although the coarse motion of a vertex over the rollback window might be simple, for simulations involving cloth, thin shells, or other objects with stiff internal forces, the fine trajectory is composed of very many high-frequency, low-amplitude oscillations.
- The interference distance we need to detect against differs for each pair of primitives in the simulation, since we always check for that pair entering its next deeper penalty layer, and different primitive pairs in the simulation have different numbers of penalty layers already active.
- We only need to know whether or not a collision occurred at some point during the rollback window – the precise time of impact is unimportant.

We propose a three-phase detection algorithm with these features in mind.

Broad phase: swept-volume k-DOPs We cull collisions between primitives that remain spatially distant for the entire rollback window by fitting a k-DOP hierarchy [Konečný and Zikan 1997] to the swept volumes of the triangles in the simulation. We have observed that (unoriented) 26-DOPs work well in practice. We place bounding volumes around swept volumes rather than around each triangle at each point in time to avoid a complete rebuild of the hierarchy at the end of each rollback window. We also avoid scenarios in which increasing history granularity leads to large hierarchies and therein costly traversal.

Since we need to detect proximity between pairs of primitives, and since that proximity varies depending on how many penalty layers are already active for that pair, during fitting we inflate the k-DOPs by the conservative, largest penalty layer thickness η_1 (the thickness of the first penalty layer). If two leaf nodes (triangles) overlap, we look at all possible pairs of primitives taken from the two triangles, look up how many penalty layers k (if any) are already active for that pair, and check if their swept volume k-DOPs, inflated by η_{k+1}, overlap. If so, we proceed to the narrow phase.

Narrow phase: Space-time separating planes Given an edge-edge or vertex-face pair that could not be culled by the broad phase, the narrow phase must determine if the primitives come within some proximity η_{k+1}. Each of the vertices that compose the pair move in a piecewise linear trajectory, so the rollback window can be subdivided into time intervals during which all vertices have constant velocity; in this setting, proximity detection amounts to finding the roots of well-known continuous collision detection polynomials (with thickness) of degree at most six [Stam 2009; Brochu et al. 2012]. Such root solves are very expensive, so we propose a second phase of culling based on separating slabs that takes advantage of the fact that we do not need the time of impact, and that over a small window of time most forces only cause small perturbations to positions.

Algorithm 3 Algorithm for culling narrow-phase collisions on the interval $[t_0, t_f]$

```
1:  procedure NarrowPhase(t_0, t_f)
2:      if (t_f ≤ t_0) then
3:          return false
4:      else if (t_f − t_0 < tol) ∧ ConstantVelocity(t_0, t_f) then
5:          return ContinuousCollisionDetection(t_0, t_f)   // Fail-safe
6:      end if
7:      t_mid = (t_f + t_0)/2
8:      if PrimitivesProximate( t_mid ) then
9:          return true
10:     end if
11:     (t_l, t_u) = NoCollisionInterval( t_mid )
12:     return NarrowPhase(t_0, t_l) ∨ NarrowPhase(t_u, t_f)
13: end procedure
```

Algorithm 3 outlines our approach. For a rollback window beginning at time t_0 and ending at $t_f = t_0 + R$, we first calculate the distance between the primitive pair at the midpoint $t_{\text{mid}} = \frac{t_0+t_f}{2}$ (line 8). If the pair is within proximity at this time, we know a collision must occur during the rollback window. Otherwise, a separating slab of thickness η_{k+1} must exist and certify the lack of collisions on some interval (t_l, t_u) around t_{mid} (line 11). If $t_l < t_0$ and $t_u > t_f$, we are guaranteed that the pair does not collide for the entire rollback window. If $t_0 < t_l$ we recursively apply this algorithm to the time window $[t_0, t_l]$, and similarly for $[t_u, t_0 + R]$. Figure 3 illustrates this algorithm.

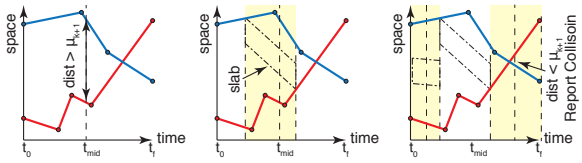

Figure 3: *Narrow phase collision detection using separating slabs. The trajectories of two vertices in 1D are shown. At t_{mid} we detect that the two vertices are sufficiently far apart, so cull an interval of time based on how long a separating slab can be shown to certify that there are no collisions. We then recurse until we have found a collision or proven that no collisions occur at any time in the rollback window.*

Failsafe: Continuous collision detection If we would recurse on an interval that is too small (we use 10^{-10} seconds for this tolerance) and the primitive pair has constant velocity within this interval, we invoke the third phase of collision detection, CTCD using root solving (culling some polynomials when we can prove using interval arithmetic that they have no roots during the rollback window), as a last resort (line 5). Invoking the failsafe is rare: in our benchmark examples, only about 0.2% of all narrow phase calls require the failsafe.

4 Parallelization

In the original ACM code, after every event is processed, KDS events associated to the vertices in the event's stencil must be rescheduled. This rescheduling alters the event queue in unpredictable ways: the KDS event might reschedule itself for an arbitrary later time, or it might remove itself from the queue and activate a penalty layer; it is therefore unclear how this rescheduling could be effectively parallelized. On the other hand, removing KDS events as the mechanism for guaranteeing collision-free simulations, and replacing them with speculative simulation followed by collision detection, opens the door to straightforward parallelization of the entire algorithm.

4.1 Parallelizing force processing

As in the ACM paper, we are concerned primarily with simulating meshes of approximately uniform resolution; for such meshes, the maximum stable time step for the internal forces does not vary much, and so as in the original ACM implementation we *conservatively bucket* material forces into super-elements [Huang et al. 2007; Harmon 2010] by setting all of their time steps to that of the stiffest element. For example, gravity forces, internal forces, and the different layers of penalty forces are each grouped into their own bucket. We then represent the group as a single item on the event queue. Once bucketed, internal forces can be parallelized very simply without synchronization: to process the grouped forces we integrate positions to the current time (see Alg. 2 line 4), compute and store for each force in the group the impulse applied by that force (see Alg. 2 line 8), and iterate over the vertices of the simulation, applying to each velocity the sum of the impulses computed in the previous step (see Alg. 2 line 9).

Each of these steps is done in parallel. Penalty layers of the same depth can be naturally grouped since they have identical stable timesteps; we process them in parallel in the same way.

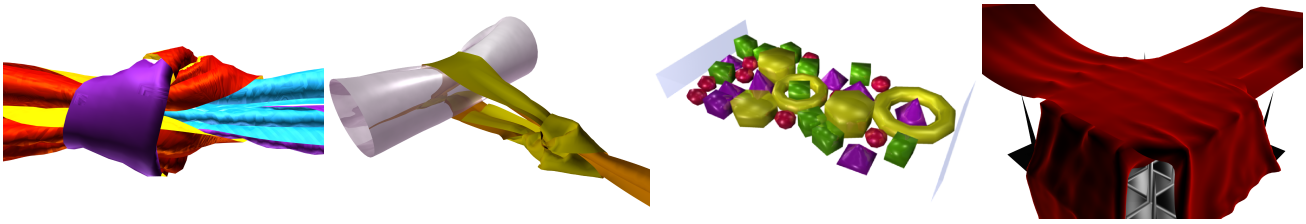

Figure 4: *The benchmarks from Harmon et al [2009]. From left to right: reef knot, bowline knot, trash compactor, and two cloths draped. Timing comparisons for these benchmarks are listed in Table 1.*

For simulations involving graduated meshes of widely varying triangle size, bucketing coarse element at the fine elements's time step is inefficient. An interesting direction for future work would be to explore using several buckets at different orders of time step magnitude to better handle such a distribution of internal force stiffnesses.

4.2 Parallelizing collision detection

To get reasonable scaling behavior, collision detection must also be parallelized. It is trivial to run the narrow phase (both the spacetime separating slabs and the root solve) in parallel. Parallelizing the broad phase effectively is more complicated, so at present we opt for a simple staggering scheme that allows us to run a sequential broad phase while still making use of all available cores by allowing the simulation to proceed to the next rollback window in parallel. A number of better methods have been proposed for efficient parallel collision detection [Kim et al. 2009; Pabst et al. 2010; Tang et al. 2010; Tang et al. 2011], and we hope to incorporate this into our framework in the future.

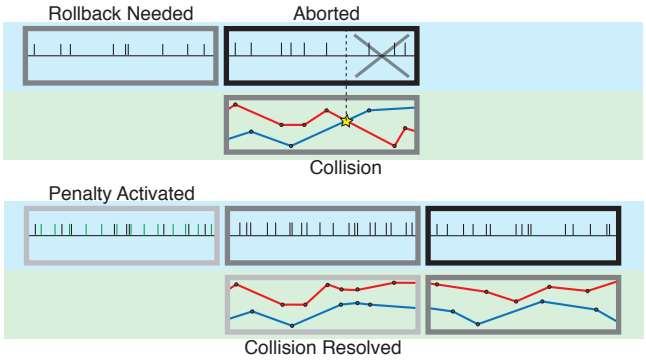

Figure 5: *After simulating a rollback window (*top-left*), we optimistically continue simulating the next window concurrently while performing collision detection (*top-middle*). If a collision is detected, we interrupt the simulation and roll back (*bottom-left*). If collision detection confirms there were no collisions in the last window, we continue simulating (*bottom-right*).*

In our current implementation, whenever we need to perform collision detection, we run it in parallel with optimistic simulation of the next rollback window. In other words, after simulating rollback window i, we perform collision detection on window i, and with any remaining cores begin simulation of window $i+1$ under the assumption that no collisions will be found. If collision detection does return a collision, we immediately stop simulating frame $i+1$ and roll back to the beginning of frame i. Figure 5 illustrates this timeline. Speculatively starting integration of the next frame is advantageous whenever collision detection ultimately finds no collisions during the previous window – in our benchmarks, this occurs 60–70% of the time.

4.3 Miscellaneous Optimizations

Several other optimizations we attempted further improved the performance of our framework.

- **Kernel fusion of the bending and stretching forces**: for simulations involving both bucketed bending and stretching membrane forces, we compute the force contribution of the stretching force at the same time as that of the bending force. Doing so halves the number of times mesh position information must be fetched and improves cache performance.
- **Bandwidth reduction**: Using reverse Cuthill-McKee reordering [Cuthill and McKee 1969], we reduce the number of cache misses incurred while integrating internal forces by reordering the mesh vertices at the start of the simulation. Other methods for cache aware or cache oblivious layouts may be even more effective [Yoon et al. 2005].

In addition to the changes listed above, we also refactored and micro-optimized the published ACM source code in several places (for instance, removing unnecessary trigonometric function calls in the bending force computation, unrolling tight inner loops, and rearranging the layout of data structures in memory to maximize cache efficiency). These changes already improved the performance of our code when run with a single thread. However, more importantly, we found that such low-level optimizations aimed at improving cache performance were essential to achieving reasonable scaling behavior with increasing number of cores.

5 Analysis and Results

Existing benchmarks We ran our method on four of five examples timed by Harmon et al [2009]; see Figure 4. We used an Intel 12-core Westmere-EP workstation (X5690 @ 3.47GHz). To ensure a fair comparison we also re-timed the publicly released ACM code on identical hardware. Table 1 shows the timing comparison; our method is more than two orders of magnitude faster than the original ACM code for all examples. Due to needing fewer events on the queue, our implementation is also more memory efficient – for the Reef Knot, our implementation uses 229 MB of RAM, compared to Harmon et al's 424 MB.

We stress that our method is a conservative optimization of Harmon et al's asynchronous algorithm: we propose changing *how* collision detection is performed (an implementation detail), but not *which* collisions are detected or how these collisions are resolved. In particular, our implementation preserves the three Harmon et al guarantees of safety, correctness, and progress.

The twister Inspired by the video accompanying Bridson's thesis [2003], we simulated a cylindrical rod suspended within a cloth cradle, Figure 6. Constant external torque applied to the rod wrings out the cloth. Since our method preserves all of ACM's guarantees,

Example	Vertices	Original ACM	One thread, no slabs	One thread	Twelve threads	**Total speedup**
Reef Knot	10642	23.6 hrs	74 mins	50.5 mins	5.8 mins	**244x**
Bowline Knot	3995	7.0 days	4.9 hrs	121 mins	15.2 mins	**663x**
Trash Compactor	714	13.8 hrs	53.5 mins	13.3 mins	2.4 mins	**345x**
Two Cloths Draped	15982	11.6 days	6.7 hrs	4.7 hrs	40.1 mins	**416x**

Table 1: *Wall clock time for each of the examples benchmarked by Harmon et al [2009]. We ran each examples using the publicly-available ACM implementation, our code with only a single thread and CTCD only (instead of spacetime separating slabs) for the collision detection narrow phase, our complete code with only a single thread, and our complete code with 12 threads. The simulation parameters were identical to those selected by Harmon et al. For the rollback window size R we used $1/300$ for all examples.*

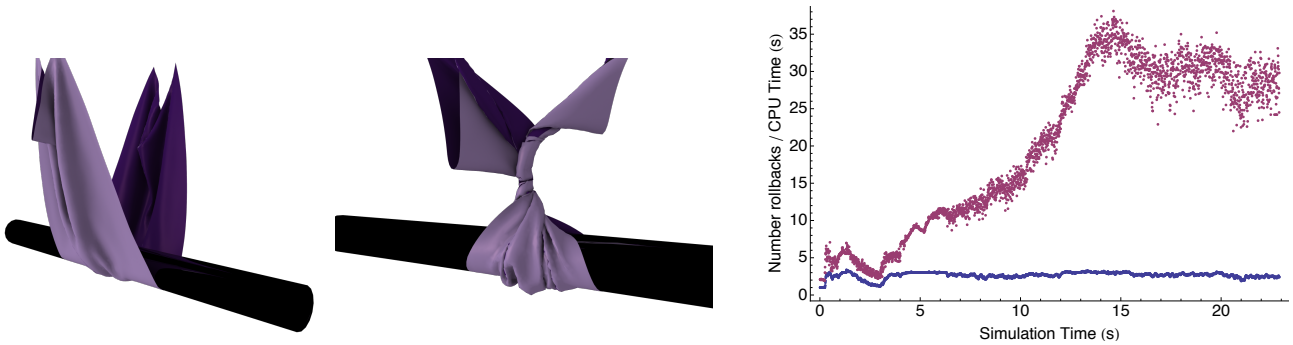

Figure 6: *A spinning rod wrings out a sheet of cloth. Our method's safety guarantee, which we inherit from ACM, ensures that no interpenetrations occur even as the cloth twists tightly around itself multiple times. We plot the number of times each frame rolls back (*blue*) and the total clock wall time spent computing each frame (*maroon*). As a point of context, the rod starts spinning at time* 3.0 *and stops spinning at* 12.0*.*

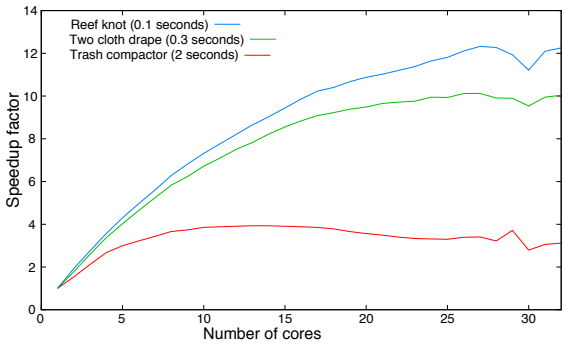

Figure 7: *Scaling behavior of our method for three examples for different numbers of available cores. Speedup factor is relative to our method run on only a single core. The reef knot is run for a short amount of time to show the behavior when the computation is dominated by material force updates. The trash compactor's poor scaling is due to the small problem size (less than a thousand degrees of freedom). The cloth drape is a more typical example of a large problem with many penalty contacts.*

including *safety*, no interpenetrations occur over the course of the simulation, despite the amount of self-contact.

Scaling of parallelism We ran several benchmark examples on an Intel 32-core Westmere-EX machine (E7-8837 @ 2.67GHz), and varied the number of cores our code was permitted to use. Figure 7 shows how the wall-clock time varies as a function of cores used. We observe reasonable scaling for up to 16 cores, with additional cores providing diminishing returns.

Profiling our code suggests to us that cache performance during event processing limits our scalability. When integrating bucketed force events, we first compute and store the force supplied by each force event, then iterate over the vertices and for each vertex look up and apply each relevant force's impulse contribution to that vertex's velocity. Splitting integration into these two steps allows us to parallelize both steps without synchronization, but increases each force event's cache footprint by requiring the impulses to be temporarily stored, and looking up that impulse during the velocity update randomly accesses memory. For processing internal forces, where the set of forces that affect each vertex is constant throughout the simulation, cache performance can be improved by partitioning the mesh into disjoint regions and processing each region in parallel (with special handling of the region boundaries), without intermediate storage. However, it is unclear how this strategy would extend to penalty force processing, and our initial experiments with partitioning the material forces resulted in only a modest (7%) performance improvement.

5.1 Choice of rollback window size

Our framework introduces one additional parameter not present in the original algorithm: the duration of the rollback window R. Whereas choice of this parameter can have dramatic effects on performance, it bears emphasis that the three guarantees are not compromised by any choice of R.

What is the optimal choice of R? Is it greater than zero, i.e. is there a point to rollback at all? In what follows, we give a theoretical argument that there exists a "sweet spot" for R, and study how the wall clock time of the benchmark examples vary with R.

Effect of R on number of rollbacks In a simulation with frequent collisions, increasing R increases the number of times a win-

dow rolls back on average, since during a large window it is more likely that the penalty forces that were added to fix one set of collisions will introduce a new set of collisions.

Effect of R on collision detection Consider an ideal simulation with frequent self-collisions uniformly distributed in time, and assume that the number of times the simulation rolls back does not change as R varies. Changing R changes the amount of time needed by collision detection for each rollback window. Fitting of the leaf nodes of the broad phase scales roughly as R, since computing the swept volumes of the simulation's triangles requires tracing the triangle's trajectory through the rollback window. The cost of the narrow phase is approximately proportional to $R \log R$, since the number of collisions is proportional to R and the cost of the divide-and-conquer separating slabs narrow phase is roughly logarithmic. Refitting interior nodes, and traversing the hierarchy, becomes more expensive as R increases (since traversal will reach the leaf level more often when more collisions are present) but has a non-trivial base constant cost independent of the size of the rollback window.

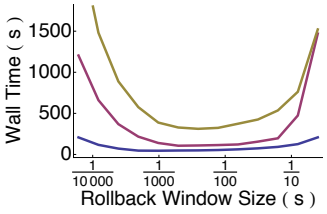

Figure 8: *The total wall clock time, as a function of R, for the trash compactor (*blue*), two-cloth drape (*maroon*), and reef knot (*tan*). The "sweet spot" for R does not vary much between simulations and the run time of the simulations is insensitive to small perturbations of R.*

Since the number of rollback windows in a simulation is inversely proportional to R, the total cost of the narrow phase is roughly $\log R$, whereas the overhead cost of BVH traversal decreases as R increases and grows unbounded as R shrinks. We therefore expect to minimize total cost at a sweet spot balancing these factors.

Effect of R on event processing The cost of event processing is proportional to the total window simulation time and resimulation time (assuming the cost of unnecessary penalty forces, etc. is negligible.) This simulation time is determined by the average number of times each window rolls back, which in turn increases with R. We thus expect wasted reprocessing work to increase with R.

Effect of R on unneeded penalty forces Unneeded penalty forces linger on the queue for two reasons: first, if multiple collisions are detected during a rollback window, it is possible that the penalty forces added to resolve one collision will, as a side-effect, also prevent the other collision, so that the forces added to prevent the second collision never act. Second, a penalty force might prevent a transient impact during one small part of a rollback window, and do nothing the remainder of the window. Both of these situations become more likely as R increases, so we expect the cost due to unneeded penalty forces to increase with increasing R.

Benchmark data We studied the effect of R on CPU time spent on collision detection, resimulation, and processing unneeded penalty layers, for three of our benchmark examples. Figure 9 plots this data. As expected, the cost of resimulating and of unneeded penalty forces increases with R, and the cost of collision detection forms a "U" shape, with the cost increasing as R becomes too small. In Figure 8 we plot the total wall clock time, as a function of R, for

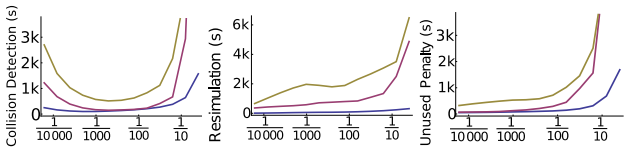

Figure 9: *We measured, for the trash compactor (*blue*), two-cloth drape (*maroon*), and reef knot (*tan*), the CPU time wasted on collision detection (*left*), resimulation (*middle*), and unneeded penalty forces (*right*) as a function of R. As we expect, resimulation and unneeded penalty forces increase as R increases, while collision detection time has a sweet spot and grows large as R becomes too small.*

the same examples. The total wall clock time behaves similarly to collision detection time, confirming that rollback is indeed useful: performing collision detection over a window is significantly cheaper than doing so after every event is processed, and the increase in other waste is modest.

We also observe from Figure 8 that small perturbations of R near the sweet spot does not significantly change the wall clock time of the simulation; hence performance does not depend critically on pinpointing the exact sweet spot. For all timings in this paper we simply used $R = 0.003$, which seems to work well for all of our examples; with more study and analysis we hope to provide, in the future, a heuristic for automatically selecting R, and an adaptive algorithm for adjusting R over the course of a simulation.

5.2 Parallel AVIs

Recent exciting work [Huang et al. 2007; Pingali et al. 2011] has examined parallelization of AVIs for physical systems without contact, and we studied the possibility of incorporating this work into our framework. Unfortunately, the available parallelism of our event queue does not appear sufficiently high for graph-based out-of-order execution to be profitable. If we leave internal and penalty forces bucketed, we have that the internal force bucket is connected to every other event (since the internal forces affect every vertex), and each penalty layer event is connected to every other penalty layer so that the event dependency graph is complete and the work-list never contains more than one event. Nevertheless, we hope to explore the possibility of combining ACM and Galois in future work, particularly for simulations involving cloth-shell coupling or other scenarios where the stable timesteps of the internal forces vary widely and naively bucketing them is expensive.

6 Conclusion

The ACM framework, as originally described in Harmon et al [2009], offered unparalleled correctness and robustness guarantees, but at a steep performance cost relative to other popular methods for simulating cloth, shells, and deformable bodies. By replacing ACM's KDS-based collision detection paradigm with one based on Time Warp, we both dramatically improve its efficiency and allow it to be easily parallelized, for a total of two orders of magnitude speedup over Harmon et al. This speedup is a significant step towards ACM being viable for production, and we look forward to future work, such as further investigation and improvement of our algorithm's scaling behavior, or ideas for adaptively selecting the rollback window size.

7 Acknowledgements

We thank Fang Da, Danny Kaufman, and Breannan Smith for their help and feedback during preparation of this paper. This research is supported in part by the Sloan Foundation, the NSF (CAREER Award CCF-06-43268 and grants IIS-09-16129, IIS-10- 48948, IIS-11-17257, CMMI-11-29917), and generous gifts from Adobe, Autodesk, Intel, mental images, NVIDIA, Side Effects Software, The Walt Disney Company, and Weta Digital.

References

BENDER, J., AND BAYER, D. 2008. Parallel simulation of inextensible cloth. In *Proceedings of VRIPhys*.

BORKAR, S., AND CHIEN, A. A. 2011. The future of microprocessors. *Commun. ACM 54*, 5 (May), 67–77.

BRIDSON, R., FEDKIW, R., AND ANDERSON, J. 2002. Robust treatment of collisions, contact, and friction for cloth animation. *ACM Transactions on Graphics 21*, 3, 594–603.

BRIDSON, R. 2003. *Computational aspects of dynamic surfaces*. PhD thesis, Stanford University.

BROCHU, T., EDWARDS, E., AND BRIDSON, R. 2012. Efficient geometrically exact continuous collision detection. *ACM Transactions on Graphics (TOG) – Proceedings of the ACM SIGGRAPH 2012*.

CUTHILL, E., AND MCKEE, J. 1969. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69 Proceedings of the 1969 24th national conference*, 157–172.

DEBUNNE, G., DESBRUN, M., CANI, M.-P., AND BARR, A. 2001. Dynamic real-time deformations using space and time adaptive sampling. In *Proceedings of SIGGRAPH 01*, 31–36.

GUIBAS, L. 1998. Kinetic data structures: a state of the art report. In *Proceedings of the 3rd Workshop on Algorithmic Foundations of Robotics (WAFR)*, 191–209.

HARMON, D., VOUGA, E., SMITH, B., TAMSTORF, R., AND GRINSPUN, E. 2009. Asynchronous contact mechanics. *ACM Transactions on Graphics (TOG) – Proceedings of the ACM SIGGRAPH 2009*.

HARMON, D., VOUGA, E., SMITH, B., TAMSTORF, R., AND GRINSPUN, E. 2011. Research highlights: Asynchronous contact mechanics. *Communications of the ACM*.

HARMON, D., ZHOU, Q., AND ZORIN, D. 2011. Asynchronous integration with phantom meshes. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*.

HARMON, D. 2010. *Robust, Efficient, and Accurate Contact Algorithms*. PhD thesis, Columbia University.

HUANG, J.-C., JIAO, X., FUJIMOTO, R. M., AND ZHA, H. 2007. DAG-guided parallel asynchronous variational integrators with super-elements. In *Proceedings of the 2007 summer computer simulation conference*, 691–697.

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

KALE, K., AND LEW, A. 2007. Parallel asynchronous variational integrators. *International Journal for Numerical Methods in Engineering 70*, 291–321.

KIM, D., HEO, J.-P., HUH, J., KIM, J., AND YOON, S.-E. 2009. HPCCD: Hybrid parallel continuous collision detection using cpus and gpus. *Computer Graphics Forum (Pacific Graphics) 28*, 7.

KONEČNÝ, P., AND ZIKAN, K. 1997. Lower bound of distance in 3D. In *Proceedings of WSCG 1997*, vol. 3, 640–649.

LEW, A., MARSDEN, J. E., ORTIZ, M., AND WEST, M. 2003. Asynchronous variational integrators. *Archive for Rational Mechanics and Analysis 167*, 85–146.

MIRTICH, B. 2000. Timewarp rigid body simulation. *SIGGRAPH '00 Proceedings of the 27th annual conference of computer graphics and interactive techniques*, 193–200.

PABST, S., KOCH, A., AND STRAER, W. 2010. Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. *Computer Graphics Forum 29*, 5, 1605–1612.

PINGALI, K., NGUYEN, D., KULKARNI, M., BURTSCHER, M., HASSAAN, M. A., KALEEM, R., LEE, T.-H., LENHARTH, A., MANEVICH, R., MÉNDEZ-LOJO, M., PROUNTZOS, D., AND SUI, X. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation*, 12–25.

SELLE, A., SU, J., IRVING, G., AND FEDKIW, R. 2009. Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE Transactions on Visualization and Computer Graphics*.

STAM, J. 2009. Nucleus: Towards a unified dynamics solver for computer graphics. In *IEEE International Conference on Computer-Aided Design and Computer Graphics*, 1–11.

SUTTER, H., AND ALEXANDRESCU, A. 2004. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Pearson Education, Inc.

TANG, M., MANOCHA, D., AND TONG, R. 2010. Mccd: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models 72*, 2, 7–23.

TANG, M., MANOCHA, D., LIN, J., AND TONG, R. 2011. Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 63–70.

THOMASZEWSKI, B., AND BLOCHINGER, W. 2006. Parallel simulation of cloth on distributed memory architectures. *Proc. Eurographics Symp. Parallel Graphics and Visualization*.

THOMASZEWSKI, B., PABST, S., AND BLOCHINGER, W. 2008. Parallel techniques for physically-based simulation on multi-core processor architectures. *Computers and Graphics 31*, 25–40.

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

VOUGA, E., HARMON, D., TAMSTORF, R., AND GRINSPUN, E. 2011. Asynchronous variational contact mechanics. *Computer Methods in Applied Mechanics and Engineering 200*, 2181–2194.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-oblivious mesh layouts. *ACM Trans. Graph. 24*, 3 (July), 886–893.

ZHENG, C., AND JAMES, D. L. 2011. Toward high-quality modal contact sound. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011) 30*, 4 (Aug.).