PCCD: Parallel Continuous Collision Detection

Duksu Kim
KAIST

Jae-Pil Heo
KAIST

Sung-eui Yoon
KAIST

Abstract

We present a novel parallel continuous collision detection (PCCD) method to utilize the widely available multi-core CPU architecture. Our method works with a wide variety of deforming models and supports self-collision detection. Our method uses a feature-based bounding volume hierarchy (BVH) to improve the performance of continuous collision detection. Also, our method selectively performs lazy reconstructions. To design a highly scalable PCCD method, we propose novel task decomposition methods for our BVH-based collision detection and dynamic task assignment methods to obtain a high load-balancing among computation workloads assigned to each thread. Our method achieves up to 7.3 times performance improvement by using 8-cores compared to using a single-core. The high performance improvement is mainly due to a few dependencies and synchronizations among different computation tasks performed in each thread. As a result, our PCCD method is able to achieve an interactive performance, 50 ms – 140 ms, on average, for various deforming benchmarks consisting of hundreds of thousand triangles.

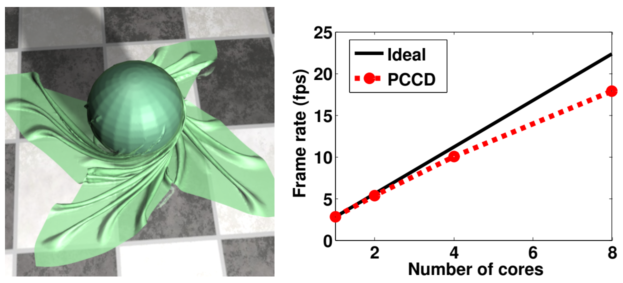

Figure 1: Cloth Benchmark: *The left image shows a frame of our cloth simulation benchmark consisting of* 92 *K triangles. The right image shows the frame rate, frames per second (fps), of our parallel continuous collision detection method with an ideal frame rate assuming a perfect scalability as a function of the number of cores in the CPU architecture. In this benchmark our method spends* 56 *ms for continuous collision detection including self-collisions on average and* 6.4 *times performance improvement by using 8-cores over using a single-core. We use an Intel Xeon machine with two* 2.83 *GHz quad-core CPUs.*

1 Introduction

Collision detection between deforming models is one of fundamental tools of various applications including games, physically-based simulation, CAD/CAM systems, computer animation, and robotics. Collision detection can be classified into two categories: discrete and continuous collision detection methods. Discrete collision detection (DCD) finds intersecting primitives at discrete time steps. DCD can be performed quite efficiently by using bounding volume hierarchies (BVHs) of input models and show interactive performance. Therefore, DCD has been widely used in many interactive applications. However, DCD methods can miss colliding primitives that occur between two discrete time steps. This issue can be quite problematic in physically based simulations (e.g., cloth simulation), CAD/CAM applications, etc. On the other hand, continuous collision detection (CCD) identifies the first time of contacts and intersecting primitives during a time interval between two discrete time steps. Typically, CCD methods model continuous motions of primitives by linearly interpolating positions of primitives between two discrete time steps.

Although CCD methods improve the accuracy of collision detection compared to DCD methods, CCD methods typically require much longer computation time. Therefore, CCD technology has not been actively used in interactive applications such as games. There are many approaches to accelerate the performance of CCD by designing specialized algorithms on certain types of models (e.g., rigid objects [Redon et al. 2002], articulated bodies [Redon et al. 2005; Zhang et al. 2007], and meshes with fixed topology [Govindaraju et al. 2005; Wong and Baciu 2005; Hutter and Fuhrmann 2007] and introducing efficient culling methods [Curtis et al. 2008; Tang et al. 2008]. However, these methods may take hundreds of milliseconds and, even, a few seconds on performing CCD for deforming models consisting of hundreds of thousand triangles.

Recently, the performance of CPUs is improved by increasing the number of cores instead of improving the clock frequency of a single core [Asanovic et al. 2006]. Also, most of commodity hardware already has two or more cores. Moreover, it is expected that hundreds of cores on a chip can be built in a near future according to the technology trend of the transistor integration capacity [Borkar 2007]. Intel has already designed a prototype of a many-core architecture with 80 cores that can have TeraFlop performance [INTEL 2006].

In this paper we address the problem of designing a parallel CCD algorithm that can utilize the performance of multi-core CPU architecture. There have been several parallel collision detection methods by using GPUs and CPUs. Particularly, there are many approaches using GPUs to perform collision detection including CCD [Govindaraju et al. 2003; Knott and Pai 2003; Heidelberger et al. 2004; Sud et al. 2006]. These methods employ image-based techniques to perform collision detection and exploit massive parallelism of GPUs. There are, however, only a few works on parallel collision detection methods using multiple CPUs. GPU based methods may miss collisions due to the discrete image resolution of the image-based methods and may not be able to provide interactive performance for CCD between large and arbitrarily deforming models.

Main results: We present a novel parallel continuous collision detection (PCCD) method to achieve the interactive performance of CCD between deforming models on commodity multi-core CPU architecture. Our PCCD method supports various kinds of deforming models and self-collision detection. Our method is based on a feature-based bounding volume hierarchy (BVH) and selectively performs a lazy BV reconstruction method to improve the performance of CCD. In order to design a highly scalable parallel CCD method, we propose novel task decomposition and dynamic task assignment methods (Sec. 4 and Sec. 5). We also propose a simple parallel BVH update method (Sec. 6). We highlight the performance of our algorithms with various benchmarks and analyze the performance of our PCCD method (Sec. 7).

Overall, our method has the following benefits:

1. **High scalability:** Due to a few dependencies and synchronizations between computational tasks, our method achieves highly scalable performance as we allocate more cores to our PCCD method. For example, we are able to achieve up to 7.3 times performance improvement by using 8-cores over using a single-core.
2. **Interactive performance:** Due to the high scalability of our method combined with benefits of using lazy BV reconstruction and a feature-based BVH, our PCCD method spends 50ms – 140ms on average and, thus, is able to achieve interactive performance for CCD including self-collisions between deforming models consisting of tens or hundreds of thousand triangles.
3. **Generality:** Our PCCD method can handle various types of

deforming models including polygon soups. Also, our task decomposition and dynamic task assignment methods can be applicable directly to other BVH-based proximity queries such as minimum separation distance.

2 Related Work

The problem of collision detection has been well studied in various fields including computer graphics, simulation, computational geometry and robotics. Also, good surveys are available [Lin and Manocha 2003; Ericson 2004; Teschner et al. 2005]. In this section, we review prior work mainly on continuous collision detection and parallel computation for collision detection.

2.1 Continuous Collision Detection (CCD)

CCD algorithms find the first time-of-contacts (ToC) in continuous time intervals. There are many different approaches and some of them include algebraic methods [Provot 1997; Kim and Rossignac 2003; Choi et al. 2008], swept volume [Hubard 1993; Abdel-Malek et al. 2002], adaptive bisection [Schwarzer et al. 2002], and conservative advancement methods [Mirtich 2000]. However, these methods are designed to particular types of input models and may be slow to be used in interactive applications.

CCD methods have been further optimized for rigid models [Redon et al. 2002] and articulated models [Zhang et al. 2007]. These methods are based on tight-fitting pre-computed hierarchies. CCD methods for deformable polygonal meshes are initially designed for meshes with fixed connectivity [Govindaraju et al. 2005; Wong and Baciu 2005; Hutter and Fuhrmann 2007] and, recently, are extended to models with topology changes [Curtis et al. 2008; Tang et al. 2008].

Culling techniques: CCD methods for inter-collision and intra-collisions, i.e., self-collisions, are very expensive and, thus, many culling techniques have been proposed. Tang et al. [2008] proposed continuous normal cones to identify mesh regions where self-collisions cannot occur and cull those regions from the consideration of self-collision detection. Also, Sean et al. [2008] proposed a new feature-based BVH, called Representative-Triangles, which partitions features (e.g., vertices, edges, and triangles of the mesh) and removes all the redundant elementary tests during self-collision detection. Our PCCD method is built on top of the feature-based BVH and can be combined together with the continuous normal cones.

2.2 Parallel Collision Detection

There are many collision detection methods parallelized by using GPUs and CPUs.

GPU-based parallel collision detection (CD): There have been considerable efforts to perform collision detection efficiently using GPUs [Heidelberger et al. 2004; Knott and Pai 2003; Govindaraju et al. 2003]. Govindaraju et al. [2003] proposed an approach for fast CD between complex models using GPU-accelerated visibility queries. Kolb et al. [2004] introduced an image-based CD algorithm for simulating a large particle system. There have been GPU-based algorithms for self-collision and cloth simulations [Vassilev et al. 2001; Baciu and Wong 2002; Govindaraju et al. 2005] specialized on certain types of input models (e.g., closed objects). These image-based techniques suit well with GPU architecture. However, due to the discrete image resolution, they may miss some collisions. Also, Gress et al. [2006] introduced a BVH-based GPU collision detection method for deformable parameterized surfaces. Sud et al. [2006] proposed a unified GPU-framework for various proximity queries including CCD.

Parallel CD on CPUs: There are relatively less work on parallel CD methods using CPUs. Lawler and Laxmikant [2002] proposed a voxel-based CD method for static model using distributed-memory parallel machines. They applied a generic load-balancing method to the problem of collision detection and achieved up to about 60% parallel efficiency. Figueiredo and Fernando [2004] designed a parallel CD algorithm for virtual prototype environment. This method is based on a culling method considering overlapping areas between BVs. However, it does not use bounding volume hierarchies to further improve the performance of collision detection. This method achieved its best performance, two times improvement, by using 4-cores over using a single-core and, then, showed lower performance as more CPUs are added. These parallel methods supported DCD of static models.

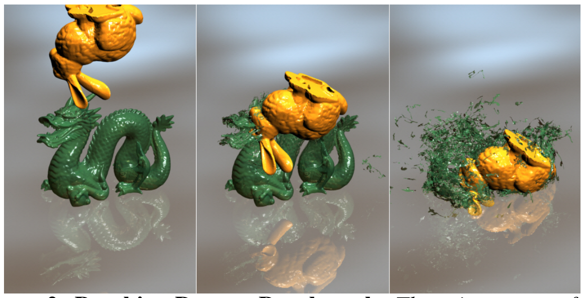
Figure 2: Breaking Dragon Benchmark: *These images are from the breaking dragon benchmark consisting of* 252 *K triangles. Our method shows* 6.7 *times improvement on average by using 8-cores and, thus, achieves an interactive performance (e.g.,* 7.7 *fps on average) for performing CCD in this benchmark.*

There are also parallel BVH construction methods. Wald [2007] proposed a fast parallel BVH construction method. Ize et al. [Ize et al. 2007] proposed an asynchronous BVH construction while using refitted BVHs. Once the construction is done, the constructed BVH is swapped with a BVH that is used for the BVH traversal. Although these techniques are designed for ray tracing, these methods can be used for collision detection and can be combined with our method.

3 Overview

In this section, we give a background on CCD and discuss issues of designing a parallel CCD method. Then, we give a brief overview of our approach to fully utilize multi-core CPU architecture for interactive CCD between deforming models.

3.1 Background on Continuous Collision Detection (CCD)

We perform CCD to compute the first time of contacts between deformable models and contact information necessary for various simulations including cloth simulation. There are two types of contacts: inter-collisions between two different models and intra-collisions, i.e., self-collisions, within a model. Our CCD algorithm detects these two types of contacts. These 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 [Provot 1997; Bridson et al. 2002].

To define a continuous motion of a primitive between two discrete time steps, a simple linear interpolation between two instances of the primitive is widely used in various interactive applications. In this linear interpolation, the VF and EE elementary tests reduce to solving cubic equations [Provot 1997], which are derived from coplanarity conditions. Our method also employs this simple linear interpolation method. However, our method can use other complex in-between motions such as a screwing motion [Redon et al. 2002].

Bounding volume hierarchies (BVHs): BVHs are widely used to accelerate the performance of DCD and CCD methods. There are different types of bounding volumes (BVs). Some of the commonly used BV types include simple shapes such as spheres [Hubbard 1993] and axis-aligned bounding boxes (AABBs) [van den Bergen 1997], or tight-fitting BVs such as oriented bounding boxes (OBBs) [Gottschalk et al. 1996], and discretely oriented polytopes (k-DOPs) [Klosowski et al. 1998], etc. We use the AABB representation due to its fast update method and wide acceptance in various collision detection methods [Teschner et al. 2005]. Given a BV node n of a BVH, we use notations of $L(n)$ and $R(n)$ to indicate the left and right nodes of the node n.

Feature-based BVHs: Each leaf node of common BVHs has triangles of models. An intermediate node has a BV that can contain all the triangles under the sub-tree rooted at the node. However, it has been known that there are many redundant VF and EE elementary tests by using such BVHs that partition each triangle in a leaf node for CCD. Recently, *Representative-triangle* [Curtis et al. 2008] has been introduced to address this problem. This method is based on a simple feature-based BVH, whose leaf node contains unique features (e.g., vertices, edges, and faces) of models instead of containing each triangle of the model. We particularly choose to use this simple feature-based BVH since it elegantly removes the redundant VF and EE elementary tests and maps well to parallel computation.

BVH update: As models are deforming, we have to update BVHs of such deforming models. There are three different approaches for BVH update methods. The first approach is to refit extents of BVs by traversing the BVH in a bottom-up manner [Larsson and Akenine-Möller 2006; James and Pai 2004]. This approach is called *BV-refitting*. Although the BV-refitting methods are quite fast, the quality, i.e., the culling efficiency, of BVHs becomes poor if models undergo drastic deformations [Yoon et al. 2007]. The second approach is to reconstruct BVHs from scratch every frame [Wächter and Keller 2006; Hunt et al. 2006] to maintain the high culling efficiency of BVHs. However, these reconstruction methods can take long computation time due to their high time complexity. Moreover, most portions of BVHs may not be accessed during the BVH traversal for collision detection due to the localized contacts between objects. Therefore, reconstructing whole BVHs has not been widely used for collision detection [Teschner et al. 2005]. The last approach is to combine the above two approaches and to selectively reconstruct small portions of BVHs that may have poor culling efficiency [Otaduy et al. 2007; Yoon et al. 2007; Larsson and Akenine-Möller 2006] and refit the rest of portions of BVHs. Typically, these selective reconstruction methods are combined with lazy BV constructions and show the best performance for collision detection in practice [Teschner et al. 2005].

3.2 BVH-based Collision Detection

When we perform collision detection between two objects, we first create a *collision test pair* consisting of two root BVs of two objects' BVHs. Then, we push the pair into a queue (or a stack). In the main loop of CD algorithm, we dequeue a pair consisting of nodes n and m from the queue and perform a BV overlap test between two BVs, n and m, of the pair. If there is an overlap, we refine two BVs with their two child BVs 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 $(L(m), R(m))$.

During performing collision detection, we may reach leaf nodes. In this case, we perform the VF and EE elementary tests between features associated with leaf nodes. We continue this process until there is no more collision pairs in the queue. Also, note that the inter-CD algorithm between two objects described above can be thought as a self-collision detection method of a virtual node whose two child nodes are root nodes of BVHs of those two objects. This technique can be applied to more than two objects. For the sake of the simplicity, we mainly focus on the self-collision detection method of a BVH, which may be constructed from BVHs of many objects.

3.3 Issues of Parallel CCD

In general, designing a parallel algorithm requires the following four steps [Culler and Singh 1999, pages 82 – 90]: 1) *decomposition* of computations into tasks, 2) *assignment* of tasks to threads, 3) *orchestration* of communication and synchronization among threads, and 4) *mapping* of threads to processors.

Conditions for accurate results: We should maintain that an output of our parallel CCD is same to that of the serial CCD method. To achieve this goal, it is required during the decomposition and assignment steps that we do not omit any computation as compared to those of the serial algorithm. Also, if two different computations have a particular order to be performed between them, the orchestration part of the parallel algorithm should maintain their computation ordering by using a proper synchronization.

Conditions for fast performance: The primary goal of designing a parallel algorithm is to take advantage of many-core architecture and, thus, improve the performance of a serial algorithm. It is desirable to avoid performing redundant computations, unless having them improves the overall performance of parallel algorithm. Also, communication and synchronization are very expensive operations in current parallel architecture and, thus, should be minimized [Culler and Singh 1999]. Finally, similar amounts of tasks of computations should be distributed to each thread to achieve high load-balancing among threads. Also, it is desirable to achieve high cache coherence in the process of the mapping different threads to processors.

Highly-scalable parallel CCD: It is relative easy to parallelize existing BVH-based continuous or discrete collision detection methods by decomposing collision pairs stored in the queue (or stack) and assigning partitioned collision pairs into threads. However, there are two major issues to design a highly-scalable parallel CCD method. First, collisions typically occur in very localized regions of meshes. Therefore, some of collision pairs may turn out to be non-colliding and, thus, terminate soon during the BVH traversal. On the other hand, other collision pairs may collide and require checking collisions between their child BVs until leaf nodes. As a result, it is very hard to predict the computation workload required to identify collisions from each collision pair. Therefore, it requires an efficient dynamic task re-assignment algorithm to achieve high load-balancing among different threads. Second, different threads can access a same BV during the BVH traversal. The problem arises when multiple threads access a same BV and trigger a lazy construction for the BV simultaneously. To guarantee the correct result of the parallel CCD, we can use synchronization mechanisms. However, a parallel algorithm with many synchronizations may not show high-scalability. Due to these two difficult issues, it is non trivial to design high-performing and highly-scalable parallel CCD methods for deforming models.

3.4 Overview of Our Approach

In order to take advantage of multi-core CPU architecture, we propose a novel parallel CCD method (PCCD) supporting inter-collisions and self-collisions between deforming models. At a high level, our PCCD method consists of two parts: 1) BVH refitting component and 2) collision detection with lazy BV reconstructions.

Our PCCD method updates a BVH of a deforming model by performing the BV refitting before we perform any collision detection. Since the BV refitting alone may not provide BVHs with high culling efficiency, we selectively perform lazy BV reconstructions during the collision detection part while traversing the BVH.

Our PCCD method is based on a novel task decomposition, *self-CD task unit*, to achieve high scalability as a function of the number of cores. A self-CD task unit is a set of collision test pairs generated by performing a collision test pair between two child nodes of a node. Particularly, we identify self-CD task units that do not access a same node during processing each task unit and, thus, perform the lazy BV reconstruction without the use of any expensive locking mechanisms, while processing these self-CD tasks in a parallel. To achieve a high load-balancing among computational workloads assigned to each thread, we employ a simple dynamic task assignment method that partitions self-CD task units of a thread to other threads that finish the computation earlier than other threads. We also pro-

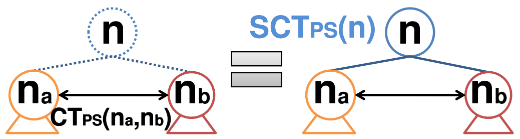

Figure 3: Converting an Inter-CD to a Self-CD: *An inter-collision detection between two nodes n_a and n_b is equal to a self-collision detection of a node n, where n_a and n_b are two child nodes of the node n.*

pose a method parallelizing each self-CD task unit for cases where there are not enough self-CD task units to utilize all the available threads.

4 Self-CD based Parallel CCD

In this section we explain our novel decomposition and task re-assignment methods for our PCCD method.

Terminology: We define a few terminologies to describe our methods. We define a *collision test pair set*, $CT_{PS}(n_a, n_b)$, to be all the collision test pairs generated by performing a collision test pair (n_a, n_b) of two nodes n_a and n_b to find inter-collisions between those two nodes. Since an inter-collision detection between two nodes can be thought as a self-collision detection within a virtual node n, we also use a *self collision test pair set*, $SCT_{PS}(n)$ to denote $CT_{PS}(n_a, n_b)$, where $L(n) = n_a$ and $R(n) = n_b$). An example of converting an inter-collision detection between two nodes into a self-collision detection of a node is shown in Fig. 3. We define that two nodes have a *parent-child relationship* if one of nodes is in the sub-tree rooted at another node.

4.1 Self-CD based Decomposition

In this section we explain our decomposition of computations to perform collision detection. Our method is based on task units of a self-collision detection, *self-CD*, each of which performs a self-collision detection of a node.

Computation of a self-CD task unit: Each self-CD task unit processes collision test pairs represented by $SCT_{PS}(n)$ of a node n. To assign task units to each thread, we push a node n into a *task unit queue* and associate the queue with a thread. Self-CD task unit has two phases: 1) setup phase and 2) collision detection phase. In the setup phase, we first fetch a node from the task unit queue and refine the node n into its two child nodes $L(n)$ and $R(n)$. Then, we push those two child nodes into the task unit queue. We also create a *scheduling queue* for dynamic task assignment to achieve a high load-balancing, which will be explained later. Then, we perform the collision detection phase. First, we create a *collision test pair queue* and assign a collision test pair $(L(n), R(n))$ into the collision test pair queue. We fetch a pair consisting of two nodes n and m from the collision test pair queue and, then, perform a BV overlap test between two BV nodes n and m of the pair. If there is a collision, 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 $(R(n), R(m))$. We continue this process until we reach leaf BV nodes. If we reach leaf BV nodes, we perform exact VF and EE elementary tests between features associated with the leaf BV nodes. If there is any collision, we put the collision result into a *result queue*.

Disjoint property of task units: During processing a self-CD task unit of a node n, $SCT_{PS}(n)$, we create and test various collision test pairs 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 $SCT_{PS}(n)$ is disjoint from another set of accessed nodes during performing $SCT_{PS}(m)$. We will utilize this disjoint property to design an efficient parallel CCD algorithm.

Serial CCD method: Before we explain our PCCD method, we first explain how to perform CCD with a single thread based on self-CD task units. Given two objects o_a and o_b with two BVHs whose two root nodes are n and m, we first create a virtual node v and set its child nodes to be n and m. Then, we perform a self-CD task unit, $SCT_{PS}(v)$, of the node v. At the end of processing the task unit of $SCT_{PS}(v)$, the collision test pair queue is empty. However, the task unit queue can have two nodes, whose are two child nodes of v. We fetch a node, n, from the task unit queue and perform $SCT_{PS}(n)$. We continue this process until there is no node in the task unit queue. Then, the result queue contains all the collisions within each object and between two objects o_a and o_b. A main property in this serial CCD method is that any pair of nodes in the task unit queue do not have parent-child relationship. Also note that this CD algorithm is easily extended to detect collisions among any number of objects.

Our serial CCD algorithm does not have any redundant collision test pairs nor miss any collision test pairs compared to a common BVH-based collision detection method explained in Sec. 3.2. Our serial CCD algorithm is constructed by simply reordering the processing order of collision test pairs of typical BVH-based CD methods into self-CD task units.

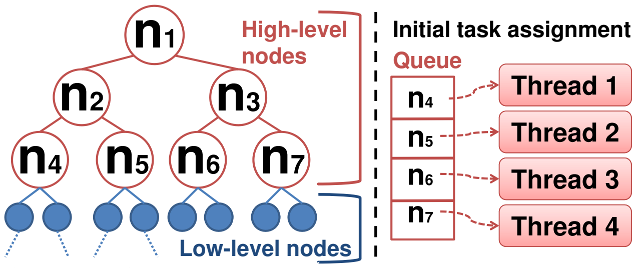

Figure 4: High-Level and Low-Level Nodes: *This figure shows high-level and low-level nodes given 4 available threads. The right portion of the figure shows an initial task assignment for 4 threads.*

4.2 Initial Task Assignment

We design each thread to process self-CD task units. Initially, each thread is initialized with a node n. Then, each thread performs a self-CD task unit represented by $SCT_{PS}(n)$. A node of BVH is guaranteed to be accessed by only one thread at any time, if there are no parent-child relationships among nodes assigned to each thread due to the disjoint property of self-CD task units. In this case, we do not need to use expensive locking mechanisms to prevent that multiple threads attempt to reconstruct a same BV node for a lazy BV reconstruction during the BVH traversal.

To guarantee that nodes assigned to each thread do not have any parent-child relationship, we use the following simple initial task assignment method. Suppose that there are p available threads that we can utilize for the parallel computation. We first traverse a BVH in a breath-first order. During the breadth-first order traversal of the BVH, we maintain a queue containing nodes in the front of the traversal. If the size of the queue is p, then, we stop the BVH traversal and assign each node of the queue into each thread. We call those nodes and all the nodes in the sub-trees rooted at those nodes *low-level nodes*. We call all the other nodes *high-level nodes*. An example of low-level and high-level nodes for 4 threads is shown in Fig. 4.

Each thread performs a self-CD task unit, $SCT_{PS}(n)$, of an initially assigned node n. At the end of performing $SCT_{PS}(n)$, a task unit queue of the thread can have two nodes which are two child nodes of the node n. Then, the thread fetches a node from the queue and performs the self-CD task unit of the node.

Once each thread finishes its computation, then, we process self-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 self-CD task units of two nodes n_2 and n_3 finishes and, then, process their parent node, n_1. During processing high-level nodes, we do not add child nodes of

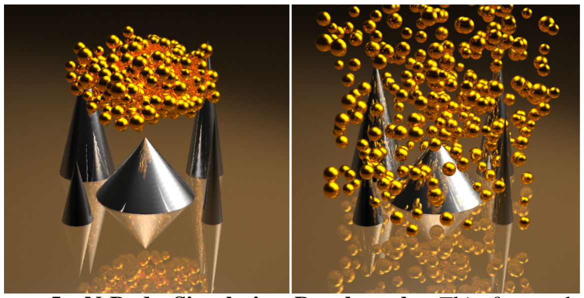
Figure 5: N-Body Simulation Benchmark: *This figure shows two frames during the N-body simulation benchmark consisting of* 146 *K triangles. Our method spends* 138 *ms on average and achieves* 7.3 *times performance improvement by using 8-cores over using a single-core.*

those nodes to the task unit queue since we already processed self-CD tasks of those child nodes. Note that there are no parent-child relationships among high-level nodes processed in parallel.

In this simple task assignment method, we can very efficiency traverse the BVH, perform the overlap tests, and invoke lazy BV reconstructions, if necessary, without any locking procedure. However, a thread can finish its assigned task units much faster than other threads due to the localized contacts among objects. In this case, it is desirable to divide task units of a thread to the thread finishing its task in order to fully utilize all the available p threads. For this, we propose a dynamic task assignment in the next section. Also, during processing high-level nodes, the number of self-CD task units is less than the number, p, of available threads. To utilize all the available threads, we also propose a method efficiently parallelizing each self-CD task unit in Sec. 5.

4.3 Dynamic Task Assignment

Suppose that a thread $t_{request}$ finishes its computation and there is no more node left in the task unit queue. In order to detect another thread t_{src} that can give its computation workload to the thread $t_{request}$, a scheduler managing the dynamic task assignment is required. We can use a centralized master scheduler running in a dedicated thread. In this approach, the requesting thread $t_{request}$ first accesses the master scheduler. Then, the master scheduler accesses global information on the available computational workload in each thread and chooses a thread to distribute its workload to the requesting thread $t_{request}$. However, a cost for the context switching to the master scheduler thread is a high. Moreover, the access to the master thread has to be serialized and, thus, it can lower the scalability of the overall PCCD method. Instead, we embed our scheduler in each thread and use it if necessary.

In order to choose a thread t_{src} that has enough workload for the requesting thread, our scheduler accesses the other thread's information. For example, each thread has its computation workload in its collision test pair queue and task unit queue. Note that any locking mechanisms are not required to read the sizes of these queues of other threads. We choose to distribute nodes stored in the task unit queue to the requesting threads. Note that each node in the task queue represents a self-CD task unit. The main reason why we do not distribute pairs in the collision test pair queue is that if we distribute pairs into other threads, the disjoint property of task units would not be applicable and, thus, it would require an expensive locking mechanism for the lazy BV reconstruction method.

Suppose that the scheduler of a thread $t_{request}$ chooses a thread t_{src} to get its computation workload. In order to request the distribution of computation workloads, the requesting thread $t_{request}$ places a request to the scheduling queue of the thread t_{src}. To place the request, a locking to the scheduling queue is required since other threads may want to request and access the same scheduling queue. Once placing the request, the thread $t_{request}$ 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 self-CD task unit. Since we do not need to lock the scheduling queue to see whether it is empty, it can be done quite efficiently. If there are no requests in the queue, the thread can continue to process another self-CD task unit by fetching a node from its task unit queue. If there are non-zero k requests in the scheduling queue, we partition its computational workload into $k+1$ sets while maintaining each partitioned workload to be roughly equal. Since we also have to leave a workload for its own thread, we partition the computation workload into $k+1$. Note that nodes in the task unit queues do not have parent-child relationships. Therefore, each thread can perform its self-CD task unit without any modification and locking during processing collision test pairs. Then, the thread t_{src} sends a wake-up message with the partitioned nodes to the requesting threads.

Once a thread $t_{request}$ receives the wake-up message, the thread assigns the received node into its task unit queue and resumes its computation by performing self-CD task units. Pseudocode of our parallel CCD algorithm based on self-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 (a 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;
  }

  Run a scheduling algorithm and sleep;
}
```

Listing 1: *Pesudocode of PCCD method*

Evaluating and partitioning the computation workload: Suppose that we choose a thread to distribute its computation workload to another requesting thread. Then, we have to partition its computation workload into equal-sized workloads. However, it is hard to predict how much computation workload processing each node in the task unit queue will require. Instead of designing a complex prediction method, we simply return the first node, i.e., the front node, in the queue to the requesting thread. Note that the front node in the queue is likely to cause the most significant or equal computational workload compared to other nodes in the queue, given the breadth-first order traversal of the BVH during processing self-CD task units. Given this simple partitioning scheme, we only need to look at the first node to evaluate the workload of each thread. More specifically, we look at the first node and see how many triangles are located under the sub-tree rooted at the node. The number of triangles under a node is already available since this information is necessary for the lazy BV reconstruction method. As a node has more triangles, we can expect that the computation workload of processing a self-CD task unit of the node is higher. Based on this simple evaluation method, we choose a thread, t_{src}. Note that there may be multiple requests in the scheduling queue. Since each thread will get the first node in the queue, we look at a node located at $k+1$th position in the queue if there are already k requests in the scheduling queue for the evaluation of the computation workload of each thread.

5 Parallelizing a Self-CD Task Unit

In this section we present a method parallelizing each self-CD task unit.

5.1 Task Decomposition for Self-CD Task Unit

During processing high-level nodes, the number of self-CD task units that can run in parallel is smaller than the number of available threads. For example, at the last step of our PCCD algorithm, we process a single self-CD task unit of a root node of the BVH. The algorithm described in the previous section can use only a single thread processing the self-CD task unit although there may be many available threads. In order to fully utilize all the available threads, we propose an algorithm that performs a self-CD task unit in a parallel manner. We designed two different methods parallelizing a self-CD task unit.

Lazy reconstruction without a locking mechanism: During processing a self-CD task unit, we process each collision test pair located in the collision test pair queue. If nodes accessed during processing a set of collision pairs are disjoint from nodes of other sets, we can process those sets of pairs in parallel without any locks for lazy BV reconstructions. Our first method is based on a decomposition of collision test pairs satisfying the above requirement. Suppose that we are performing a self-CD task unit of a node n. Then, we first perform a collision test pair between $L(n)$ and $R(n)$. If there is a BV overlap between two nodes, we create four refined collision test pairs, $(L(n), L(m))$, $(L(n), R(m))$, $(R(n), L(m))$, and $(R(n), R(m))$. Note that two collision pairs of $(L(n), L(m))$ and $(R(n), R(m))$ do not share any same node. Therefore, their refined collision test pairs cannot access a same node. Therefore, we can perform these two collision test pairs into two threads in a parallel manner. Then, we wait for the termination of two threads processing these pairs and, then, we can do a similar parallel computation for another two collision test pairs of $(L(n), R(m))$ and $(R(n), L(m))$. The main advantage of this approach is that we do not need any locking mechanism for lazy reconstructions during the BVH traversal. However, we found that this method shows very poor scalability (e.g., 2 times speedup by using even 8 cores). Since the proposed algorithm requires synchronization before processing another two pairs and computation time of processing a collision test pair may vary a lot, this method does not effectively utilize available threads.

Lazy reconstruction with a locking mechanism: Our second method is based on a simple observation: we do not perform many lazy BV reconstructions during processing high-level nodes since we already traversed many nodes and performed lazy reconstructions during processing self-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 for a thread to wait for a locked node. Since we guarantee to avoid performing multiple lazy reconstructions on a same node simultaneously by using a locking mechanism, we can arbitrarily partition the pairs of the collision test pair queue into k 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. We found that this method works well.

6 Parallel BVHs Update

In this section, we explain our parallel BVH update method to efficiently deal with deforming models.

6.1 BV Refitting

Before we perform the CCD during traversing a BVH, we first update the BVH. Our BVH update method combines BV refitting and selective BV reconstruction methods. Our algorithm first traverses the BVH in a bottom-up manner and refits the BVs. To design a parallel BVH update method utilizing k threads, we traverse the BVH in a breadth-first order using a queue until the queue has k nodes. Then, we assign each node in the queue to a thread and, then, each thread performs the BV refitting to the sub-tree rooted at the node. Since the BVH is unlikely to be balanced, a thread can finish its BV refitting earlier than other threads. We can use a similar dynamic task assignment explained in Sec. 4.3 for dynamic load balancing of the parallel BVH refitting method. However, we found this approach is rather a heavy-weight solution for this problem and requires some overhead for the BV refitting process.

Model	Triangles (K)	Number of frames	Representative image	CCD time (ms)
Cloth simulation	92	465	Fig. 1	56
Exploding dragon	252	480	Fig. 2	130
N-body simulation	146	375	Fig. 5	138

Table 1: Dynamic Benchmark Models: *This table shows the complexity of the benchmarks with their representative images, the number of frames used in tests, and the CCD computation time on average.*

Instead, we use a simple load-balancing scheme. Given k available threads, we collect $2k$ nodes in the queue during the breadth-first order BVH traversal. Then, we assign the first k nodes into each thread in a round robin fashion. If a thread finishes its job, we assign the next available node in the queue to the thread.

6.2 Selective BV Reconstruction

BV refitting methods works quite well for dynamic models with minor deformations. However, for models with topological changes and drastic deformations, BV refitting methods show poor performance due to the expanded BVs. We use a selective BV reconstruction method to address the problem of the BV refitting and improve the performance of CCD. To identify BVs whose culling efficiency is lower and can be improved by the BV reconstruction, we use a heuristic metric proposed by Larsson and Akenine-Möller [2006]. This metric measures a ratio of the volume of a BV node to the sum of volumes of its child nodes. If the ratio is less than 0.9 as suggested by [Larsson and Akenine-Möller 2006], we consider the node to have low culling efficiency.

Lazy reconstruction: For the BV reconstruction of a node, we use a simple median-based partitioning of triangles associated with the node. Although this method runs fast, the reconstruction time of a node is much more expensive than a BV overlap test. Also, due to the localized contacts among objects, small portions of a BVH are likely to be accessed during performing collision detection. Therefore, we employ a lazy reconstruction method, which reconstructs a node when we access the node and find that the culling efficiency of the node is low based on the metric we described above.

7 Implementation and Results

In this section we describe our implementation and highlight the results of our PCCD method with different benchmarks.

We have implemented our PCCD method on an Intel Xeon desktop machine with two 2.83 GHz quad-core CPUs and WindowsXP. We use *OpenMP* library [Dagum and Menon 1998] to parallelize our method. We rely on OpenMP for the mapping of threads to processors. We construct a BVH for each object and, then, merge these multiple BVHs into a one big BVH by recursively merging two BVHs into one. Then, a self-collision detection of a merged single BVH is equal to inter-collision detections among multiple objects.

Benchmarks: We test our method with three types of dynamic scenes. These include:

- **Cloth simulation:** A cloth drapes on a ball and, then, the ball is spinning (Fig. 1). This benchmark consists of 92 K triangles

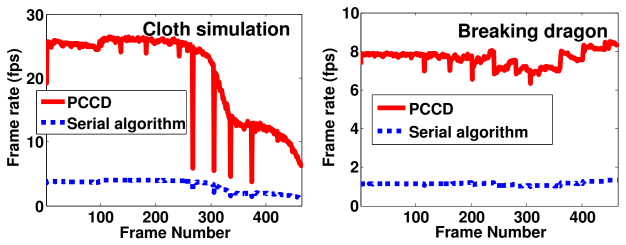

Figure 6: Frame Rates: *These two graphs show frame rates, frames per second (fps), of the cloth simulation (Fig. 1) and the breaking dragon benchmarks (Fig. 2). Our method achieves 17.9 fps and 7.7 fps for two benchmarks respectively by using 8-core machine. Compared to the single-core method, our method is able to achieve about 7 times performance improvement for two tested benchmarks.*

and undergoes severe non-rigid deformations.

- **Breaking dragon:** A bunny collides with a dragon model. Then, the dragon model breaks into numerous pieces (Fig. 2). This model has 252 K triangles.
- **N-body simulation:** This benchmark consists of multiple moving objects consisting of 146K triangles (Fig. 5). Each object may undergo a rigid or deformable motion and objects collide with each other and the ground.

These models are downloaded from the UNC dynamic scene benchmarks[1]. These models have different model complexity and characteristics. As a result, they suit well to testing the performance of our algorithm. Detail information of each benchmark is shown in Table 1.

7.1 Results

We test our PCCD method with 8 cores and measure the time spent on performing the CCD including self-collision detections. Our method spends 56 milliseconds (ms), 130 ms, and 138 ms on average for the cloth simulation, the breaking dragon, and N-body simulation respectively. These computation times translates to 17.9, 7.7, and 7.3 frame per seconds (fps) on average for these three benchmarks respectively. The frame rates of our PCCD method for the cloth simulation and the breaking dragon benchmarks are shown in Fig. 6. The downward spikes appeared in the graph of frame rate in the cloth simulation benchmark are caused by infrequent BV reconstructions of high regions of BVHs, which require high computation time.

We also measure the performance of our PCCD method as a function of the number of cores. Particularly, we measure the performance of the PCCD method with 1, 2, 4, and 8 cores. We are able to achieve high scalability for our method with different number of cores. For example, our PCCD method shows 6.4, 6.7, and 7.3 times performance improvement by using 8-cores over a single-core version in the cloth simulation, the breaking dragon, and N-body simulation respectively. The performance of the PCCD method as a function of the number of cores is shown in Fig. 7.

7.2 Analysis and Comparison

At a high level, our PCCD method consists of four components; 1) BV refitting, 2) parallel CCD with low-level nodes, 3) parallel CCD with high-level nodes, and 4) other serial components and miscellaneous parts (e.g., setup of the threads). The portion of each component over the whole computation is shown in Fig. 8 and the scalability of each component is shown in Fig. 9.

Parts of BV refitting and parallel CD with high-level nodes take small portions, 7%–13% and 2%–10% respectively, of the whole computation. Since contacts occur in a few localized regions of

[1] `http://gamma.cs.unc.edu/DynamicB`

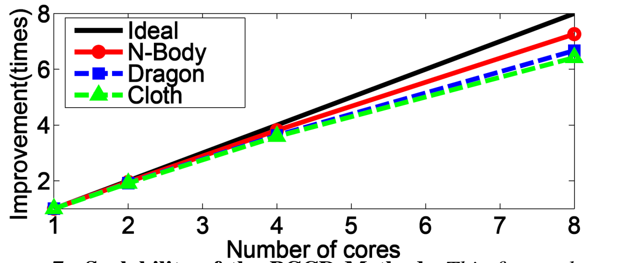

Figure 7: Scalability of the PCCD Method: *This figure shows the performance improvement of our PCCD method as a function of the number of cores over using a single-core with different benchmarks. Ideal performance is linearly increased from the single-core performance.*

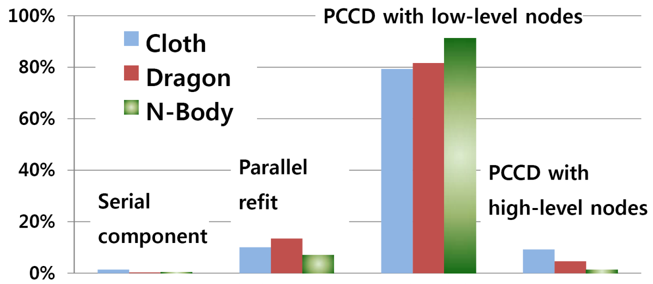

Figure 8: Portions of Components of PCCD Method: *This graph shows relative portions of different components of our PCCD method with different benchmarks. The most time consuming part is processing self-CD task units for low-level nodes.*

meshes, it is very hard to predict the amount of work associated with each collision pair and a node. Therefore, it is hard to achieve a high load-balancing among different threads. Because of this, these two parts show relatively low scalability (e.g., four times improvement by using 8-cores over the single-core performance). The part of parallel CCD with low-level nodes takes a major portion, 80%–90%, of our whole computation. This part shows more than 7 times improvement by using 8-cores over the single-core version due to the dynamic load-balancing method. Since the portion of processing self-CD tasks of low-level nodes is the largest, its high scalability translates to the high scalability of our overall PCCD method. Also, serial components of our PCCD method take a very small portion (e.g., less than 2%).

Comparison with GPU-based methods: We compare our algorithm over the state of the art GPU-based CCD method proposed by Sud et al. [2006]. In this method, they also used a cloth simulation benchmark consisting of 15 K triangles similar to our cloth simulation benchmark although the model complexity is different. They used AMD Athlon 4800 X2 CPU and a GeForce 7800 GPU. Their method spends about 700 ms. Note that our method spends 56 ms in our cloth simulation benchmark model and the model complexity of our cloth simulation benchmark is 6 times higher than the one used in [Sud et al. 2006]. GPU-based techniques based on data read-backs like [Sud et al. 2006] may achieve much lower performance improvement compared to the GPU performance improvement since the performance of read-backs has not been improved in past years. Also, according to the paper of [Sud et al. 2006], their reported read-back time spanned between 50 ms and 60 ms, which is comparable even to the whole computation time of our PCCD method in our cloth simulation benchmark consisting of 92 K triangles.

8 Conclusion and Future Work

We have presented a novel parallel continuous collision detection method utilizing the multi-core CPU architecture. Our method is based on a novel task decomposition, self-CD task unit, to reduce dependencies and synchronizations among different threads.

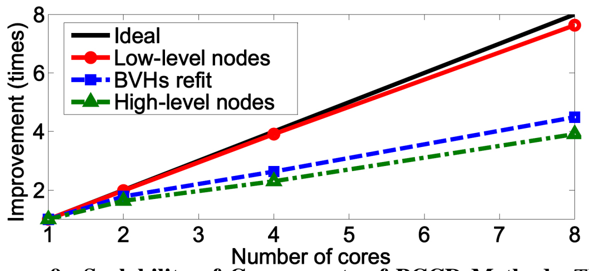

Figure 9: Scalability of Components of PCCD Method: *This figure shows the scalability of different components of our PCCD method as a function of the number of cores.*

Also, we have proposed a simple, but efficient dynamic task reassignment method to achieve high load-balancing among different threads. Our method can achieve interactive performance for CCD including self-collision detection between deforming models consisting of hundreds of thousand triangles. Moreover, we are able to achieve up to 7.3 times performance improvement by using 8-cores over using a single-core.

There are many avenues for future work. First we would like to extend our current PCCD method to exploit the SIMD functionality of commodity architecture. This will be particularly important for Larrabee architecture since it has 16 wide SIMD functionality. Also, we would like to test our method with a parallel machine with more than 8-core machines and further improve the scalability of our method. Currently, 16-core machines have low performance/-cost ratio compared to 8-core machines. We expect that this will be resolved soon. Finally, we would like to design parallel algorithms for other proximity queries including minimum separation distance and penetration depth queries.

References

ABDEL-MALEK, K., BLACKMORE, D., AND AND, K. J. 2002. Swept volumes: foundations, perspectives, and applications. *International Journal of Shape Modeling 12*, 1, 179–197.

ASANOVIC, K., R., CATANZARO, B., GEBIS, J., HUSBANDS, P., K., PATTERSON, D., PLISHKER, W., SHALF, J., WILLIAMS, S., AND YELICK, K. 2006. The landscape of parallel computing research: A view from Berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Dept., Univ. of California, Berkeley.

BACIU, G., AND WONG, S. 2002. Image-based techniques in a hybrid collision detector. *IEEE Trans. on Visualization and Computer Graphics 9*, 2, 254–271.

BORKAR, S. 2007. Thousand core chips – a technology perspective. *ACM/IEEE Design Automation Conference*, 746–749.

BRIDSON, R., FEDKIW, R., AND ANDERSON, J. 2002. Robust treatment for collisions, contact and friction for cloth animation. *ACM SIGGRAPH*, 594–603.

CHOI, Y.-K., WANG, W., LIU, Y., AND KIM, M.-S. 2008. Continuous collision detection for two moving elliptic disks. *IEEE Trans. on Robotics 22*, 2, 213–224.

CULLER, D., AND SINGH, J. P. 1999. *Parallel computer architecture: a hardware/-software approach.* Morgan Kaufmann.

CURTIS, S., TAMSTORF, R., AND MANOCHA, D. 2008. Fast collision detection for deformable models using representative-triangles. *Symp. on Interactive 3D Graphics*, 61–69.

DAGUM, L., AND MENON, R. 1998. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering 5*, 46–55.

ERICSON, C. 2004. *Real-Time Collision Detection.* Morgan Kaufmann.

FIGUEIREDO, M., AND FERNANDO, T. 2004. An efficient parallel collision detection algorithm for virtual prototype environments. *International Conf. on Parallel and Distributed Systems*, 249–256.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph*, 171–180.

GOVINDARAJU, N., REDON, S., LIN, M., AND MANOCHA, D. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *EG. Workshop on Graphics Hardware*, 25–32.

GOVINDARAJU, N., KNOTT, D., JAIN, N., KABAL, I., TAMSTORF, R., GAYLE, R., LIN, M., AND MANOCHA, D. 2005. Collision detection between deformable models using chromatic decomposition. *ACM Trans. on Graphics 24*, 3, 991–999.

GRESS, A., GUTHE, M., AND KLEIN, R. 2006. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum 25*, 3, 497–506.

HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2004. Detection of collisions and self-collisions using image-space techniques. *Proc. of Winter School of Computer Graphics 12*, 3, 145–152.

HUBARD, P. M. 1993. Space-time bounds for collision detection. Tech. Rep. cs-93-04, Dept. of Computer Science, Brown University, Feb.

HUBBARD, P. M. 1993. Interactive collision detection. *Proc. of IEEE Symposium on Research Frontiers in Virtual Reality* (October), 24–31.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. *IEEE Symposium on Interactive Ray Tracing*, 81–88.

HUTTER, M., AND FUHRMANN, A. 2007. Optimized continuous collision detection for deformable triangle meshes. *Proc. Winter School of Computer Graphics*, 25–32.

INTEL, 2006. Intel 80 core prototype. Tech-research web site: http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

IZE, T., WALD, I., AND PARKER, S. G. 2007. Asynchronous bvh construction for ray tracing dynamic scene on parallel multi-core architectures. *Eurographics Symposium on Parallel Graphics and Visualization*.

JAMES, D. L., AND PAI, D. K. 2004. BD-Tree: Output-sensitive collision detection for reduced deformable models. *Proc. of ACM SIGGRAPH*, 393–398.

KIM, B., AND ROSSIGNAC, J. 2003. Collision prediction for polyhedra under screw motions. *ACM Symposium on Solid Modeling and Applications*, 4–8.

KLOSOWSKI, J., HELD, M., MITCHELL, J., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1, 21–36.

KNOTT, D., AND PAI, D. K. 2003. CInDeR: Collision and interference detection in real-time using graphics hardware. *Proc. of Graphics Interface*, 73–80.

KOLB, A., LATTA, L., AND REZK-SALAMA. 2004. Hardware-based simulation and collision detection for large particle system. *EG. Symp. on Graphics hardware*, 123–131.

LARSSON, T., AND AKENINE-MÖLLER, T. 2006. A dynamic bounding volume hierarchy for generalized collision detection. *Computers and Graphics 30*, 3, 451–460.

LAWLOR, O. S., AND LAXMIKANT, V. K. 2002. A voxel-based parallel collision detection algorithm. *Supercomputing*, 285–293.

LIN, M., AND MANOCHA, D. 2003. Collision and proximity queries. *Handbook of Discrete and Computational Geometry*.

MIRTICH, B. V. 2000. Timewarp rigid body simulation. *Proc. of the 27th annual conference on Computer graphics and interactive techniques*, 193–200.

OTADUY, M., CHASSOT, O., STEINEMANN, D., AND GROSS, M. 2007. Balanced hierarchies for collision detection between fracturing objects. *IEEE Virtual Reality Conference*, 83–90.

PROVOT, X. 1997. Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*, 177–189.

REDON, S., KHEDDAR, A., AND COQUILLART, S. 2002. Fast continuous collision detection between rigid bodies. *Computer Graphics Forum 21*, 3, 279–287.

REDON, S., KIM, Y. J., LIN, M. C., AND MANOCHA, D. 2005. Fast continuous collision detection for articulated models. *Computing and Information Science in Engineering 5*, 2, 126–137.

SCHWARZER, F., SAHA, M., AND LATOMBE, J.-C. 2002. Exact collision checking of robot paths. *Workshop on Algorithmic Foundations of Robotics*.

SUD, A., GOVINDARAJU, N., GAYLE, R., KABUL, I., AND MANOCHA, D. 2006. Fast proximity computation among deformable models using discrete voronoi diagrams. *Proc. of ACM SIGGRAPH*, 1144–1153.

TANG, M., CUITS, S., EUI YOON, S., AND MANOCHA, D. 2008. Interactive continuous collision detection between deformable models using connectivity-based culling. *ACM Symp. on Solid and Physical Modeling*, 25 – 36.

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

VAN DEN BERGEN, G. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4, 1–13.

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

WÄCHTER, C., AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. *Proceedings of the Eurographics Symposium on Rendering*, 139–149.

WALD, I. 2007. On fast construction of sah-based bounding volume hierarchies. *IEEE Symposium on Interactive Ray Tracing*, 33–40.

WONG, W. S.-K., AND BACIU, G. 2005. Dynamic interaction between deformable surfaces and nonsmooth objects. *IEEE Trans. on Visualization and Computer Graphics 11*, 3, 329–340.

YOON, S., CURTIS, S., AND MANOCHA, D. 2007. Ray tracing dynamic scenes using selective restructuring. *Proc. of Eurographics Symposium on Rendering*, 73–84.

ZHANG, X., REDON, S., LEE, M., AND KIM, Y. J. 2007. Continuous collision detection for articulated models using taylor models and temporal culling. *ACM Transactions on Graphics 26*, 3, 15.