HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs

Duksu Kim Jae-Pil Heo Jaehyuk Huh John Kim Sung-eui Yoon

KAIST (Korea Advanced Institute of Science and Technology)
Project URL: http://sglab.kaist.ac.kr/HPCCD

Abstract

We present a novel, hybrid parallel continuous collision detection (HPCCD) method that exploits the availability of multi-core CPU and GPU architectures. HPCCD is based on a bounding volume hierarchy (BVH) and selectively performs lazy reconstructions. Our method works with a wide variety of deforming models and supports self-collision detection. HPCCD takes advantage of hybrid multi-core architectures – using the general-purpose CPUs to perform the BVH traversal and culling while GPUs are used to perform elementary tests that reduce to solving cubic equations. We propose a novel task decomposition method that leads to a lock-free parallel algorithm in the main loop of our BVH-based collision detection to create a highly scalable algorithm. By exploiting the availability of hybrid, multi-core CPU and GPU architectures, our proposed method achieves more than an order of magnitude improvement in performance using four CPU-cores and two GPUs, compared to using a single CPU-core. This improvement results in an interactive performance, up to 148 fps, for various deforming benchmarks consisting of tens or hundreds of thousand triangles.

1. Introduction

Collision detection between deforming models is a fundamental technique in various applications including games, physically-based simulation, CAD/CAM, and computer animation. Collision detection is classified as two categories: discrete and continuous methods.

Discrete collision detection (DCD) finds intersecting primitives at discrete time steps. DCD can be performed quite efficiently, but may miss colliding primitives that occur between two discrete time steps. This issue can be quite problematic in physically-based simulations, CAD/CAM, etc. On the other hand, continuous collision detection (CCD) identifies intersecting primitives at the first time-of-contact (ToC) during a time interval between two discrete time steps. Typically, CCD is performed by using bounding volume hierarchies (BVHs) of input models. The BVHs are hierarchically traversed to find contacts among models. At the leaf nodes of the BVHs, elementary tests detecting the first ToC and the corresponding intersecting primitives are performed [Pro97]. CCD methods, however, require much more computation time compared to DCD methods and have not been widely used in interactive applications.

To improve the performance of CCD methods, many prior approaches accelerate the performance of CCD by designing specialized algorithms on certain types of models (e.g., rigid objects [RKC02], articulated bodies [ZRLK07], and meshes with fixed topology [GKJ*05, WB05]), developing efficient culling methods [CTM08, TCYM08], and introducing CPU or GPU parallel collision detection methods [GRLM03, KP03, HTG04, SGG*06, LL02, FF04, TMT09]. Prior methods supporting self-collision detections and the general polygonal models may take hundreds of milliseconds and even a few seconds on performing CCD for deforming models consisting of hundreds of thousand triangles and may not be suitable for interactive applications.

Recently, instead of continuing to increase the clock frequency of a single core, the number of cores on a single chip has continued to increase [Bor07]. Current commodity CPUs have up to four or eight cores and current GPUs have more than hundreds of cores [NVI08]. With the number of cores expected to continue to increase, designing algorithms that can properly exploit the multi-core architectures will be critical to achieve overall performance improvement.

Main contributions: We present a novel hybrid parallel continuous collision detection (HPCCD) method utilizing both CPUs and GPUs to achieve the interactive performance of CCD between deforming models consisting of tens or hundreds of thousands of triangles. Our HPCCD method supports various kinds of deforming models and self-collision detection. Our method uses BVHs of deforming models and selectively performs a lazy BV reconstruction method to improve the performance of CCD. Since CPUs are capable of complex branch predictions and efficiently support irregular memory accesses, we use CPUs to perform the BVH traversal and culling. In order to design a highly scalable CPU-based hierarchy traversal and culling, we propose a novel task decomposition that leads a lock-free parallel algorithm in the main loop of our collision detection method, although we use locks in non-critical parts (Sec. 4). Since GPUs are highly optimized for the regular streaming

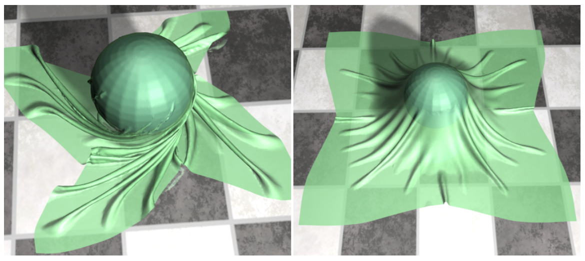

Figure 1: *These images show two frames of our cloth simulation benchmark consisting of 92 K triangles. In this benchmark, our method spends 23 ms for CCD including self-collisions on average and achieves 10.4 times performance improvement by using four CPU-cores and two GPUs over a serial CPU-based CCD method.*

floating-point operations, we use GPUs to execute the elementary tests of the CCD that reduce to solving cubic equations (Sec. 5).

In order to test our method, we apply our method to various benchmarks consisting of tens or hundreds of thousands of triangles (Sec. 6). In the tested benchmarks, our method improves the performance of CCD by more than an order of magnitude using four CPU-cores and two GPUs compared with using a single CPU-core. This performance improvement results in the interactive performance, up to 148 fps, for CCD in our benchmarks. This performance improvement is caused by reducing dependencies among parallel computational tasks and exploiting both CPUs and GPUs. We conclude and present future directions in Sec. 7 .

2. Related Work

The problem of collision detection (CD) has been well studied and excellent surveys are available [LM03, Eri04, TKH*05]. In this section, we review previous work related directly to our method.

2.1. Continuous Collision Detection (CCD)

There are many different CCD approaches and some of them include algebraic methods [Pro97, KR03], adaptive bisection [SSL02], etc. CCD methods have been further optimized for rigid models [RKC02] and articulated models [ZRLK07]. CCD methods for deformable polygonal meshes were initially designed for meshes with fixed connectivity [GKJ*05, WB05] and, recently, have been extended to models with topology changes [CTM08, TCYM08]. Also, a few culling techniques have been proposed to remove redundant elementary tests for CCD [TCYM08, CTM08].

2.2. Parallel Collision Detection

There have been considerable efforts to perform collision detection efficiently using GPUs [HTG04, KP03, GRLM03]. Govindaraju et al. [GRLM03] proposed an approach for fast CD between complex models using GPU-accelerated visibility queries. There have been GPU-based algorithms for self-collision [VSC01, GKJ*05] specialized for certain types of input models (e.g., closed objects). Sud et al. [SGG*06] proposed a unified GPU-framework for various proximity queries including CCD between various deforming models.

A few CPU-based parallel CD methods also have been proposed. Lawlor and Laxmikant [LL02] proposed a voxel-based CD method for static models and achieved up to 60% parallel efficiency by using distributed-memory parallel machines and applying a generic load-balancing method. Figueiredo and Fernando [FF04] designed a parallel CD algorithm for a virtual prototype environment. This method achieved its best performance, 100% improvement, by using four CPU-cores over using a single-core and then showed lower performance as more CPUs were added. These parallel methods supported only DCD for static models. Tang et al. [TMT09] proposed a front-based task decomposition method that utilizes multi-core processors for collision detection between deformable models. Their CPU-based parallel method achieves a high scalability by even using 16 CPU-cores. We will compare our method with this CPU-based parallel method in Sec. 6.3.

2.3. Lock-Free Parallel Algorithms

The traditional synchronization based on locks can degrade the performance of parallel algorithms because of lock contention [CS99]. To address this problem, there have been many efforts to reduce or eliminate the use of locks by designing lock-free algorithms relying on atomic swap instructions [Her03]. However, these lock-free algorithms are based on the assumption that actual lock contentions are rare and thus reducing conflicting accesses to shared data structure is crucial. On the other hand, our HPCCD method eliminates conflicting accesses to shared BVH data in the main loop of CD part based on our novel task decomposition method, although locks are only used in non-critical parts.

3. Overview

In this section, we give a background on CCD and an overview of our hybrid parallel method.

3.1. Background on CCD

In order to find intersecting primitives at the first ToC during a time interval between two discrete time steps, CCD methods model continuous motions of primitives by linearly interpolating positions of primitives between two discrete time steps. We also use this linear continuous motion. There are two types of contacts: inter-collisions between two different models and intra-collisions, i.e., self-collisions, within a model. Both contacts arise in two contact configurations, vertex-face (VF) case and edge-edge (EE) cases. These two cases are detected by performing VF and EE elementary tests, which reduce to solving cubic equations given the linear continuous motion between two discrete time steps [Pro97].

BVHs are widely used to accelerate the performance of

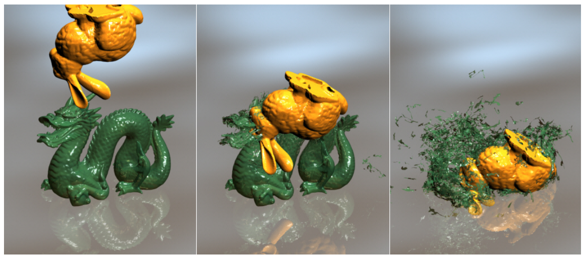

Figure 2: *These images are from the breaking dragon benchmark consisting of 252 K triangles, the most challenging benchmark in our test sets. Our method spends 54 ms for CCD including self-collisions and achieves 12.5 times improvement over a serial CPU-based method.*

CCD methods. Some of the commonly used types of bounding volumes (BVs) include spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), etc [TKH*05, LM03]. We use the AABB representation because of its fast update method, simplicity, and wide acceptance in various collision detection methods [TKH*05]. Given a BV node n of a BVH, we use notations of $L(n)$ and $R(n)$ to indicate the left child and right child nodes of the node n. As models are deforming, we have to update BVHs of such deforming models. We update BVHs based on a selective restructuring method, which reconstructs small portions of BVHs that may have poor culling efficiency and refits the rest of portions of BVHs by traversing the BVH in a bottom-up manner [YCM07, TKH*05]. We also combine the selective restructuring method with a lazy BV construction method.

3.2. Common BVH-based Collision Detection

For BVHs of deforming models, we merge these BVHs into a BVH and then perform our CCD method with the merged BVH. In this case, inter-collisions among multiple objects and self-collisions within each object can be computed by performing self-collision detection with the merged BVH [TKH*05]. We initially perform collision detection between two child nodes of the root node of the BVH. To do that, we create a *collision test pair* consisting of these two nodes. Then, we push the pair into a queue, called *collision test pair queue*. In the main loop of the CD algorithm, we dequeue a pair (n,m) consisting of BV nodes n and m from the queue and perform a BV overlap test between two BV nodes, n and m, of the pair. If there is an overlap, we refine two BV nodes with their two child BV nodes and create four different collision pairs, $(L(n),L(m))$, $(L(n),R(m))$, $(R(n),L(m))$, and $(R(n),R(m))$. If we have to find self-collisions within nodes n and m for dynamically deforming models, we also create two additional collision pairs, $(L(n),R(n))$ and $(L(m),R(m))$. When we reach leaf nodes during the BVH traversal, we perform the VF and EE elementary tests between features (e.g., vertex, edges, and faces) associated with the leaf nodes. We continue this process until there is no more collision pairs in the queue.

Issues of parallelizing the BVH-based CD: Parallelizing the BVH-based CD is rather straightforward. One naive approach is to divide the pairs stored in the collision test pair queue into available threads. Then, each thread performs the BVH traversal and adds collision test pairs into its own queue without any locking. However, threads have to use a locking mechanism for reconstructing a BV of a node. We found that this naive method shows poor scalability (Fig. 7). Two issues cause this low performance. The first one is that contacts among objects occur in localized regions of objects and processing a pair may generates a high number of additional pairs or may terminate soon after. This high variance of the computational workload associated with each pair requires frequent redistributions of computational workload for a load-balancing among threads and results in a high overhead. The second one is that using locks to avoid simultaneous reconstructions on the same node serializes these multiple threads and hinders the maximum utilization of multi-core architectures. In this paper, we propose a scalable parallel CD method that addresses these issues.

3.3. Overview of Our Approach

At a high level, our HPCCD method consists of two parts (see Fig. 3): 1) CPU-based BVH update and traversal with lazy BV reconstructions and 2) GPU-based VF and EE elementary tests.

Our HPCCD method first updates a BVH of a deforming model by refitting the BVs. Then, we perform the BVH traversal and culling by using multiple CPU threads. During the BVH traversal, we also perform selective BV reconstruction method in a lazy manner. In order to design a highly scalable algorithm, we decompose the BVH traversal into *inter-CD task units*, which enable a lock-free parallel algorithm in the main loop of our collision detection method. These inter-CD task units are guaranteed to access different sets of nodes and do not require any locking mechanism for lazy BV reconstructions on BV nodes. We also propose a simple dynamic task reassignment method for high load-balancing among threads by partitioning inter-CD task units of a thread to other threads. When reaching leaf nodes during the BVH traversal, we send potentially intersecting triangles contained in the two leaf nodes to GPUs and perform elementary tests constructed from the triangles using GPUs. In order to minimize the time spent on sending data, we asynchronously send the mesh information to GPUs during the BVH update and traversal. We only send colliding primitives and their contact information (e.g, a contact point and normal) at the first ToC to CPUs after finishing the tests.

4. Inter-CD based Parallel CCD

In this section, we explain our novel decomposition and task reassignment methods for the CPU-based hierarchy traversal and culling of the HPCCD method.

We define a few terminologies to describe our method. We define an *inter-collision test pair set*, $ICT_{PS}(n)$, to denote all the collision test pairs generated in order to find inter-collisions between two child nodes of a node n. We define

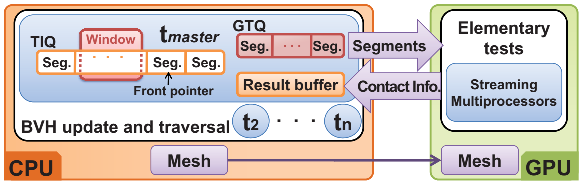

Figure 3: *This figure shows the overall structure of the HPCCD method. It performs the BVH update and traversal at CPUs and the elementary tests at GPUs.*

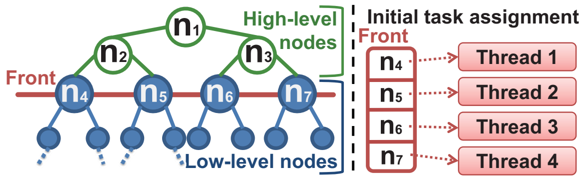

Figure 4: *This figure shows high-level and low-level nodes given four available threads. The right portion of the figure shows an initial task assignment for the four threads.*

two nodes to have a *parent-child relationship* if one node is in the sub-tree rooted at the other.

4.1. Inter-CD based Decomposition

Our task decomposition method for the parallel CPU-based BVH traversal is based on task units of an inter-collision detection, *inter-CD*. Each inter-CD task unit processes collision test pairs represented by $ICT_{PS}(n)$ of a node n. To assign task units to each CPU thread, we push a node n into a *task unit queue* and associate the queue with the thread. Inter-CD task unit has two phases: 1) setup phase and 2) the BVH traversal phase performing BV overlap tests. In the setup phase of an inter-CD task, we first fetch a node, n_S, from its task unit queue and refine the node into its two child nodes $L(n_S)$ and $R(n_S)$. If we have to perform the self-collision detection, we push those two child nodes that will generate other inter-CD task units into the task unit queue. We also create a *scheduling queue* for dynamic task reassignment for a load-balancing, which will be explained in Sec. 4.3.

After the setup phase, we perform the BVH traversal phase. During the BVH traversal phase, we use a collision test pair queue as used in the common BVH-based traversal explained in Sec. 3.2. We assign a collision test pair $(L(n_S), R(n_S))$ into the collision test pair queue. Then, we fetch a pair consisting of two nodes n and m from the collision test pair queue and perform a BV overlap test between two BV nodes n and m of the pair. If there is a BV overlap, we refine both of those two nodes into $L(n)$, $R(n)$, $L(m)$, and $R(m)$. Then, we construct four collision test pairs, $(L(n),L(m))$, $(L(n),R(m))$, $(R(n),L(m))$, and $(R(n),R(m))$, and push them in the collision test pair queue. We continue this process until we reach leaf nodes. If we reach leaf nodes, we perform exact VF and EE elementary tests between features associated with the leaf nodes by using GPUs. If there is any collision, we put the collision result into a *result buffer*.

Disjoint property of inter-CD task units: During processing an inter-CD task unit of a node n, we create and test various collision test pairs, $ICT_{PS}(n)$, of nodes that are in the sub-tree rooted at the node n. If there is no parent-child relationship between two nodes, say n and m, we can easily show that a set of accessed nodes during performing $ICT_{PS}(n)$ is disjoint from another set of accessed nodes during performing $ICT_{PS}(m)$. We will utilize this disjoint property to design an efficient parallel CCD algorithm (Sec. 4.2 and Sec. 4.3).

Serial CCD method: Before we explain our HPCCD method, we first explain how to perform CCD with a single thread based on inter-CD task units. Given a BVH whose root node is n_R, we perform an inter-CD task unit, $ICT_{PS}(n_R)$, of the node n_R. At the end of processing the task unit of $ICT_{PS}(n_R)$, the collision test pair queue is empty. However, the task unit queue may have two nodes, which are two child nodes of n_R. We fetch a node, n, from the task unit queue and perform $ICT_{PS}(n)$. We continue this process until there is no node in the task unit queue. Then, the result queue contains all the self- and intra-collisions among the original deforming models. An important property in this serial CCD method is that any pair of nodes in the task unit queue do not have the parent-child relationship.

Note that our serial CCD algorithm based on inter-CD task units is constructed by simply reordering the processing order of collision test pairs of typical BVH-based CD methods explained in Sec. 3.2.

4.2. Initial Task Assignment

Each thread is initialized with a node n. If nodes assigned to threads satisfy the disjoint property, we do not need to use expensive locking mechanisms to prevent multiple threads from attempting to reconstruct the same BV node for a lazy BV reconstruction during the BVH traversal.

To guarantee that nodes assigned to threads do not have any parent-child relationship and thus satisfy the disjoint property, we compute such nodes by maintaining a front while we traverse the BVH from its root node in the breadth-first order (see Fig. 4). If the size of the front is same as the number of available threads, we stop the BVH traversal and assign each node in the front to each thread. We refer to those nodes and all the nodes in the sub-trees rooted at those nodes as *low-level nodes* while all the other nodes are referred to as *high-level nodes*. An example of low-level and high-level nodes for four threads is shown in Fig. 4.

In the same manner with the serial CCD method described in Sec. 4.1, each thread finds collisions that occur in a subtree of the node that is initially assigned to the thread. Once each thread finishes its computation, we process inter-CD task units of high-level nodes. First, we process parent nodes, n_2 and n_3 in the case of Fig. 4, of initially assigned nodes to each thread. We wait until the processing of inter-CD task units of two nodes n_2 and n_3 finishes and then process their parent node, n_1. While processing high-level nodes, we do not add child nodes of those nodes to the task unit queue since we already processed inter-CD tasks of those child nodes.

In this simple task assignment method, we can traverse the BVH, perform the overlap tests, and invoke lazy BV reconstructions, if necessary, without any locking. However, a thread can finish its assigned task units much earlier than other threads because of the localized contacts among objects. In this case, it is desirable to divide task units of a thread to the thread finishing its task to fully utilize all the available p threads. For this, we propose a dynamic task reassignment in the next section.

4.3. Dynamic Task Reassignment

Suppose that a thread finishes its computation and there is no more nodes left in the task unit queue. We will refer to this thread as a *requesting thread* $t_{request}$. In order to utilize this requesting thread $t_{request}$, we detect another thread t_{dist} called a *distribution thread* that can give its computation workload to the requesting thread $t_{request}$. At a high level, we choose to distribute an inter-CD task unit that may have the highest computation workload to the requesting thread. More specifically, we choose a distribution thread t_{dist} with the highest number of triangles associated with the front node in its task unit queue among threads. Then, the distribution thread t_{dist} gives the front node in its task unit queue to the requesting thread $t_{request}$.

The main rationale of this approach is as follows. Nodes in the task unit queue represent inter-CD task units and can be distributed to other threads, since there are no parent-child relationships among these nodes. The front node in the task unit queue in each thread is likely to cause the highest computational workload among nodes in the queue given the breadth-first order traversal of the BVH. Moreover, we expect that there will be more computational workload of processing an inter-CD task unit of a node as the number of triangles associated with the sub-tree rooted at the node is higher. We found that this simple dynamic task reassignment method works quite well in the tested benchmark and achieves up to a 7 times performance improvement by using 8 CPU-cores compared to using a single CPU-core.

Suppose that a requesting thread $t_{request}$ chooses a distribution thread t_{dist}. To request the distribution of computation workloads, $t_{request}$ places a request to the scheduling queue of the thread t_{dist}. To place the request, a locking to the scheduling queue is required since other threads may attempt to access the same scheduling queue to place their requests. After placing the request, $t_{request}$ sleeps.

In each thread, we check whether its own scheduling queue is empty or not by looking at its size right after finishing all the collision test pairs and before performing another inter-CD task unit. If there are no requests in the queue, the thread continues to process another inter-CD task unit by fetching a node from its task unit queue. If there are requests in the scheduling queue, we distribute its computational workload stored in the task unit queue to the requesting threads. After distributing the computational workload, the distribution thread t_{dist} sends wake-up messages with the partitioned nodes to the requesting threads. Once a requesting thread $t_{request}$ receives the wake-up message, the thread pushes the received node into its task unit queue and resumes its computation by performing inter-CD task units. Pseudo-code of our parallel CCD algorithm based on inter-CD task units is shown in Listing 1. Note that we do not perform any synchronization nor locking in the main collision detection loop.

```
Perform_Self_CD (node n) {
  TaskUnit_Q <- n;

  while (! TaskUint_Q.Empty ()) {
    n <- TaskUint_Q.Dequeue ();

    if (n has child nodes) {
      TaskUint_Q <- L(n) and R(n);
      Pair_Q <- (L(n),R(n));
    }

    while (! Pair_Q.Empty ()) { // Main CD loop
      Pair <- Pair_Q.Dequeue () ;
      Perform lazy reconstruction for nodes of
       Pair;

      if (IsOverlap (Pair)){
        if (IsLeaf (Pair) )
          Perform elementary tests;
        else
          Pair_Q <- Refine (Pair);
      }
    }

    if (! SchedulingQ.Empty ())
      Distribute its work to the requesting
       thread;
  }

  Request to a distribution thread and sleep;
}
```

Listing 1: ***Pseudocode of HPCCD method***

4.4. Parallelizing an Inter-CD Task Unit

During processing high-level nodes, the number of inter-CD task units that can run in parallel is smaller than the number of available threads. In order to fully utilize all the available CPU threads, we propose an algorithm that performs an inter-CD task unit in a parallel manner.

Our method is based on a simple observation: we do not perform many lazy BV reconstructions while processing high-level nodes since we already traversed many nodes and performed lazy reconstructions during processing inter-CD task units of low-level nodes. Therefore, we choose to use a locking mechanism for lazy BV reconstructions. Since reconstructions of BVs happen rarely during processing high-level nodes, there is a very low chance of a thread waiting for a locked node. By using the locking mechanism, we can arbitrarily partition the pairs of the collision test pair queue into k

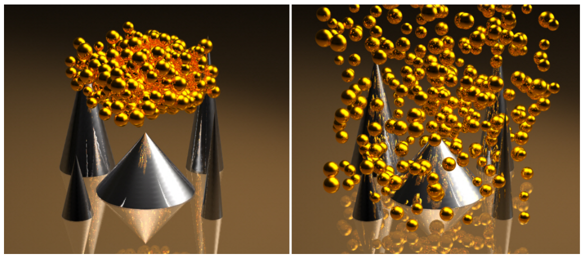

Figure 5: *This figure shows two frames during the N-body simulation benchmark with two different model complexities: 34 K and 146 K triangles. Our method spends 6.8 ms and 54 ms on average and achieves 11.4 times and 13.6 times performance improvements for two different model complexities.*

available threads. For partitioning, we sequentially dequeue and assign a pair into k threads in a round robin fashion. We choose this approach since collision test pairs located closely in the queue may have similar geometric configurations and thus have similar computation workload during processing collision test pairs.

5. GPU-based Elementary Tests

Once we reach leaf nodes of the BVH, we perform the VF and EE elementary tests. To perform VF and EE elementary tests between two potentially colliding primitives at GPUs, we need to send necessary information to the video memory of GPUs. Since sending data from main memory of CPUs to the video memory of GPUs can take high latency, we send the mesh information to GPUs during the BVH update and traversal asynchronously in order to hide the latency of sending the data. Then, when we reach leaf nodes of BVHs, we send two triangle indices of two potentially intersecting triangles. At the GPUs, we construct the VF and EE tests from two triangles referred by the two indices and solve cubic equations to compute the contact information at the first ToC.

5.1. Communication between CPUs and CPUs

In our HPCCD framework, multiple CPU threads generate elementary tests simultaneously. For sending data from CPUs to GPUs, an easy solution would be to let each CPU working thread send two triangle indices to GPUs. However, we found that this approach requires a high overhead since GPUs have to maintain individual device contexts for each CPU thread [NVI08]. Instead, we use a master CPU-GPU communication thread, t_{master} (see Fig. 3). Each CPU thread requests the master thread to send the data to GPUs. The overall communication interface between the CPUs and GPUs is shown in Fig. 3.

The master thread maintains a *triangle index queue* (TIQ). The TIQ consists of multiple (e.g., 128) segments, each of which can contain thousands (e.g., 2K) of a pair of two triangle indices. Each segment can have three different states: "empty", "full", and "partial" states. If all the elements of a segment are empty or filled, the segment has the state of "empty" or "full" respectively. Otherwise, it has the "partial" state. The TIQ has a window that looks at c consecutive segments to see whether they are full and ready to be transfered from main memory to the video memory, where c is set to be the one fourth of the maximum size of the TIQ. Also, the TIQ has a front pointer that indicates a next available empty segment. Initially, the master thread gives two empty segments to each CPU thread. Once a CPU thread requests a new empty segment from the master thread, the master thread gives the empty segment referred by the front pointer and updates the position of the front pointer by finding an empty segment sequentially in the TIQ. The master thread also maintains a *GPU task queue* (GTQ) that holds elements, each of which contains segments that has been sent to the GPUs, a state variable indicating whether the GPU finishes processing the elementary tests of the sent segments, and an output buffer that can contain the contact information at the first ToC.

As each CPU working thread performs the BVH traversal, it adds two triangle indices to one segment from the two assigned segments. When the segment does not have additional space to hold any more triangle indices, the thread sets the state of the segment to be "full". Then, the thread asks a new segment from the master thread. Meanwhile, the thread does not wait for the master thread and asynchronously continues to perform the BVH traversal with the other segment. In this way we can hide the waiting time for a new empty segment that has been requested to the master thread.

Procedure of the master thread: The master CPU-GPU communication thread performs the following steps in its main loop. The first step is to look at the consecutive segments in the TIQ's window. If there are multiple consecutive "full" segments, we send these consecutive "full" segments to GPUs with one data sending API call and push them in one element of the GTQ. The reason why we send consecutive segments is to reduce the number of calls of data sending API, which has a high kernel call overhead [NVI08]. We then update the window position by sliding it right after the transmitted consecutive segments in the TIQ. Note that any "partial" segments are not sent and will be checked again for the transmission when the window contains these segments later. When a GPU finishes processing all the elementary tests constructed from the segments received, the GPU sets the state variable of the element of the GTQ to be "data ready" and stores all the colliding primitives in the output buffer of the queue element. As a next step, the master thread goes over all the elements in the GTQ and remove elements that have the "data ready" state. We also update the result buffer that contains the contact information at the first ToC with output buffers of these removed elements. Then, we make segments of these "data ready" elements to have the "empty" state in order to be reused for a next request of "empty" segments. The final step of the master thread is to process requests of "empty" segments from CPU threads.

Load balancing between CPUs and GPUs: When we use a low-end GPU and high-end CPUs, we found that the GPU may not process all the elementary tests generated from these CPU-cores, thus requiring additional GPUs to load-balance and achieve a further performance improvement. Without having additional GPUs, we can load-balance the elementary tests across both the CPUs and GPUs to achieve an additional performance improvement by using CPUs to process the elementary tests instead of generating and sending the elementary tests to the GPUs. To do this, a CPU working thread checks whether the GTQ's size is bigger than the number of GPU cores, right before the CPU thread finishes to fill a segment and requests an "empty" segment. If the GTQ's size is bigger than the number of GPU cores, we assume that the GPUs are busy and cannot process all the segments produced by CPUs. In this case, the CPU thread processes the half of the elementary tests of the "full" segment and continues to performs the BVH traversal until it fills the half-empty segment. Once the segment becomes full again, the CPU thread checks the queue size of GTQ and performs the same procedure again.

6. Implementation and Results

We have implemented our HPCCD method and tested it with two different machines. The first one is an Intel Xeon desktop machine with two 2.83 GHz quad-core CPUs and a GeForce GTX285 GPU. The second one is an Intel i7 desktop machine with one 3.2 GHz quad-core CPU and two Geforce GTX285 GPUs. The Intel i7 processor supports the hyper-threading [CS99]. We will call these two machines 8C1G (8 CPU-cores and 1 GPU) and 4C2G (4 CPU-cores and 2 GPUs) machines for the rest of the paper. We use the *OpenMP* library [DM98] for the CPU-based BVH traversal and CUDA [NVI08] for the GPU-based elementary tests. We also use a feature-based BVH using *Representative-Triangles* [CTM08] in order to avoid any redundant VF and EE elementary tests.

Parallel BVH update: Before we perform the CCD using a BVH, we first refit BVs of the BVH. Our algorithm traverses the BVH in a bottom-up manner and refits the BVs. To design a parallel BVH refitting method utilizing k threads, we compute $2k$ nodes in a front as we did for the initial task assignment of inter-CD task units to threads in Sec 4.2. Then, we assign the first k nodes stored in the front to each thread and then each thread performs the BV refitting to the subtree rooted at the node. Since the BVH is unlikely to be balanced, a thread can finish its BV refitting earlier than other threads. For the thread finishing its refitting, we assign the next available node in the front to the thread. During the BVH traversal, we identify and selectively restructure BVs with low culling efficiency. To do that, we use a heuristic metric proposed by Larsson and Akenine-Möller [LAM06]. We perform a lazy BV reconstruction by using a simple median-based partitioning of triangles associated with the node.

Benchmarks: We test our method with three types of dynamic scenes (see Table 1). The first benchmark is a cloth simulation, where a cloth drapes on a ball and then the ball

Model	Tri. (K)	Image	Avg. CCD time (ms)
Cloth simulation	92	Fig. 1	23.2
Breaking dragon	252	Fig. 2	53.6
LR N-body simulation	32	Fig. 5	6.8
HR N-body simulation	146	Fig. 5	53.8

Table 1: ***Dynamic Benchmark Models***

is spinning (Fig. 1). This benchmark consists of 92 K triangles and undergoes severe non-rigid deformations. In our second benchmark, a bunny collides with a dragon model. Then, the dragon model breaks into numerous pieces (Fig. 2). This model has 252 K triangles. Our final benchmark is a N-body simulation consisting of multiple moving objects (Fig. 5). We compute two different versions with different model complexities of this benchmark: a high-resolution (HR) version has 146 K triangles and a low-resolution (LR) has 32 K triangles. Each object in this benchmark may undergo a rigid or deformable motion and objects collide with each other and the ground. These models have different model complexities and characteristics. As a result, they are well suited for testing the performance of our algorithm.

6.1. Results

We measure the time spent on performing our HPCCD including self-collision detection with two different machines. We achieve the best performance with the 4C2G machine, the four CPU-cores machine with two GPUs.

In the 4C2G machine, the HPCCD method spends 23.2 milliseconds (ms), 53.6 ms, 6.8 ms, and 53.8 ms on average for the cloth simulation, the breaking dragon, and LR/HR N-body simulations respectively; we will report results in this benchmark order for the rest of the tests. These computation times translate to about 43, 19, 148 and 19 frame per seconds (fps) for the four benchmarks respectively. Compared to using a single CPU-core, we achieve 10.4, 12.5, 11.4, and 13.6 times performance improvements. We also show the performance of HPCCD with different numbers of CPU threads and GPUs with the two different machines (see Fig. 6).

We measure the scalability of our CPU-based BVH update and traversals of our HPCCD method as a function of CPU threads (e.g., 1, 2, 4, and 8 threads) without using GPUs in the 8C1G machine (see Fig. 7) with all the benchmarks. The CPU part of our HPCCD method shows 6.5, 6.5, 6.4, and 7.1 times performance improvements by using 8 CPU-cores over a single CPU-core version in the four benchmarks respectively. We achieve a stable and high scalability near the ideal linear speedup across our benchmarks that have different model complexities and characteristics. This high scalability is due to the lock-free parallel algorithm used in the main loop of the collision detection.

6.2. Analysis

Our HPCCD method consists of four components; 1) BV refitting, 2) parallel CCD with low-level and high-level nodes, 3) performing elementary tests, and 4) other serial compo-

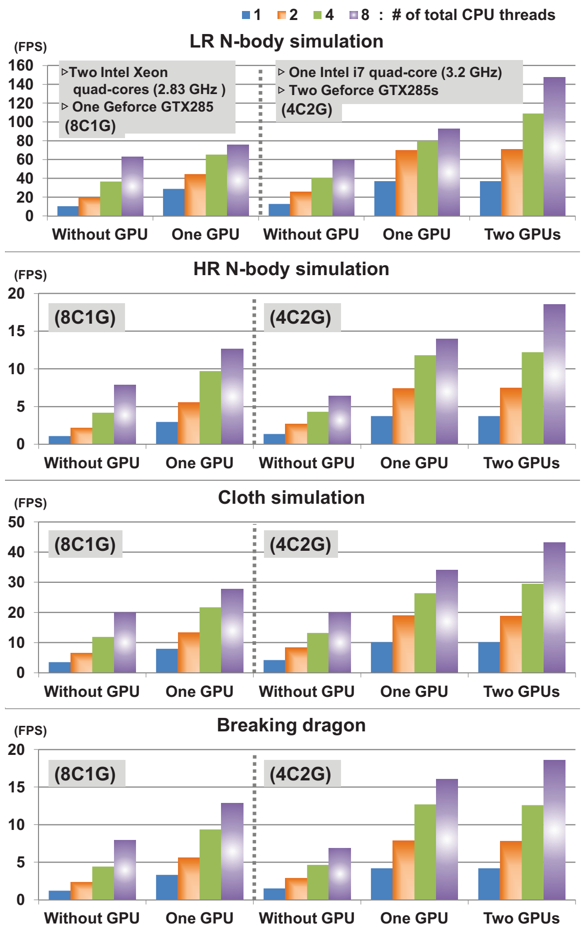

Figure 6: *We measure the performance of our HPCCD with four different benchmarks in two machines as we vary the numbers of CPU threads and GPUs. We achieve 10 times to 13 times performance improvements by using the quad-core CPU machine with two GPUs.*

nents and miscellaneous parts (e.g., setup of threads). In the serial version of the HPCCD using only one CPU-core, these four components take about 3%, 19%, 77%, and 1% on average in our tested benchmarks respectively; the results will be reported in the order each component appeared above for the rest of the section. By offloading the elementary tests to GPUs and parallelizing other components except for the serial part, we were able to achieve up to a 13.6 times performance improvement. By using the 4C2G machine, the four components take about 12%, 81%, not-available, and 6.46% respectively. Since the time spent on performing elementary tests in GPUs are overlapped with the CPU-based BVH traversal, we could not exactly measure that component. However, that component takes less than or equal to that of the CPU-based BVH traversal.

Scalability of different components: We measure the scalability of the first and the second components in the 8C1G machine without using the GPU. We achieve 3.0 and 6.5 performance improvements by using 8 CPU-cores over a single CPU-core version for the first and second components respectively. The low scalability of the first BV refitting component is caused by its small workloads of the BV refitting operations. Therefore, the overhead of our simple load balancing methods and frequent needs for load balancing lower the scalability. We also measure the scalability of the combination of the second and the third components; it shows 7.5 times performance improvement by using 8 CPU cores. Since the portions of the second and the third components are dominant in the CCD, we achieve a high scalability in the whole CCD process despite the low performance improvement of the BV refitting component.

We also measure the scalability of the third component of performing elementary tests as a function of the number of GPUs used in the 4C2G machine. We perform the CCD by using only a single CPU-core and measure the time, T_e, spent on performing elementary tests after serializing the components of the HPCCD. We then measure how much the overall CCD time is reduced from using a single CPU-core to using GPUs for the elementary tests. We refer to the reduced time as T_r. The scalability of the third component can be computed by a simple equation, $T_e/(T_e - T_r)$. According to this equation, we achieve 2.8 times and 4.6 times performance improvements for the third component by using one GPU and two GPUs respectively on average in the benchmarks. However, we expect to achieve a higher scalability when using multiple CPUs, since we can generate more elementary tests efficiently and thus utilize GPUs better.

Segment size in the TIQ: The size of a segment in the TIQ affects the performance of the HPCCD, since it determines the granularity of the communications from CPUs to GPUs and between the master and slave threads. A small segment size may lead to a high communication overhead between the master and slave threads. On the other hand, a large segment size may cause GPUs to be idle at the beginning of the BVH traversal, since GPUs should wait for "full" segments from CPUs. We found that 2K entries for a segment show the best performance for the tested benchmarks in the tested two machines. However, bigger entries (e.g., 4K to 16K entries) show only minor (e.g., 2 %) performance degradation.

Limitation: Our algorithm has certain limitations. Note that the serial part of the HPCCD method takes 6.46% with the 4C2G machine. According to the Amdahl's law [CS99], we can achieve only 15 times more improvement in addition to the 13 times performance improvement we have achieved by using the 4C2G machine, although we would use unlimited resource of CPUs and GPUs. Also, our method can detect a case when GPUs do not keep up with CPUs and use CPUs to perform elementary tests to achieve a higher performance. However, our current algorithm does not attempt to achieve a higher performance when GPUs are idle. We can implement processing inter-CD task units in GPUs using the CUDA and perform the inter-CD tasks in GPUs when GPUs are idle. However, it requires further research to map the hierarchical traversal well in the streaming GPU architectures. Also, we mainly focus on the efficient handling of large deforming

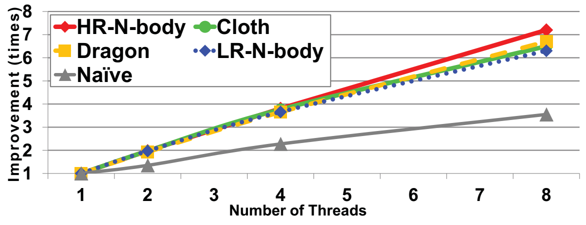

Figure 7: *This figure shows the performance improvement of our HPCCD method as a function of the number of CPU-cores without using GPUs over using a single CPU-core. The gray line, marked by triangles, shows an average performance improvement of the naive approach described in Sec. 3.2 with all the tested benchmarks.*

models with consisting of tens or hundreds of thousands of triangles. If the model complexity is small or we have to handles models consisting of a small number of rigid bodies, our method may not get a high scalability since there are not many inter-CD task units that we can parallelize.

6.3. Comparisons

It is very hard to directly compare the performance of our method over prior methods. However, most prior approaches use either GPUs [GRLM03, KP03, HTG04, SGG*06] or CPUs [LL02, FF04, TMT09] to accelerate the performance of CCD. One distinct feature of our method over prior methods is that it maps CPUs for the BVH traversal and GPUs for performing elementary tests. Since these two different components, the traversal and elementary tests, are more suitable to CPUs and GPUs respectively, we decompose the computation of CCD in such a way that it can fully exploit the multi-core architectures of CPUs and GPUs. Therefore, our method achieved more than an order of magnitude performance improvement over using a single CPU-core and showed interactive performance for large-scale deforming models.

We compare our method with the current state of the art technique proposed by Sud et al. [SGG*06]. This method supports the general polygonal models and CCD including self-collision. We contacted authors of this technique, but we were not able to get the binary of this method. Therefore, we compare results of our method with their results reported in their paper. Note that this comparison is rather unfair, since they used a GeForce 7800, 3-years old graphics card. Since their tested benchmarks are different from ours, we measure an average ratio of model complexities of tested benchmarks to the CCD times spent for processing those benchmarks. The ratio of our method is 107 times higher than that of their method. This means that our method can process 107 times bigger model complexity given a unit time or run 107 times faster given a unit model complexity than their method. Also, according to the GPU performance growth data from the GeFroce 7800 to GeForce GTX 280 for the last 3 years [NVI08], the performance has been improved about 6 times. Therefore, based on this information, we conjecture that our method is about 5 times to 10 times faster than their method even though they would use two GeForce GTX 285 GPUs that our method was tested. Moreover, they reported that the performance of their method is limited by the data read-back performance. Although the performance of their method would have been improved by using a recent GPU, data read-backs has not been much improved in past years (e.g., about 3 times improvement in terms of data bandwidth for three years in the past [NVI08]). They reported that the data read-back from a GPU and other constant costs even for small models span between 50 ms and 60 ms at least, which is even higher than or comparable to the whole computation time of our HPCCD method tested with large-scale deforming models.

We also compare our method with a CPU-based parallel CCD method proposed by Tang et al. [TMT09]. This method also achieved a high performance improvement by using 16 CPU-cores. However, our method achieves about 50% and 80% higher performance with the same tested benchmarks: the cloth simulation and LR N-body simulation respectively. Since the portion of elementary tests is larger in the LR N-body simulation, our hybrid method achieves a higher performance improvement with the LR N-body simulation. Also, according to the Google Product Search [†] and its reported lowest prices for CPUs and GPUs, the price of the 16 CPU-cores (USD 7200) used in [TMT09] is 4.4 times higher than that of the CPU (USD 995) and two GPUs (USD 640) of our 4C2G machine. Therefore, our method achieves about 7 times higher performance per unit cost.

7. Conclusion and Future Work

We have presented a novel, hybrid parallel continuous collision detection method utilizing the multi-core CPU and GPU architectures. We use CPUs to perform the BVH traversal and culling since CPUs are capable of complex branch predictions and efficiently support irregular memory accesses. Then, we use GPUs to perform elementary tests that reduce to solving cubic equations, which are suitable for the streaming GPU architecture. By taking advantage of both of CPUs and GPUs, our method achieved more than an order of magnitude performance improvement by using a four CPU-core and two GPUs over using a single CPU-core. This resulted in an interactive performance for CCD including self-collision detection among various deforming models consisting of tens or even hundreds of thousand triangles.

There are many avenues for future work. In addition to addressing our current limitations, we would like to extend our current HPCCD method to exploit the Larrabee architecture [SCS*08], which provides the 16-wide SIMD functionality. We would also like to test the scalability of our method with more CPUs and GPUs. Finally, we would like to design parallel algorithms for other proximity queries including minimum separation distance and penetration depth queries.

[†] `http://www.google.com/products`

Acknowledgment

We would like to thank anonymous reviewers for their constructive feedbacks. We also thank Min Tang, Dinesh Manocha, Joon-Kyung Seong, Young J. Kim, Samuel Brice, and members of SGLab. for their supports and code sharing. The tested models are courtesy of the UNC dynamic model benchmarks. This project was supported in part by MKE/MCST/IITA [2008-F-033-02,2008-F-030-02], MCST/KEIT [2006-S-045-1], MKE/IITA u-Learning, MKE digital mask control, MCST/KOCCA-CTR&DP-2009, KRF-2008-313-D00922, and MSRA E-heritage.

References

[Bor07] BORKAR S.: Thousand core chips – a technology perspective. *Design Automation Conference* (2007), 746–749. 1

[CS99] CULLER D., SINGH J. P.: *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999. 2, 7, 8

[CTM08] CURTIS S., TAMSTORF R., MANOCHA D.: Fast Collision Detection for Deformable Models using Representative-Triangles. *Symp. on Interactive 3D Graphics* (2008), 61–69. 1, 2, 7

[DM98] DAGUM L., MENON R.: OpenMP: an industry standard api for shared-memory programming. *IEEE Computational Sci. and Engineering 5* (1998), 46–55. 7

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2004. 2

[FF04] FIGUEIREDO M., FERNANDO T.: An efficient parallel collision detection algorithm for virtual prototype environments. *Parallel and Distributed Systems* (2004), 249–256. 1, 2, 9

[GKJ*05] GOVINDARAJU N., KNOTT D., JAIN N., KABAL I., TAMSTORF R., GAYLE R., LIN M., MANOCHA D.: Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics 24*, 3 (2005), 991–999. 1, 2

[GRLM03] GOVINDARAJU N., REDON S., LIN M., MANOCHA D.: CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *EG. Workshop on Graphics Hardware* (2003), 25–32. 1, 2, 9

[Her03] HERLIHY M.: Obstruction-free synchronization: Double-ended queues as an example. In *Int. Conf. on Distributed Computing Systems* (2003), pp. 522–529. 2

[HTG04] HEIDELBERGER B., TESCHNER M., GROSS M.: Detection of collisions and self-collisions using image-space techniques. *Proc. of Winter School of Computer Graphics 12*, 3 (2004), 145–152. 1, 2, 9

[KP03] KNOTT D., PAI D. K.: CInDeR: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface* (2003), 73–80. 1, 2, 9

[KR03] KIM B., ROSSIGNAC J.: Collision prediction for polyhedra under screw motions. *ACM Symposium on Solid Modeling and Applications* (2003), 4–8. 2

[LAM06] LARSSON T., AKENINE-MÖLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers and Graphics 30*, 3 (2006), 451–460. 7

[LL02] LAWLOR O. S., LAXMIKANT V. K.: A voxel-based parallel collision detection algorithm. *Supercomputing* (2002), 285–293. 1, 2, 9

[LM03] LIN M., MANOCHA D.: Collision and proximity queries. *Handbook of Discrete and Computational Geometry* (2003). 2, 3

[NVI08] NVIDIA: CUDA programming guide 2.0, 2008. 1, 6, 7, 9

[Pro97] PROVOT X.: Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface* (1997), 177–189. 1, 2

[RKC02] REDON S., KHEDDAR A., COQUILLART S.: Fast continuous collision detection between rigid bodies. *Computer Graphics Forum 21*, 3 (2002), 279–287. 1, 2

[SCS*08] SEILER L., CARMAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics 27*, 3 (2008). 9

[SGG*06] SUD A., GOVINDARAJU N., GAYLE R., KABUL I., MANOCHA D.: Fast Proximity Computation among Deformable Models using Discrete Voronoi Diagrams. *ACM SIGGRAPH* (2006), 1144–1153. 1, 2, 9

[SSL02] SCHWARZER F., SAHA M., LATOMBE J.-C.: Exact collision checking of robot paths. *Workshop on Algorithmic Foundations of Robotics* (2002). 2

[TCYM08] TANG M., CURTIS S., YOON S.-E., MANOCHA D.: Interactive continuous collision detection between deformable models using connectivity-based culling. *ACM Symp. on Solid and Physical Modeling* (2008), 25 – 36. 1, 2

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 19*, 1 (2005), 61–81. 2, 3

[TMT09] TANG M., MANOCHA D., TONG R.: Multicore collision detection between deformable models. In *SIAM/ACM Joint Conference on Geometric and Solid & Physical Modeling* (2009). to appear. 1, 2, 9

[VSC01] VASSILEV T., SPANLANG B., CHRYSANTHOU Y.: Fast cloth animation on walking avatars. *Computer Graphics Forum (Eurographics) 20*, 3 (2001), 260–267. 2

[WB05] WONG W. S.-K., BACIU G.: Dynamic interaction between deformable surfaces and nonsmooth objects. *IEEE Trans. on Visualization and Computer Graphics 11*, 3 (2005), 329–340. 1, 2

[YCM07] YOON S., CURTIS S., MANOCHA D.: Ray tracing dynamic scenes using selective restructuring. *Eurographics Symp. on Rendering* (2007), 73–84. 3

[ZRLK07] ZHANG X., REDON S., LEE M., KIM Y. J.: Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling. *ACM Transactions on Graphics 26*, 3 (2007), 15. 1, 2