# Scalable Asynchronous Contact Mechanics using Charm++

Xiang Ni*, Laxmikant V. Kale* and Rasmus Tamstorf†

*Department of Computer Science, University of Illinois at Urbana-Champaign
†Walt Disney Animation Studios

***Abstract*— This paper presents a scalable implementation of the Asynchronous Contact Mechanics (ACM) algorithm, a reliable method to simulate flexible material subject to complex collisions and contact geometries. As an example, we apply ACM to cloth simulation for animation. The parallelization of ACM is challenging due to its highly irregular communication pattern, its need for dynamic load balancing, and its extremely fine-grained computations. We utilize CHARM++, an adaptive parallel runtime system, to address these challenges and show good strong scaling of ACM to $384$ cores for problems with fewer than $100k$ vertices. By comparison, the previously published shared memory implementation only scales well to about $30$ cores for the same examples. We demonstrate the scalability of our implementation through a number of examples which, to the best of our knowledge, are only feasible with the ACM algorithm. In particular, for a simulation of 3 seconds of a cylindrical rod twisting within a cloth sheet, the simulation time is reduced by $12\times$ from $9$ hours on $30$ cores to $46$ minutes using our implementation on $384$ cores of a Cray XC30.**

## I. Introduction

Thin materials like sheet metal, films, cloth, and composites have been studied extensively throughout the years and are used broadly in manufacturing. These studies often involve contact such as in crash simulations in the automotive industry and simulation of clothing in the garment and entertainment industries. A particularly difficult aspect of these simulations is the handling of impact and resting contact. Due to the thin nature of the objects, many algorithms suffer from tunneling artifacts where an object can end up going through itself or another object. These artifacts can be avoided using methods that are popular in graphics, e.g. [1], but often at the expense of violating other physical properties of the system such as conservation of momenta. In animation, these problems can force the artists to spend much time in finding acceptable compromises, while in scientific and engineering applications, the errors can quickly exceed acceptable thresholds.

The asynchronous contact mechanics (ACM) framework [2] is a reliable method to simulate flexible materials subject to complex collisions and contact geometries. It guarantees that no collisions are ever missed while retaining conservation of physical invariants. This is achieved by introducing a conceptually unbounded sequence of nested penalty layers for collision response. Furthermore, by using asynchronous integration, it localizes the computational efforts to the region(s) where needed.

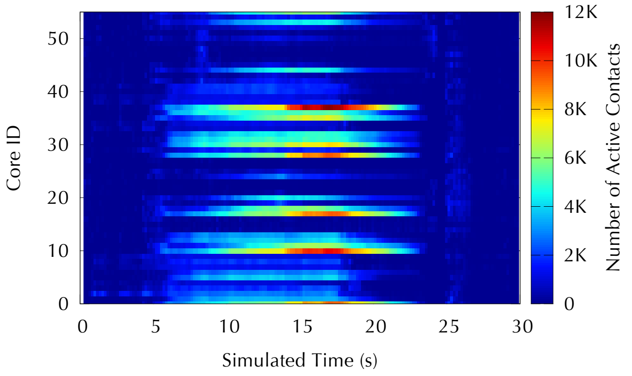


**Fig. 1:** *Dynamic changes in the number of active contacts on* $56$ *cores for a* $30s$ *simulation.*

The guarantees provided by ACM come at the cost of high computational requirements; the original algorithm is impractical for most real applications. The performance was improved by more than two orders of magnitude by introducing speculative execution and shared memory parallelization based on Intel's Threading Building Blocks (TBB) [3]. However, this implementation scales poorly beyond 10-15 cores, and the simulations remain expensive for interesting problems.

In this paper, we present a scalable parallel implementation of the ACM algorithm on distributed memory clusters using CHARM++, a parallel runtime system [4]. As a use case, we apply this to cloth simulation for animation. We conduct strong scaling experiments and show good scaling of our implementation in four examples with complex contact scenarios on a Cray XC30. To achieve this, we address three major challenges:

- **Highly irregular communication pattern.** The contact regions are unpredictable and change as the simulation progresses. Hence, communication patterns are unknown a priori. This dynamic message pattern fits naturally with the message-driven parallel model of CHARM++.
- **Dynamic load balancing.** When contact occurs, the corresponding collision response leads to more work in the region where the contact occurs. Figure 1 illustrates the dynamic variations in the number of contacts on a 56-core run for a 30s simulation of our twister example (see Section VII). The adaptive runtime system in CHARM++ is ideally suited to address this problem.

- **Very fine grained computation.** The average computation grain size is on the order of tens of microseconds at scale. The automated scheduling of the computation object in CHARM++ helps us execute them more efficiently and overlaps the computation and communication.

## II. Background and Related Work

In this section, we provide a brief introduction to asynchronous contact mechanics and CHARM++, and discuss the related work.

### A. *Asynchronous Contact Mechanics*

In essence, the original ACM algorithm implements a symplectic explicit integration scheme for a dynamic system using penalty forces for collision response. To advance time, all forces in the system are computed at fixed intervals and applied to vertices as impulses (kicks). Between these time steps vertices are allowed to drift, i.e., follow linear trajectories. At the end of a period of time, referred to as a "collision window", which includes multiple time steps, collision detection is performed (Figure 2). If any collisions are detected, then additional (and stronger) penalty forces are added to the system and time is rolled back to the beginning of the collision window; this is then re-simulated. In the event of rollback, the simulation state is restored to the snapshot that is taken at the beginning of a collision window. At the end of a collision window without any collisions, penalty forces that are no longer needed are removed, and the system proceeds to the next collision window.

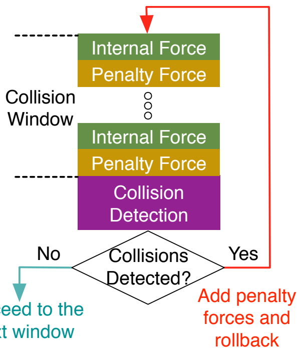


**Fig. 2:** *The overall flow of ACM.*

Due to the well-known CFL condition, stiff forces must be integrated with small time steps for the simulation to remain stable. Hence, as stronger and stronger penalty forces are added, the time steps must become smaller and smaller. Unfortunately, if tiny time steps are used globally, the simulation effectively grinds to a halt. Furthermore, using adaptive time stepping is problematic as it destroys the good properties of the symplectic integration scheme (preservation of momenta and approximate energy conservation).

A key insight in the ACM algorithm is the fact that the strong penalty forces are (usually) only needed to respond to very localized collisions. Hence, it is advantageous to use asynchronous integration as this effectively allows the time step size to vary spatially across the domain. Another key insight is that by decomposing the penalty forces into conceptually infinite sums of forces with finite stiffnesses, each one can be integrated using a fixed time step which preserves the good properties of the symplectic integration scheme.

For each force there is a clock, and whenever it ticks, a set of impulses are delivered to the set of vertices that it affects (its *stencil*). Since different forces have different stencils and time steps, a single vertex may receive impulses at irregularly spaced intervals in time. Furthermore, some vertices may receive impulses much more frequently than others. Effectively, this allows the computational effort to be spent where it is needed, but it also leads to load balancing challenges. In addition, the number of points needed to specify the trajectory of a vertex during a single collision window will vary between vertices. Hence, there is load imbalance in terms of computation as well as communication since the communication cost of trajectories needed for collision detection now varies across the domain as does the number of force evaluations.

### B. CHARM++

CHARM++ [4] is an over-decomposed object-based message-driven parallel runtime system. In CHARM++, the application domain is decomposed into multiple logical units, called *chares*, that are mapped to different cores. The number of chares is independent of the number of cores. Thus, applications can be run on any number of cores without being limited by the decomposition granularity.

Chares communicate via messages that are mediated by the CHARM++ runtime system (RTS). Typically, chares are reactive entities that are scheduled by the RTS when there are messages available for them. Thanks to the reactive nature of chares and the message-driven execution model, unexpected messages can be handled easily. This is especially important for irregular applications like ACM where the communication pattern is constantly changing. CHARM++ also allows programmers to guide the order of execution by specifying the priority of each message. Under the hood, the RTS uses this priority to decide the scheduling order of chares when messages are available for multiple chares.

CHARM++ supports two modes to run applications: non-SMP and SMP. In non-SMP mode, one CHARM++ process is launched on each physical core. Each process controls both the event processing and communication for all the chares mapped to it. In SMP mode [5], one CHARM++ process is launched on a set of cores within a physical node. Each process has a pool of worker threads and an associated communication thread that are mapped to distinct physical cores within the physical node. In the rest of the paper, we use the term node to refer to such a group of worker threads and communication thread. Worker threads on a node share the memory address space of the parent process, but execute their own independent schedulers. Each worker appears as an individual rank that has chares mapped to it, and processes messages in its local queue. The communication thread handles the communication for all other worker threads on the same node. A node-level queue is also shared by the worker threads. The workers process the messages in the node-level queue either when they do not have messages in their local queue or when the priority of messages in the node-level queue is higher than the priority of the messages in their local queue. Similar strategy has been explored by Kale et al. [6] in the context of MPI.

### *C. Related Work*

Prior work has considered cloth simulation on distributed memory clusters, but the earliest work in [7] and [8] did not consider collision detection and response at all. Also, this work did not scale beyond 16 cores. The work by [9] and [10] added collision handling, but both considered a more traditional synchronous integration scheme, and neither of them considered scaling beyond 16 cores.

More recently, some work has been done on distributed collision detection in games [11], and for engineering applications [12], but this work does not include the remaining parts necessary for a complete simulation system. Our work is based on [3] which considers a full simulation system using asynchronous integration, but only in the context of a shared memory system.

## III. Internal Force Calculation

Similar to previous works [3, 9], we partition the triangle faces of the cloth mesh, provided as an input to ACM, using METIS [13], and then distribute them among all the cores. Each partition is represented as a *chare* in Charm++; it is responsible for computation of both material forces and penalty forces within the partition. In this section, we focus on the computation of material forces, while we propose a way to decompose the penalty force calculation to ensure load balancing in Section VI.

The internal material forces include *stretch* and *bend* forces. Stretch forces are modeled using "constant strain triangles" [14], while bend forces are modeled using "discrete shells" [15]. Each triangle defines a stencil for the stretch force computation, while the stencil for the bend force computation consists of two adjacent triangles.

Communication is needed during the force computations if any vertex of a stencil is shared by more than one partition. We denote the stencils containing such vertices as "boundary stencils". The remaining stencils are "internal stencils". For boundary stencils, the partition to which the stencil belongs communicates the resulting forces for shared vertices to all other partitions overlapping with the stencil.

For bend stencils where the two triangles belong to different partitions, we arbitrarily assign the stencil to one of the two partitions. In this case, we also have an "external vertex", which is a vertex that does not belong to the partition that owns the stencil. Prior to the bend force computation, communication is necessary to obtain information for such external vertices. After the bend forces have been computed for the stencil, another round of communication is needed to update both shared and external vertices with force contributions.

Our goal for the internal force calculation is to *maximize the overlap of local computation and external communication*. Algorithm 1 shows the pseudocode for the internal force calculation. Each chare first starts the communication for bend force calculation (line 1). Next, we calculate the stretch force of the boundary stencils, so that we can asynchronously send the force contribution for the shared vertices as early as possible (line 2, 3).

**Algorithm 1** Internal force calculation

```
 1: sendExtVertexRequests()
 2: processBoundaryStretchStencils()
    // send stretch force contributions for shared vertices
 3: sendStretchForceUpdates()
 4: overlap
 5:   when recvExtVertex()
 6:     processBoundaryBendStencil()
 7:     if allBoundaryBendStencils.complete()
          // send bend force contributions for shared+external vertices
 8:       sendBendForceUpdates()
 9:     end if
10:   when recvBendForceUpdates()
11:     bufferBendForceUpdates()
12:   when recvStretchForceUpdates()
13:     bufferStretchForceUpdates()
      // local computations can be performed while waiting for messages
14:   localInternalCalc()
15: end overlap
16: foreach vertex do
17:   accumulateForceContributions()
18: end for
```

As suggested by its name, the **overlap** keyword (used in line 4) enables overlapped progress of multiple code blocks (or control flows), i.e. based on the availability of data, the Charm++ RTS atomically executes code fragments from different blocks in a mixed order. In Algorithm 1, these blocks are at line 5, 10, 12, and 14 representing computations of boundary bend stencils, buffering of boundary bend forces, buffering of boundary stretch forces, and computations of internal stencils, respectively. The computation block followed by the **when** keyword is triggered whenever the corresponding message is received. Hence, whenever we receive the positions of external vertices (line 5), we calculate the bend forces that depend on those vertices (line 6). Once all the bend forces have been computed, we send the force contributions for the external and shared vertices to the neighboring partitions (line 8). While waiting for the messages, local computation such as internal bend and stretch force calculation can be scheduled (line 14). This enables overlap of communication and computation, and helps mask message latency.

The change in velocity of a vertex is based on the forces accumulated from multiple stencils. It is important to ensure that these forces are accumulated in the same order for boundary vertices across different runs. Otherwise, the accumulated round-off error will lead to inconsistent results. Moreover, we also need to ensure in-order force updates for all the vertices when running on different number of cores so that the result is reproducible. For this purpose, each stencil is assigned a unique index, which is independent of the number of cores the application is run on. During the computation, each force contribution is initially stored in a temporary array of the target vertex (line 11, 13). In the end, each vertex accumulates the forces contributions ordered by the indices of the stencils that generated the forces (line 17).

## IV. Broad Phase Collision Detection

After several iterations of material force calculations described in the previous section, at the end of a collision window, collision detection is performed. For example, our twister simulation performs about 140 material force calculations in a collision window. Broad phase collision detection is the first step in the collision detection used to quickly identify a potential set of collisions.

Broad phase collision detection is conducted differently for collisions within a partition and for collisions among different partitions. Locally inside each partition, a bounding volume hierarchy based on discrete oriented polytopes with 26 bounding planes (26-DOPs) is used to detect potential collisions [3]. Globally among all the partitions, we leverage an existing voxel-based parallel collision detection library in Charm++ [16]. Both methods aim to quickly eliminate non-colliding collision pairs of primitives and return the remaining pairs as potentially colliding pairs.

The existing collision detection library in Charm++ uses an axis-aligned bounding box to bound the volume swept by a triangle. This method is extremely fast but it fails to fit the object as well as the 26-DOPs based method. As a result, we found that it finds more potential collisions compared to the 26-DOPs method used in the TBB implementation [3].

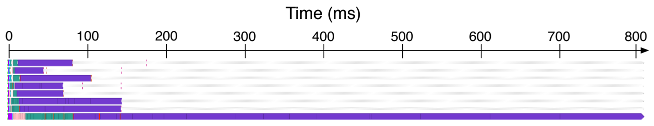
To solve this problem, we add an extra phase to further filter the output of the Charm++ library using the 26-DOPs method in parallel. Though adding an extra phase leads to more computation, the time spent on the added computation is orders of magnitude less than the time saved in the following narrow phase detection due to this optimization.

## V. Narrow Phase Collision Detection

After the broad phase, narrow phase collision detection is performed to find the real collisions using a more expensive method. The input to the narrow phase collision detection is a list of potentially colliding pairs of primitives. Each pair consists of either two edges or a vertex and a face. A pair may contain primitives that belong to the same partition (intra-partition collision) or are from two different partitions (inter-partition collision). We apply the space-time separating planes method from [3] to cull collisions, but to achieve good performance in a distributed system, *careful consideration of load-balancing of both computation and communication* is required. We elaborate on this in the remainder of this section.

### A. Computation Imbalance

In the base implementation of the narrow phase, the potential collision pairs are distributed evenly among all the cores. Figure 3(a) shows a timeline of a representative set of cores for narrow phase from a run on 96 cores. This view is generated using Projections [17], the performance analysis tool for Charm++. In the timeline view, the $x$ axis represents the execution time while the $y$ axis shows the cores that are being traced. Different colored horizontal bars represent different kinds of work performed by these cores. In this paper, we always present the timeline profiles for a few representative cores that show the general trend of all the cores.



(a) computation imbalance in the base version (810ms).

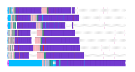

(b) communication imbalance after profiling-based load balancing(150ms).

**Fig. 3:** *Timeline profile of representative cores from runs on* 96 *cores of the twister example at* $0.7s$ *into the simulation. Computation in narrow phase is shown as the purple bars. Blue bars show the time spent on communication.*

Purple bars in Figure 3 represent the computation time spent in the narrow phase on a core. The important observation from Figure 3(a) is that the length of purple bars, and hence the computation time, is not equal for different cores. This is despite the fact that the potential collision pairs are evenly distributed. The reason for this is that the time spent on each potential collision pair depends on the trajectory length of each of the vertices that make up the pair. For vertices that are already affected by multiple active penalty layers, their trajectories are longer than the trajectories of other vertices.

To overcome the computation imbalance, we profile the time spent on each potential collision pair during the narrow phase. The information from the previous narrow phase is then used to *predict the computation time* of a collision pair in the current phase, and based on this we distribute the load as evenly as possible. This strategy works well because the number of penalty layers attached to a vertex changes slowly, and hence the length of the trajectory for the vertex also changes slowly.

One problem with this approach is that the bookkeeping overheads for profiling each individual potential collision pair and the overheads of matching the data from the previous phase to the data in the next phase is quite high, especially as the problem size increases. Hence, in order to reduce these overheads, we treat all the potential collision pairs that belong to the same pair of partitions as equals, i.e. we assume that they all have the same detection time due to their spatial locality.

Figure 3(b) shows the timeline profile of the same representative cores shown in Figure 3(a) for a narrow phase run with the profiling-based load distribution approach. *The overall time is reduced by* $5\times$; it can be seen that the purple bars are now much more evenly distributed, demonstrating an improved load estimation and distribution of collision pairs. However, we now observe that the blue blocks, which

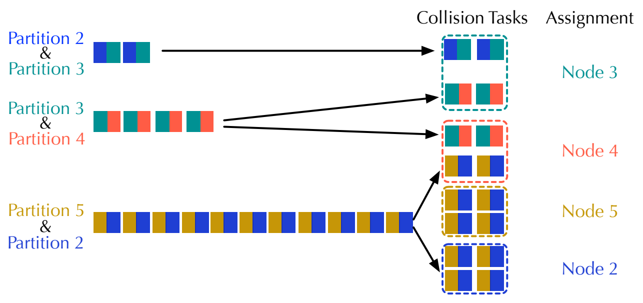

Collision Tasks
Assignment
Partition 2 & Partition 3
Partition 3 & Partition 4
Partition 5 & Partition 2
Node 3
Node 4
Node 5
Node 2

**Fig. 4:** *Construction and assignment of collision tasks. A potential collision pair is shown as a dual-color block, representing the partition pair it belongs to. We first group the potential collision pairs based on the associated partition pairs. Next, we decompose each group to ensure that the computation time of each collision task is below a predefined threshold. Then, we apply Algorithm 2 to distribute the collision tasks. In this example, we assume one partition per node and equal computation time for each collision pair.*

represent communication, are not evenly distributed.

## B. Locality Aware Load Balancer

The communication imbalance in Figure 3(b) is because detection takes longer for the collision pairs that belong to the partition on the bottom-most core in the figure. As a result, the collision pairs from this partition are distributed to many other cores, who all need to query the trajectory information from the bottom-most core.

This motivates us to take the node locality into consideration when distributing the work using profiled data. Thus, with the help of the SMP mode in CHARM++, we consider each physical node as an execution unit for distributing the collision pairs. We prioritize to assign the expensive collision pairs, e.g. associated with partition $A$, to the node that contains the core which partition $A$ resides on. Due to such locality based assignment, the communication cost can be reduced.

In the new locality aware load balancer, a master node is used to perform load balancing decisions using Algorithm 2. The input to this algorithm is constructed using meta information sent to the master node by all other nodes. The meta information is constructed from the list of potential collision pairs each node holds after the broad phase. Note that sending the entire list is not scalable as the problem size and number of nodes increase, and thus meta information is used. Here, the meta information is the number of potential collision pairs associated with each partition pair. Due to the work distribution in the collision library, each node may hold partial potential collision pairs that belong to multiple partition pairs.

Upon receiving all of the meta information, the master node transforms it into a list of potential collision tasks. Figure 4 shows how each task is formed. First, we group the potential collision pairs according to the partition pairs they belong to. As illustrated in Figure 4, there are three groups after this step. If the computation time of a group is beyond certain threshold, we further decompose it for ease of distribution. Note here that the computation time of each task should exceed the communication cost of distributing it. In the end, the three groups are decomposed into 8 potential collision tasks as shown in the Collision Tasks column in Figure 4.

**Algorithm 2** Locality aware load balancer

```
Input: $N \leftarrow$ number of nodes
Input: $L \leftarrow$ list of potential collision tasks
Output: $A[1..N] \leftarrow$ an array of the list of collision tasks
    assigned to each node
1:  $Load[1..N] \leftarrow 0$
2:  Set $avgLoad$ to be the expected average load per node
3:  for each task $T$ in $L$ do
4:      $P_1, P_2 \leftarrow$ nodes that the two partitions associated with
        $T$ reside on
5:      if $Load[P_1]$ + $T$.load $< avgLoad$ then
6:          $Load[P_1]$ = $Load[P_1]$ + $T$.load
7:          Push($A[P_1]$,$T$) // adding $T$ to $P_1'$s work list
8:      else if $Load[P_2]$+$T$.load $< avgLoad$ then
9:          $Load[P_2]$ = $Load[P_2]$ + $T$.load
10:         Push($A[P_2]$, $T$)
11:     else
12:         Push($PL$, $T$) // adding $T$ to the pending work list
13:     end if
14: end for
15: sort($PL$) // sort $T$ in $PL$ based on load from biggest to smallest
16: for each task $T$ in $PL$ in sorted order do
17:     $Load[P]$ = $Load[P]$+$T$.load
18:     Push($A[P]$, $T$) // Assign $T$ to the least loaded node $P$
19: end for
```

In Algorithm 2, for each potential collision task, we try to assign it to a node that holds one of the partitions the task belongs to (lines 4 to 10). We refer to such node as a home node. If after adding the task, the load on a home node will be greater than the average load per node (calculated based on the total load of all the tasks), then we push the task to the pending list (line 12) in order to process it in the second round. Otherwise, the task is assigned to a home node. The second round is essentially a greedy load balancer: we assign the tasks in the decreasing order of their loads to the node that currently has the least load (lines 16 to 18).

At the end of Algorithm 2, each collision task is assigned to a certain node. However, the detailed information about potential collision pairs is still scattered among all the nodes. To address this, the master node sends instructions to each node on how to send its potential collision pairs to their target nodes assigned by Algorithm 2.

## C. Node Aware Narrow Phase Detection

The locality aware load balancer can help balance the work load among all the nodes while limiting the communication. However, inside each node, some cores may still get more requests for trajectories than others. To handle this imbalance, we implement a node aware approach for narrow phase detection illustrated in Algorithm 3. In our approach, the cores that have less communication can naturally offload the computation work from the cores with more communication on the same node.

**Algorithm 3** Narrow phase detection

```
Input: $L \leftarrow$ list of potential collision tasks on each node
      // send data request MSGs for the external vertices in L
 1: sendDataRequest()
 2: while $L$ is not empty do
 3:     when recvMsg($M$)
        // receives are strictly prioritized in the order of case blocks
 4:       switch $M$.type()
 5:       case Data Request:
 6:          sendDataReply()
 7:       case Data Reply:
 8:         if $T$.ready()
            // Task T has got all the trajectories needed
 9:           if $T$.size() $>$ THRESHOLD
              // decompose T into subtasks and redistribute them
10:             Send within-node subtask work request messages
11:           else Process $T$
12:       case Work Request:
13:          Process subTask($M$.start, $M$.end)
14: end while
```

As shown in Algorithm 3, each node first sends data requests for trajectories of external vertices (line 1). Thereafter, different actions are taken based on the type of the message received. Using priorities allowed by the Charm++ RTS, the algorithm ensures that the messages are scheduled in the following order: data request, data reply, work request. This is done to ensure that the critical path is sped up.

On receiving a data request message $M$ (line 5), core $P$ replies with the requested trajectories to the sender node $N$. Once the data reply messages is received by node $N$, it can be processed by any core, $P'$, that becomes available on node $N$ (line 7). Core $P'$ checks whether all the data needed to process the corresponding collision task is available (line 8). If so, it then checks how many potential collision pairs are contained in that task $T$ (line 9). When the number of potential collision pairs is below the pre-defined threshold, task $T$ is processed locally by $P'$ (line 11). Otherwise task $T$ is further decomposed so that other cores on the same node can contribute to complete the task (line 10).

These decomposed subtasks are distributed through the node-level queue ($Q$) in the CHARM++ RTS, which is shared by all the cores on the same node. Each collision task is essentially a list of potential collision pairs as described in Figure 4. Thus, each subtask is represented as a tuple of *start index* and *end index*, specifying a subrange of the original task. For decomposing a large task, core $P'$ sends several within-node work request messages containing the *start index* and *end index* of each subtask (line 10). Internally these messages are pushed into the node-level queue $Q$ by the CHARM++ RTS. Later, when any core on the same node as $P'$ is available, the CHARM++ RTS dequeues the work request message from $Q$ and provides them to the core (line $12, 13$).

The task decomposition described in the previous paragraph is needed because any long computation may delay the processing of data request messages. Replying to data request messages promptly can help reduce the critical path and allows the remote nodes to start processing the collision tasks. Thus, processing the data request message is given higher priority in comparison to other messages.

## D. Node Level Data Cache

Our implementation is further enhanced by the use of a node-level software data cache. This cache contains pointers to the trajectory information for all the vertices on that node. The use of the node-level cache not only balances the communication requests within each node but also reduces communication within a node. With a node-level data cache, before sending a data request message for an external vertex, a core first checks whether the node-level cache contains the vertex. If the data is available locally, extra communication is avoided. Otherwise, a data request message is sent; this message is no longer targeted to a certain core, but to the node of the core. When a data request is received on a node, whichever core becomes available first on that node processes it. This is possible because any core can look up the data pointer in the cache for a correct reply.

The node-level data cache is especially helpful in reducing the communication load on the cores that receive many data requests. As a positive side effect, this optimization also reduces the critical path since the requests are served earlier than the default case, in which they have to wait for a specific core on the home node to become available.

In the future, we plan to extend the node-level software cache to store the data reply message as mentioned in [18]. Thus, if two cores on the same node need the same data, only one request is sent. This will reduce the number of data request messages and improve performance.

## E. Optimized Results

Figure 5 shows the time spent in the narrow phase of one round of collision detection when running on 12 to 192 cores using different optimization strategies. As can be seen in Figure 5, profiling based approach reduces the narrow detection time by up to $80\%$ in comparison to the base implementation with simple load balancing. It also helps improve the scalability as we run the narrow phase on larger core counts. The fully optimized version, which includes the profiling and locality based load balancing and node-level optimizations, further reduces the narrow phase timing by 70% and provides much better scaling. The cost of the locality aware load balancing is minimal: $1.5$ ms for the run on 192 cores. All in all, the narrow phase detection time on 192 cores is sped up by $26\times$ from 780 ms to 30 ms using the optimizations presented in this section.

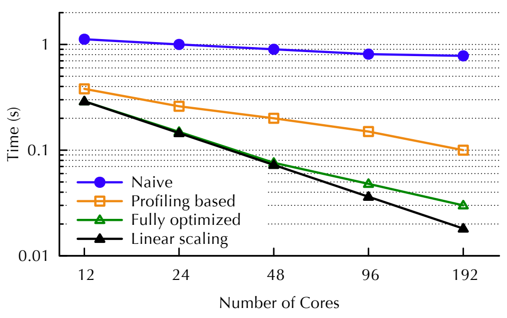


**Fig. 5:** *Time spent on narrow phase detection using different strategies at* $0.7$ *simulated second of the twister example.*

## VI. Collision Response

A penalty force calculation is added for each collision pair of material primitives (edge-edge or vertex-face) found by collision detection. The uneven distribution of collision regions leads to an imbalance in the penalty force calculation required to avoid collisions. Figure 6(a) shows the Projections timeline view for 15 cores of a node, in which the yellow bars denote penalty force computations. It is easy to see that the length of the yellow bars is different among the 15 cores; in particular, core 6 is overloaded with penalty force calculations. As a result, there is substantial idle time (shown as white bars) on other cores due to the direct and indirect communication dependence they have with core 6.

### A. Intra-node Work Redistribution

In Figure 6(a), it would be best if other cores on the same node as core 6 could help to offload some of the penalty force calculations to improve the overall utilization. We achieve this by making use of the SMP mode in Charm++. As described in Section II, in SMP mode, all the cores within a node share memory address space and a node level queue. Therefore, to distribute the work, the overloaded core decomposes its penalty force calculations into small work pieces and distribute them through the node level queue in a similar way as discussed earlier in Algorithm 3. Algorithm 4 presents the pseudocode for the intra-node work redistribution of penalty force calculations.

If a core is overloaded, we first decompose all its penalty force calculation work into smaller work units that can be distributed and then push them to the shared queue (lines 3 to 8). Otherwise, the penalty force computations are conducted locally (line 10). When a core has finished all its local work (under-loaded core) or has pushed its work to the queue (overloaded core), the Charm++ RTS automatically checks the status of the shared node level queue (lines 12 to 15) and schedules the work in it to available cores. The effectiveness of this optimization can be seen in Figure 6(b); the yellow bars on core 6 are shorter due to help from cores 8 and 14. This reduces the penalty force computation time by up to 12%.

### B. Node Level Phase Barrier

Ideally, in Figure 6(b), we would want all the less loaded cores to help core 6 for penalty force calculations. However, as can be seen, this is not the case; only core 8 and 14 share the node level tasks with core 6. This is because of the *uncoordinated progress made by the cores within a node*. When the under-loaded cores finish their penalty calculations and check the node level queue for additional work, the queue is typically empty. This is because overloaded cores are delayed by their own penalty force calculations. As a result, the under-loaded cores begin the next iteration of heavy internal force calculations (shown in green bars). Later, when the overloaded cores try to distribute the work, the under loaded cores cannot be involved since they are busy with the internal force calculations.

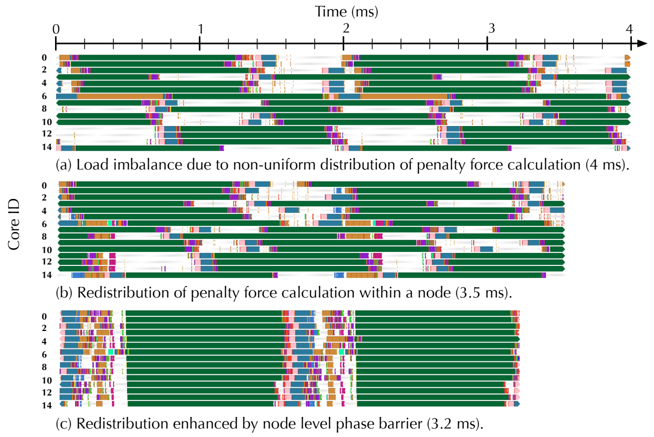


**Fig. 6:** *Timeline profile for all the cores on the same node. These are runs on* $128$ *cores for the twister example at* $0.7s$ *into the simulation. Yellow bars denote the time spent on penalty force computations while internal force computations are shown in green. White bars indicate idle time.*

**Algorithm 4** Penalty force calculation

```
Input: $Q \leftarrow$ shared common node level queue
Input: $N \leftarrow$ number of penalty force primitives per core
Input: $n \leftarrow$ minimal number of the penalty force primitives
    per work unit
 1: if overloaded with penalty force computation then
 2:     $left \leftarrow N$, $start \leftarrow 0$
 3:     while $left \neq 0$ do
 4:         $worksize = min(left, n)$
 5:         $end = start + worksize$
            // Create small work unit and then push to the queue
 6:         Push($Q$, PenaltyWork($start$, $end$))
 7:         $start = end$, $left = left - worksize$
 8:     end while
 9: else
10:     Do all the penalty force calculations
11: end if
12: while $Q$ is not empty do
13:     PenaltyWork $work$ = Pop($Q$)
14:     Process $work$
15: end while
```

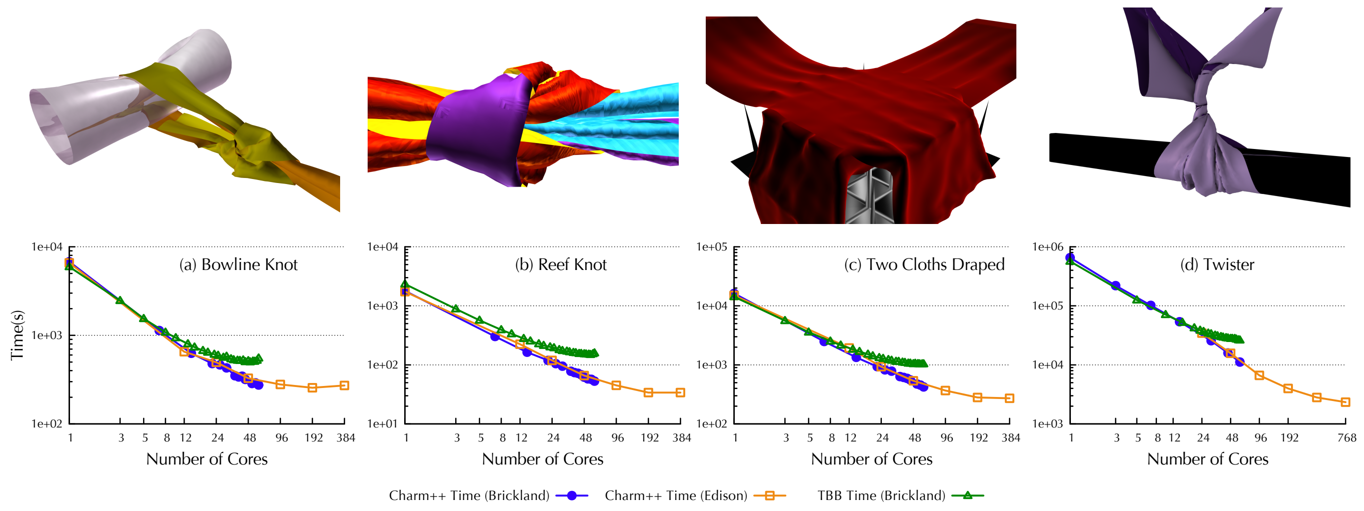


**Fig. 8:** *Performance comparison between* CHARM++ *and TBB versions of ACM. Benchmarks are from Ainsley et al. [3]. From left to right: bowline knot (3995 vertices, 5.0 s), reef knot (10642 vertices, 2.0 s), two cloths draped (15982 vertices, 3.95 s) and the short twister (99942 vertices, 3.0 s). Results for the* CHARM++ *version are obtained from both Edison and Brickland.*

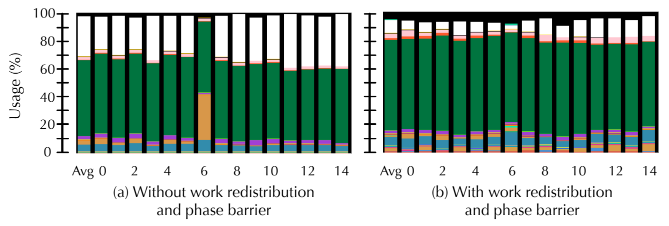


**Fig. 7:** *Utilization before and after the optimization.*

To address this mismatched progress issue, we add a node-level phase barrier before the start of the internal force calculations. As a result, the under loaded cores now wait for the overloaded cores to distribute the work and can assist them. As seen in Figure 6(c), yellow bars, i.e. penalty force calculation tasks, are now distributed to every core within a node. Hence, the total computation time is further reduced by 9%. Figure 7 shows the utilization gains achieved by the optimizations for collision response. The average utilization improves from 68% to 82%, while the total time goes down by 20%.

## VII. Analysis and Results

### A. Experimental Setup

We use four out of five examples from Ainsley et al [3] (shown in Figure 8) to evaluate the performance of the CHARM++ version of ACM. The last of the original examples is omitted due to its small size (only 719 vertices). The twister example in our simulations has $100k$ vertices whereas the one reported in [3] only had 8000 vertices. We refer to the first 3 seconds of the twister simulation as the *short twister*, while the full 50 second simulation is referred to as the *long twister*.

The experiments were conducted on two systems. One is Edison at NERSC, a Cray XC30 machine based on Intel E5-2695@2.4GHz (12 core Ivy Bridge) processors. The other is Brickland which is a 4 socket system with Intel E7-4890@2.8GHz (15 core Ivy Bridge) processors and a total of 60 cores. In particular, we use the latter to re-run the TBB version of ACM alongside the CHARM++ implementation. Edison is used to scale beyond what is possible in a shared memory system, but we note that the performance for Edison is comparable to that of Brickland for runs using less than 60 cores. The main difference appears to be due to the difference in clock speed between the two systems. For the experiments of the CHARM++ version, SMP mode is used; cores within a socket are grouped together to provide the worker threads and the communication thread of a CHARM++ node.

### B. Scaling Analysis

The green curves in Figure 8(a) to (d) shows that the TBB version stops showing good scaling after 30 cores. In contrast, the CHARM++ version shows good scaling and is faster than the TBB version by 2× even on one node at the same core count (60) for all the examples. The good scaling continues beyond one node for the CHARM++ version; the simulation time is further reduced when running on hundreds of cores of Edison. For the three relatively small examples (bowline knot, reef knot and two cloths draped), the CHARM++ version stops scaling after 192 cores. However, it is important to observe that with that many cores, there are only tens of vertices per core for these small examples. For the biggest example, the twister, the scaling continues well beyond 192 cores till 384 cores. The compute time for 3 seconds of simulated time is reduced from 3.1 hours on 60 cores of Brickland to 46 mins on 384 cores of Edison using the CHARM++ version. These results show that the fully optimized CHARM++ version handles the extremely fine-grained communication of ACM on 384 cores very well (one message every 30 us).

Table I lists the best performance achieved using both the TBB and the CHARM++ version of ACM. The speedup is the

| Example | Best Performance with TBB (#cores) | Best Performance with CHARM++ (#cores) | Speedup |
|---|---|---|---|
| Bowline Knot | 8.4 mins (50) | 4.3 mins (192) | 2.0 |
| Reef Knot | 2.5 mins (55) | 34 secs (192) | 4.4 |
| Two Cloths Draped | 17.3 mins (60) | 4.5 mins (384) | 3.8 |
| Short Twister | 7.3 hrs (60) | 39 mins (768) | 11.2 |

**TABLE I:** *Best performance achieved using TBB and* CHARM++ *version of ACM.*

ratio between the best performance numbers. We note that the TBB implementation has a 17% advantage due to the clock speed difference between Brickland and Edison, which we have not adjusted for. For all the four example, the speedup of the CHARM++ version is at least 2×. For the short twister example, by increasing the hardware resources by 13×, we see that we are able to reduce the simulation time by 11×.

### *C. The Long Twister*

The twister example simulates a twisting rod wringing out a sheet of cloth. Torque is applied to the rod from 3s to 5s, but inertia keeps it wringing the cloth harder and harder until about 12$s$. At that point, the cloth starts unwinding and eventually the rod falls out. We ran the full 50s simulation on 60 cores of Brickland. Figure 9 shows the average time taken for every 10th collision window during the 50s simulation.

We study the scaling of the twister example at different representative times into the simulation: 5s, 10s, 15s and 25s. To speedup the experiments, we restart the run from checkpoints taken at the representative times. Figure 10 shows the time taken for one collision window of the twister example at different stages in simulation from 1 core to 768 cores on Edison. The time measured here is for one successful collision window without rollbacks. From Figure 10, we can see that the CHARM++ version of ACM scales very well to 384 cores at all stages of the simulation. The collision detection time overtakes the time spent on force calculation when there are more collisions (at 10s and 15s). When there are fewer collisions (at 5s and 25s), the collision detection time does not scale well at the limits of strong scaling. As we scale from 384 cores to 768 cores, limited parallelism results in collision detection time reducing marginally from 33ms to 32ms as shown in Figure 10(d). Overall, both the force calculation and collision detection scale well to 384 cores at all stages of the simulation.

### *D. Parallel Efficiency*

Figure 11 shows the parallel efficiency for the TBB version and CHARM++ version of ACM. Due to the time constraints, we did not acquire the result for the CHARM++ version running on one core for the short twister example. Thus the parallel efficiency of the CHARM++ version on Edison is based on the time taken for one collision window at 25 simulated seconds while the parallel efficiency for the TBB version is based on the short twister example. As shown in Figure 11, for all the three smaller examples, at the same core count (48), the parallel efficiency of the CHARM++ version is more than twice that of the TBB version. For the twister

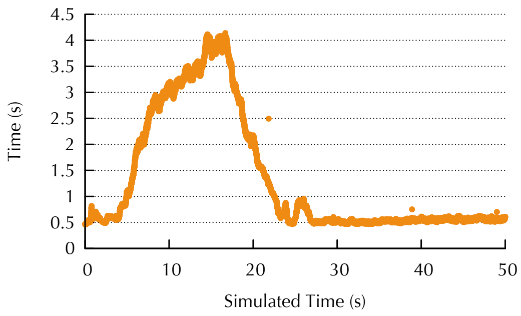


**Fig. 9:** *Average compute time for every 10th collision window including rollbacks. The data is for the long twister example on Brickland using* 60 *cores. Each window represents 1/3000 seconds of simulated time.*

example, the parallel efficiency of the CHARM++ version remains as high as 60% for execution on 96 cores after which it gradually decreases to 20% on 768 cores. In contrast, the efficiency of TBB version falls below 40% even for 48 cores.

## VIII. Limitations and Future Work

We have made no attempt at reducing the actual number of computations required for a given simulation in this work. As such the ACM algorithm remains an expensive approach for running cloth simulations, but it provides a number of guarantees that the cheaper approaches cannot offer. If we consider the frame rate, then our current performance (3 simulated seconds of 100k vertices in 46 minutes on 384 cores) corresponds to less than 40 seconds per frame (assuming 24 frames per second). This is comparable to current simulation times for feature animation work, but still much more expensive in terms of compute resources.

One of the primary reasons that limits scaling of the CHARM++ version of ACM beyond 384 cores is the penalty force calculation. So far, we have addressed the load balancing issue across nodes only for collision detection. In contrast, the penalty force calculation is load balanced only within a node. In the future, we plan to leverage the over-decomposition feature of CHARM++ in the force calculation phase as well to dynamically redistribute chares to achieve better load balance across nodes: migrating chares from overloaded nodes to under loaded nodes. We also plan to integrate the load balancers we have presented to the CHARM++ runtime system; currently they are implemented as part of our application.

## IX. Conclusion

In this paper, we have described the design and optimizations necessary to parallelize ACM using CHARM++. We have shown strong scaling up to 384 cores of Edison for relatively small-sized problems with less than 100$k$ vertices, i.e. about 300 vertices per core. The improved scaling has led to a speedup of more than 10× compared to previously published results, which makes it practical to use the ACM algorithm at the expense of more compute resources.

By way of the above, we have also demonstrated that CHARM++ is well-suited for dynamic irregular applications

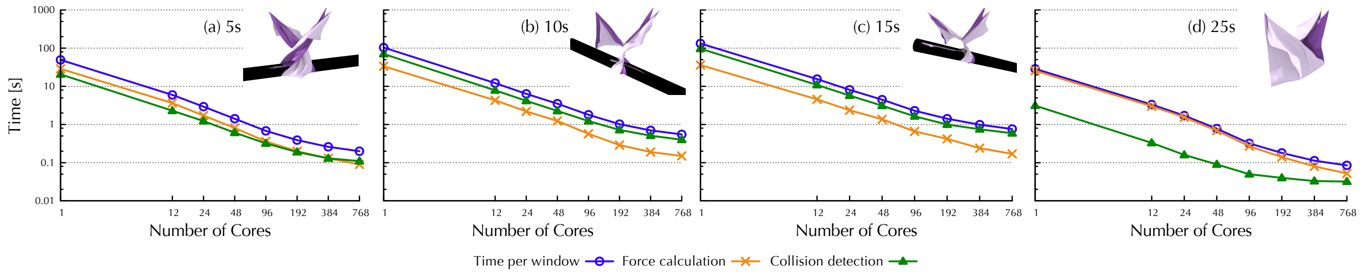


**Fig. 10:** *Time taken for one collision window of the twister example on Edison at different times into the simulation.*

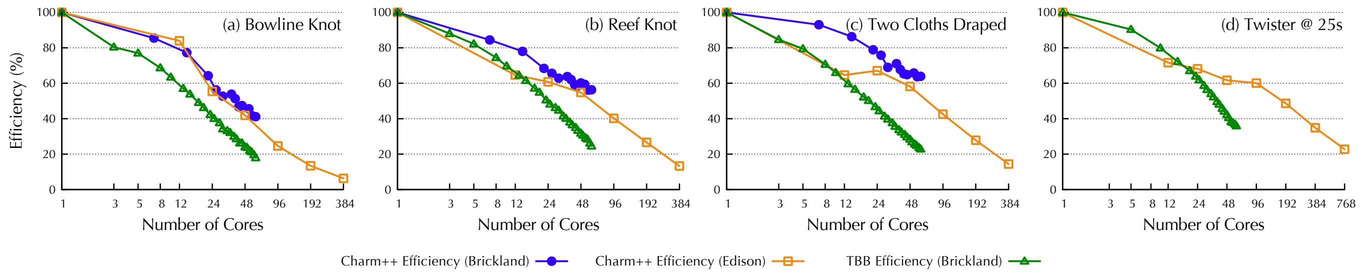


**Fig. 11:** *Parallel efficiency for TBB and* CHARM++ *version of ACM. Parallel efficiency is measured as the ratio between the actual speedup and linear speedup.*

like ACM. The message-driven execution feature eases the handling of the dynamic message patterns of ACM and helps overlap communication and computation. Furthermore, we have seen that the SMP mode of CHARM++ helps exploit the shared memory multiprocessor node used in HPC systems and improves the scalability of ACM.

## ACKNOWLEDGMENTS

The authors would like to thank Intel for providing access to the Brickland machine. This research used resources of the National Energy Research Scientific Computing Center, supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Part of this work was done while the first author was at WDAS or was funded by WDAS.

## REFERENCES

[1] R. Bridson, R. Fedkiw, and J. Anderson, "Robust treatment of collisions, contact and friction for cloth animation," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 594–603, Jul. 2002.
[2] D. Harmon, E. Vouga, B. Smith, R. Tamstorf, and E. Grinspun, "Asynchronous Contact Mechanics," in *ACM Transactions on Graphics (TOG)*, vol. 28, no. 3. ACM, 2009, p. 87.
[3] S. Ainsley, E. Vouga, E. Grinspun, and R. Tamstorf, "Speculative Parallel Asynchronous Contact Mechanics," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, p. 151, 2012.
[4] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC'14. New York, NY, USA: ACM, 2014.
[5] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé, "Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study," in *TeraGrid'10*, no. 10-13, Pittsburgh, PA, USA, August 2010.
[6] V. Kale and W. Gropp, "Load balancing for regular meshes on smps with MPI," in *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*, 2010, pp. 229–238.
[7] F. Zara, F. Faure, and J.-M. Vincent, "Physical Cloth Simulation on a PC Cluster," in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, ser. EGPGV '02. Eurographics Association, 2002, pp. 105–112.
[8] ——, "Parallel Simulation of Large Dynamic System on a PC Cluster: Application of Cloth Simulation," *International Journal of Computers and Applications*, vol. 26, no. 3, pp. 173–180, 2004.
[9] B. Thomaszewski and W. Blochinger, "Physically based simulation of cloth on distributed memory architectures," *Parallel Computing*, vol. 33, no. 6, pp. 377 – 390, 2007, parallel Graphics and Visualization.
[10] A. Selle, J. Su, G. Irving, and R. Fedkiw, "Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 15, no. 2, pp. 339–350, March 2009.
[11] T. Chen and C. Verbrugge, "A protocol for distributed collision detection," in *Network and Systems Support for Games (NetGames), 2010 9th Annual Workshop on*, Nov 2010, pp. 1–6.
[12] M. Anderson, M. Brodowicz, L. Dalessandro, J. DeBuhr, and T. Sterling, "A Dynamic Execution Model Applied to Distributed Collision Detection," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 470–477.
[13] G. Karypis and V. Kumar, "A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[14] M. J. Turner, R. W. Clough, H. C. Martin, and L. P. Topp, "Stiffness and deflection analysis of complex structures," *J. Aeronautical Society*, vol. 23, no. 9, pp. 805–824, 1956.
[15] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder, "Discrete shells," in *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 2003, pp. 62–67.
[16] O. S. Lawlor and L. V. Kalé, "A Voxel-Based Parallel Collision Detection Algorithm," in *Proceedings of the 16th International Conference in Supercomputing*. ACM, June 2002, pp. 285–293.
[17] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling Applications to Massively Parallel Machines Using Projections Performance Analysis Tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.
[18] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, "Adaptive Techniques for Clustered N-Body Cosmological Simulations," *arXiv preprint arXiv:1409.1929*, 2014.