MCCD: Multi-Core Collision Detection between Deformable Models using Front-Based Decomposition

Min Tang[1], Dinesh Manocha[2], Ruofeng Tong[1]

http://www.cs.unc.edu/~geom/PCD/

[1]*Department of Computer Science, Zhejiang University, China*

[2]*Department of Computer Science, University of North Carolina at Chapel Hill, USA*

Abstract

We present a novel parallel algorithm for fast continuous collision detection (CCD) between deformable models using multi-core processors. We use a hierarchical representation to accelerate these queries and present an incremental algorithm that exploits temporal coherence between successive frames. Our formulation distributes the computation among multiple cores by using fine-grained front-based decomposition. We also present efficient techniques to reduce the number of elementary tests and analyze the scalability of our approach. We have implemented the parallel algorithm on 8 core and 16 core PCs, and observe up to 7X and 13X speedups respectively, on complex benchmarks.

Key words: Continuous collision detection, Deformable models, Parallel collision detection, Bounding volume hierarchies

1. Introduction

Fast continuous collision detection (CCD) is an important problem that arises in physically-based simulation, virtual environments and robot motion planning. CCD can check for collisions between the discrete positions of the objects by computing an interpolatory motion and reducing the problem of collision checking to finding roots of non-linear polynomial equations [1]. In practice, CCD also is used to compute the first time of contact between the discrete instances.

In this paper we mainly deal with fast CCD computation between non-rigid models, where the scene may consist of breaking objects or objects undergoing deformable motion. This problem has been studied in the literature and many efficient algorithms are known. However, current methods are unable to offer interactive performance (i.e. tens of milliseconds or less) on complex benchmarks.

One of the goals of this paper is to exploit the current architectural trends for faster collision queries. Recent and future commodity processors are becoming increasingly parallel. At a broad level, there are two kinds of commodity processors: multi-core CPUs that include the best performing serial cores and many-core processors (e.g., GPUs) that are designed with the goal of achieving higher parallel code performance. These processors have different characteristics, and in this paper we limit ourselves to using multi-core CPUs to accelerate CCD queries. This is orthogonal to recent work on using GPUs for faster collision detection.

Most of the current computers, including desktops and laptops, have dual or quad core processors. Moreover, high-end desktop workstations used for CAD/CAM or virtual prototyping may consist of 8-16 cores. The number of core is expected to increase at the rate corresponding to Moore's Law. Given these architectural tends, there is relatively little work on developing faster algorithms for CAD/CAM systems and interference computations that can exploit the multiple cores.

One of the main challenges in developing efficient parallel collision detection algorithm is balancing the load evenly among multiple cores and obtain high memory and cache throughput. Most prior collision detection algorithms use bounding volume hierarchies to accelerate the computations. In case of deformable models, the traversal cost of the hierarchies can vary considerably based on the relative configuration of the primitives of the models. This can result in varying loads and irregular access patterns. One of the challenges is to design parallel hierarchy traversal algorithms that can work well on all different configurations and can scale well with the number of core.

Our parallel approach is based on recent work on CCD between deformable models [2, 3]. Specifically, we present a novel parallel CCD algorithm that maps well to current multi-core CPU architectures. Our algorithm performs incremental computations that utilize coherence between successive frames and also checks for self-collisions among deformable models (Fig. 1). We use the notion of BVTT (bounding volume traversal tree) and maintain a BVTT front. This front is updated in an incremental manner and is used to decompose the overall computation into sub-tasks among multiple cores. We also present efficient parallel techniques to adaptively update the front and reduce the number of elementary test between the primitives.

Main Contributions:

1. We extend prior front-based incremental algorithms for rigid models to deformable models by including self-

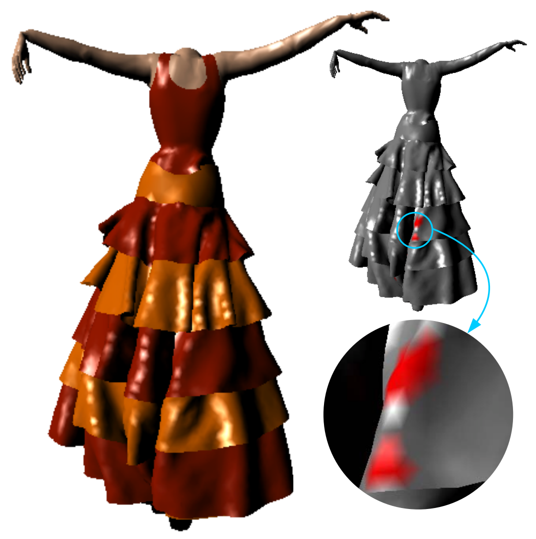

Figure 1: **Self-collisions among deformable models:** A large number of self-collisions appear on the waving skirt. The colliding areas are highlighted.

collision checking. A 2-3 tree representation is used for the resulting BVTT instead of the conventional binary tree representation, and we show that our algorithm can reduce the traversal cost significantly.

2. We present adaptive strategies to compute the BVTT front by proposing new metrics, and reduce the number of elementary test between the primitives based on orphan sets [3].
3. We present a highly scalable parallel CCD algorithm using front-based decomposition (FBD), and observe almost linear speedups with the number of core on a 16-core workstation. The algorithm tends to minimize the synchronization overhead between different threads and uses cache-friendly layouts for improved performance.
4. We analyze the complexity and scalability of the FBD algorithm, and compare with prior parallel algorithms. We show that our FBD algorithm offers improved theoretical and empirical performance.

The parallel CCD algorithm has been tested on many complex benchmarks with $4K - 92K$ triangles that correspond to deformable simulations, breaking objects and N-body simulations. As compared to an optimized serial implementation [3], we obtain $6.4X - 7.7X$ speedups on a 8-core workstation, and $10.1X - 13X$ speedups on a 16-core workstation. In practice, our algorithm can check for all collisions, including self-collisions, at $5.3ms - 32.5ms$ per frames on these benchmarks. To the best of our knowledge, this is the fastest known algorithm for parallel collision detection running on commodity systems.

Organization:

The rest of the paper is organized as follows: Sec. 2 gives a brief survey of prior work in CCD and parallel collision detection. We introduce our notation and address some issues in the design of parallel collision detection algorithms in Sec. 3. The overall parallel CCD algorithm is presented in Sec. 4, and we analyze its performance in Sec. 5. We present implementation details and the results from benchmarks in Sec. 6. We compare its performance with prior approaches and highlight some limitations in Sec. 7.

2. Related Work

Collision detection problem has been extensively studied in computer graphics, simulation, computational geometry and robotics literature. We refer the reader to some recent surveys [4, 5, 6]. In this section, we give a brief overview of hierarchical methods, CCD and parallel algorithms.

2.1. BVH for Deformable Models

Bounding volume hierarchies have been widely used for collision detection. Different hierarchies are characterized based on the choice of bounding volumes including spheres[7, 8, 9], axis-aligned bounding boxes (AABBs)[10], oriented bounding boxes (OBBs)[11], or discretely oriented polytopes (k-DOPs) [12]. These hierarchies can be computed in top down or bottom manner. Recently, there has been considerable interest in developing fast techniques to update the hierarchies for deformable models. Many techniques based on refitting and selective restructuring of the hierarchies have been proposed [13, 14, 15].

2.2. Continuous Collision Detection

Many efficient algorithms have been designed for continuous collision detection (CCD) between rigid [16], articulated [17, 18] and deformable models [19, 20, 3]. Most of these approaches linearly interpolate between the vertices of the model and compute the first time of contact based on hierarchical culling and performing elementary tests between the triangle pairs.

2.3. Incremental Collision Detection

Many researchers have exploited spatial and temporal coherence for faster collision detection in interactive applications. The main idea is to perform incremental collision computations between the successive frames. These include incremental methods for convex polytopes [21, 22] and convex hulls [23]. Some hierarchical algorithms maintain a front in the hierarchy and use the front for faster traversal [12, 24, 23, 25]. Most of these techniques have been used to accelerate collision checking between rigid models. An event-based algorithm [26] and kinetic separation lists [27] are proposed for deformable models.

2.4. Parallel Collision Detection and Simulation

Different algorithms have been proposed to use the parallel capabilities of commodity processors to accelerate collision detection. These include GPU-based algorithms that utilized the rasterization capabilities of many-core GPUs for faster interference computations [28, 29]. Some of the recent work includes faster computation of hierarchies using multi-core CPUs [30] or many-core GPUs [31, 32].

A key component in collision checking is traversing the hierarchies. This corresponds to traversing the resulting BVTT (bounding volume traversal tree) [33] in a depth-first manner. Many researchers have designed parallel algorithm to accelerate depth-first hierarchical traversals and these methods can be applied to collision detection. Rao and Kumar [34] showed that the efficiency of parallel depth-first search is strongly influenced by the work distribution scheme and architecture features. The scalability of different load balancing techniques is also analyzed [35]. Reinefeld and Schnecke [36] compared load balancing strategies of depth-first search methods and proposed a scheme that uses fine-grained fixed-sized work packets. Static and dynamic load balancing methods are used in [37] to accelerate collision detection based on communication between the processors.

Assarsson and Stenström [38] demonstrated three times speedup on eight processors for collision detection between CAD models undergoing rigid motions. Gringerg [39] proposed a method to extract parallelism by using static task partition for collision checking on computing clusters.

There have been considerable efforts on developing faster algorithms for physical simulation using multiple cores or distributed architectures. Chen et al. [40] have demonstrated the benefit of using many-core architectures for collision detection and physical simulation for a limited set of models. Faster collision detection algorithms for cloth simulation have been presented for distributed memory architectures [41, 42]. Multi-threaded techniques are used in [43] to accelerate implicit time integration and collision handling for cloth simulation. A self-collision detection based task decomposition has been used in [44] to map the computation to multi-core processors.

3. Notation and Parallel Hierarchical Computation

In this section we introduce our notations and present a simple scheme to parallelize the hierarchical algorithm.

3.1. Notation and Background

Collision detection among geometric models is accelerated using bounding volume hierarchies of the models. Our approach is also based on computing and updating bounding volume hierarchies. After the traversal, we perform overlap tests between the primitives. For the rest of the paper, we assume that the primitives are triangles and we perform continuous tests, i.e., elementary tests between them to check for collisions based on linearly interpolating motion between the vertices of the triangles. We use the following terms in the rest of the paper.

BVHs: Bounding volume hierarchies (BVHs) are used to accelerate collision and proximity queries. We represent the scene containing deformable models using a single BVH. We check for self-collisions and inter-object collisions by starting from the root node of the BVH and traversing the hierarchy. Our traversal algorithm is independent of the underlying bounding volumes.

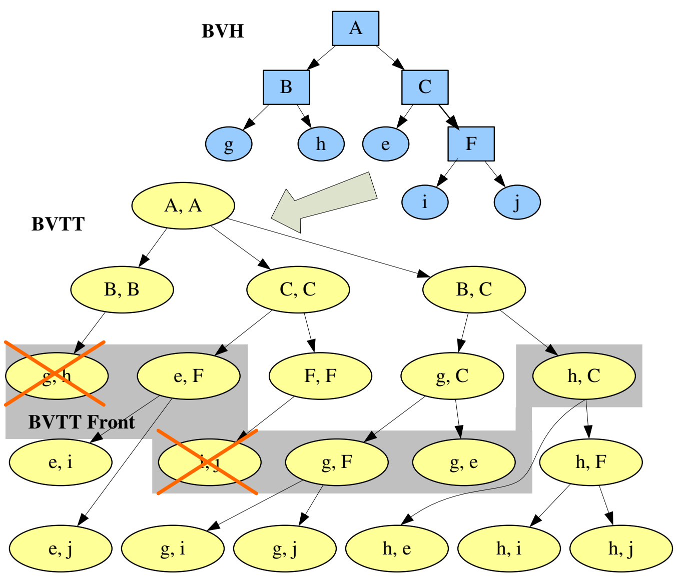

Figure 2: **BVH, BVTT, and BVTT front:** The scene is organized as a single BVH (upper part). The BVTT and a front of BVTT at a given simulation time step are shown in the lower part. The self-collision queries are also performed by this BVTT.

BVTT: A bounding volume test tree (BVTT) represents the hierarchy of overlap tests performed during the traversal. BVTT is a representation which is primarily used for design and analysis of collision detection algorithms [45, 23, 33, 27]. We extend the prior formulation to handle self-collisions and deformable models. Each node in the BVTT represents a single overlap test between a pair of bounding volumes (BVs) or a self-collision test on one of the BVs. As a result, the BVTT is no longer a binary tree, and some internal nodes of our BVTT could have three children nodes.

BVTT Front: A front of the BVTT is a set of internal and leaf nodes where the traversal terminates while performing a collision query during a given time instance. The front reflects how much of the tree is traversed for each instance of the collision query.

An example of a BVH, the corresponding BVTT and the BVTT front are shown in Fig. 2. In this case, the entire scene consisting of deforming models is organized as a single BVH (e.g., the upper part). The traversal during the collision query is represented as a BVTT (the lower part). The nodes where the traversal of the BVTT terminates are stored as a BVTT front (shown in gray).

3.2. CCD Computation

There is extensive literature on fast algorithms for CCD computation between rigid, articulated and deformable models. However, current algorithms are unable to achieve interactive performance (i.e. 20 frames a second or more) on complex models due to following reasons [19, 20, 3]:

- **Elementary tests:** An exact CCD test between two triangles reduces to solving root of univariate polynomial equations. For the simplest case of linear interpolating motion, a CCD test reduces to performing 15 elementary tests between the features (i.e., faces, edges, and vertices): 9 VF tests and 6 EE tests. Each elementary test reduces to solving a cubic equation, and takes about 3 usec on average [28] on a single core. In practice, the CCD test between triangles is about 140X slower than discrete collision detection test which involves overlap tests between edges and faces (about 0.3 usec per triangle pair [46]).

- **Self-collisions:** Since the deformable motion and topological changes can result in self-collisions, the features of adjacent triangles in a mesh need to be tested for collision. In many cases, checking for self-collisions can take 50% – 90% of the total collision query time [28].

- **False positives:** The hierarchical approaches use bounding volumes (e.g. AABBs, K-DOPs, OBBs, etc.) for culling away primitives that are not in close proximity. However, the continuous formulation of the motion combined with self-collisions leads to a very high number of pairwise tests and false positives [47, 20]. Even with tight fitting bounding volumes, such as k-DOPs, current algorithm can result in more than 90% of the overlap tests as false positives [47].

Our goal is to improve the performance by designing a new CCD algorithm that maps well to current multi-core processors. The key to design a good parallel algorithm is to balance the load evenly among the cores. Our approach is based on the serial hierarchical algorithm described in [3]. The algorithm consists of four stages:

- **Updating BVH:** It involves recomputing the bounding volumes of the features, and update the BVs so that they enclose the underlying model hierarchically.

- **BVTT traversal & pair computation:** Traverse the BVTT in a depth-first manner, perform bounding volume tests, and collect potentially colliding non-adjacent triangle pairs.

- **Non-adjacent pair tests:** Perform elementary tests on non-adjacent triangle pairs which pass bounding volume tests.

- **Adjacent pair tests:** Use the orphan set formulation [3] to perform necessary elementary tests between adjacent pairs.

Based on our experiments of several benchmarks of one of the fastest serial implementation, the percentage of the running time spent in each stage are shown in Table 1. These benchmarks are described in Sec. 6.2.

3.3. *Parallel Hierarchical Computation (PHC)*

As shown by Table 1, by using orphan set, the running time spent on adjacent pairs is negligible. Its ratios are below 0.4% for all the benchmarks. An orphan set [3] is a subset of feature

Stage	Princess Fig. 14	Flamenco Fig. 15	N-body Fig. 16	Cloth-ball Fig. 17
BVH Updating	48.15%	14.74%	19.91%	24.96%
BVTT traversal	39.45%	39.4%	37.10%	45.26%
Non-adjacent pairs	12.03%	45.45%	42.96%	29.68%
Adjacent pairs	0.32%	0.35%	0.01%	0.09%

Table 1: **Running time ratio:** This table shows the running time ratio of each stage of a serial CCD algorithm [3] on different benchmarks.

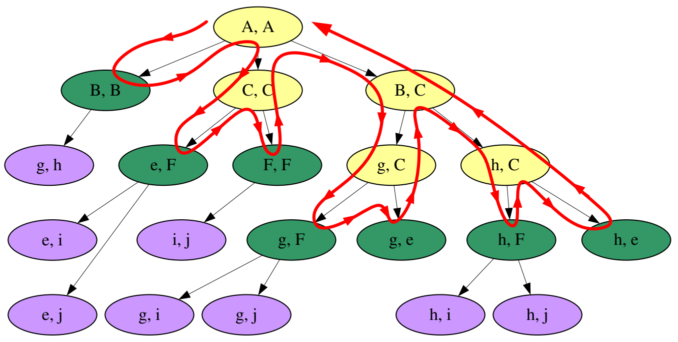

Figure 3: **Parallel CCD with PHC:** PHC collect all the collision query subtasks (green nodes) by breadth-first scanning of BVTT. Then all the sub-tasks will be executed in parallel.

pairs belonging to adjacent triangles which are not been tested during the processing of non-adjacent triangle pairs. By only testing on the orphan set, the number of elementary tests for adjacent triangles can be reduced by 99.9%. The orphan sets are computed by analyzing the connectivity of meshes.

In terms of other three stages, BVH updating and non-adjacent pair processing are relatively simple to parallelize on multiple cores. On the other hand, parallelization of BVTT traversal & pair computation is relatively difficult. As a result, the main challenge for parallel collision detection is to come up with appropriate parallel techniques for BVTT traversal and computing the potentially overlapping pairs.

The simplest parallel algorithms for CCD computations are based on extending the conventional parallelization methods to this problem [41, 42, 43, 44]. The resulting approach is PHC, which stands for parallel hierarchical computation.

In the second stage of the algorithm, i.e., stage BVTT traversal & pair computation in [3], PHC needs to first traverse the BVTT in breadth-first order, collect all the collision query subtasks and put them in a stack. Then all the sub-tasks in the stack are executed in parallel on multiple processors or cores. An example of the performance of PHC is shown in Fig. 3.

However, such an approach (i.e. PHC), consists of a serial part corresponding to breadth-first traversal of BVTT and collecting the sub-tasks. In practice, such a naïve approach has two major bottlenecks:

- **Evaluation of collision query:** Due to the dynamic nature of deforming models, it is hard to estimate or evaluate the computation load of collision query between two BVH nodes. A poor estimate would result in load imbalance and

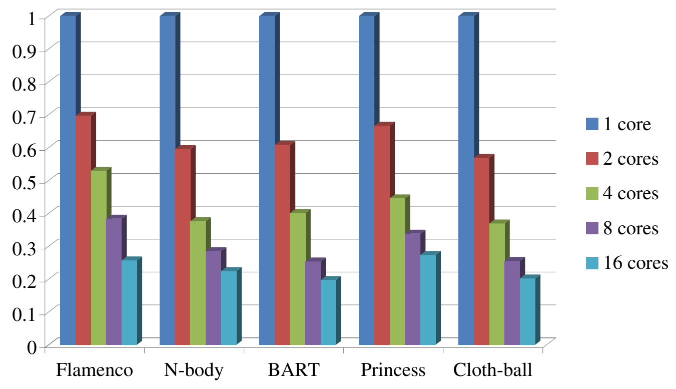

Figure 4: **Speedup of PHC based parallel CCD:** For the benchmarks, the speedups are varying from 3.7X to 5.1X when all the 16 cores are used.

the cost of a collision query can highly vary based on the relative configuration of the primitives.

- **Scalability:** To achieve good parallelization, PHC needs to allocate enough sub-tasks to keep all the cores busy. In such cases, the breadth-first traversal becomes a serial overhead in terms of scaling the computation on a large number of cores.

We have implemented a PHC based parallel CCD algorithm, and observed $3.7X - 5.1X$ speedups for our benchmarks on a 16 core-workstation (Fig. 4). Furthermore, the approach doesn't scale well with the number of cores. As a result, we need an improved hierarchy based algorithm for faster parallel collision detection.

4. Parallel Continuous Collision Detection using FBD

In this section, we present a novel parallel CCD algorithm using front based decomposition (FBD). It is based on a novel front-based algorithm for deformable models, and uses a front to perform improved task decomposition among multiple cores.

4.1. Our Approach

In order to design a parallel algorithm that scales well with the number of core, several characteristics of the current multi-core CPUs need to be taken into account:

- Synchronization overhead: Thread synchronization can be fairly expensive on multi-core processors especially for interactive applications. As a result, our goal is to design an algorithm that has low synchronization overhead.

- Fine-grained task decomposition: A key issue in the design of collision detection algorithms for many-core CPUs is designing an appropriate fine-grained task decomposition scheme. The underlying task decomposition should adapt based on the relative configuration of the primitives and able to balance the load among the different cores.

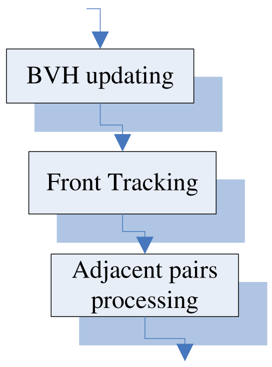

Figure 5: **Our new hierarchical CCD algorithm with three stages:** BVH updating, front tracking, and adjacent pairs processing. Different execution modes are used for each stage respectively according to the underlying computation.

- Cache-friendly memory layouts: The cores on modern multi-core processors often share some of the on-chip caches. This can result in a slow down as multiple threads attempt to access the same memory locations. We use cache-friendly memory layouts of BVH and the BVTT front to improve cache utilization and thus boost runtime performance.

Given these constraints, we design a three-stage parallel collision detection algorithm for deformable models . The three stages are shown in Fig. 5. In order to eliminate the synchronization bottleneck, the two stages in the original serial algorithm, BVTT traversal and non-adjacent pair processing, are merged into a single stage, known as the BVTT front tracking. In this stage, we use FBD to perform fine-grained task decomposition and achieve good scalability.

The pseudo code description of the parallel algorithm performing continuous collision detection at each simulation time step are shown in Alg. 1. At the first stage, BVH updating, we calculate the coordinates of vertices at current time step(Line 3), update BVs of features based on the new coordinates (Line 4), and refitting the BVH using the updated BVs. At the second stage, front tracking, we update the BVTT front incrementally (Line 9), and perform collision query based on the BVTT front (Line 10). Finally, at the third stage, adjacent pairs processing, we handle self-collisions caused by adjacent triangle pairs using orphan set (Line 13).

For all these three stages, different execution modes are used according to the characteristics of their underlying tasks: For BVH updating, the structure of BVH stays unchanged during the simulation. So we used static task partitioning to parallelize it. Dynamic task partitioning is used for BVTT front tracking due to the dynamic nature of deforming models. In our current formulation, we perform adjacent-pair processing in serial because it takes less than 0.4% of the total running time.

A key component of our algorithm is efficient computation of BVTT front for deformable models. We use two techniques, orphan-set based front reduction and adaptive front rebuilding, to compute the BVTT front at each frame of the simulation. Finally, we use cache-friendly memory layouts for the BVH and BVTT front. The layout of BVH is compute only during the

Algorithm 1 Parallel CCD: Parallel continuous collision detection at current simulation time step t_i.
Input: vertex coordinates V_{i-1}, BVH B_{i-1}, and BVTT front L_{i-1} at previous time step, orphan set O.
Output: vertex coordinates V_i, BVH B_i, BVTT front L_i at current time step, collision information.

```
// Stage 1: BVH Updating, executing in parallel by
// static task partitioning.
CalcInterpolatedVertices($t_i$, $V_i$)
UpdateFeatureBoundingVolumes($V_i$, $V_{i-1}$)
RefitBVH($B_i$, $B_{i-1}$)

// Stage 2: Front Tracking, executing in parallel by
// dynamic task partitioning.
FrontTracking($L_i$, $L_{i-1}$, $B_i$)
CollisionQuery($L_i$)

// Stage 3: Adjacent pairs processing, executing in serial.
DoOrphanSet($O$)
```

initial building stage and is fixed for the rest of the simulation, while the layouts of BVTT fronts are computed dynamically at each frame of the simulation.

4.2. *BVTT for Deformable Models*

Hierarchical collision detection is performed by traversing the BVTT. The BVTT nodes where the traversal terminates are represented by a BVTT front. The BVTT nodes in prior work are defined as collision nodes: i.e., node $\{k, j\}$, where k and j are different BVH nodes for the two models been tested [45, 23, 33, 27]. So the BVTT formulation is a binary tree, and it can only be used to detect inter-object collisions.

In order to efficiently process intra-object collisions as well, we extend the concept of BVTT by introducing self-collision nodes and therefore, some nodes thereby can have three children:

Self-collision nodes of BVTT: A self-collision node $\{n, n\}$ of a BVTT correspond to the self-collision queries for a particular node n in a BVH B. If n is not a leaf node of B, $\{n, n\}$ in the BVTT will have three children: a self-collision node $\{n \rightarrow left, n \rightarrow left\}$, a self-collision node $\{n \rightarrow right, n \rightarrow right\}$, and a collision node $\{n \rightarrow left, n \rightarrow right\}$.

As a result, the BVTT for deformable models is no longer a binary tree. It becomes a special case of 2-3 tree, i.e., the BVTT is composed of collision nodes and self-collision nodes, a collision node has two children, while a self-collision node has three children.

4.3. *Front based Decomposition (FBD)*

By using the BVTT for deformable models, we can handle both inter-object and intra-object collisions in a unified manner. As shown in Fig. 6, we use front tracking to perform the collision queries in an efficient manner by utilizing temporal coherence. Specifically, we track the front between successive frames and use incremental computations to update it.

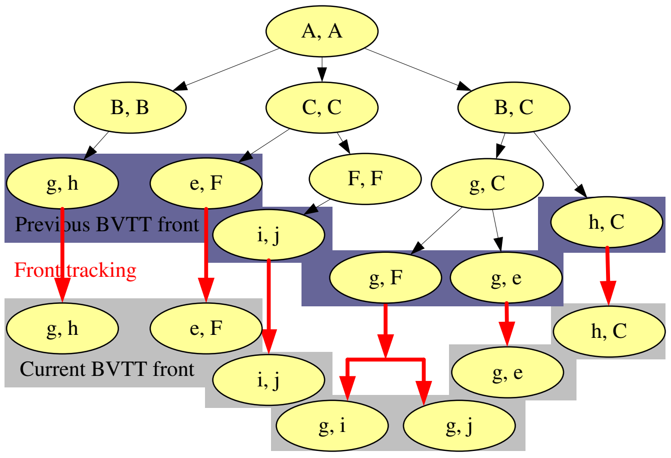

Figure 6: **Front based decomposition:** Since nodes of previous BVTT front can be updated independently, current BVTT front can be tracked fully in parallel.

As the objects deform or move during the simulation, the BVTT front needs to be updated correspondingly. In order to maintain a valid front, two operator "sprouting" and "pruning" are used [24, 23, 25]. These two operators correspond to the expanding and shrinking of BVTT front respectively. By using sprouting operator, a front is sprout to the level where bounding volumes are disjoint or leaf nodes in the tree are reached. Pruning the two sibling BVTT nodes whose parent is disjoint recursively generates more compact front as well as reduced computation load.

The sprouting of each front node is independent of each other. This make it suitable for parallel processing. The pruning operator need to search for sibling nodes in the BVTT front, is hard for parallel execution.

In practice, we do not use any pruning operator since it will affect the parallel performance. Instead, we use the sprouting operator to track the BVTT front incrementally, and a adaptive rebuilding strategy is used to compute more compact BVTT front.

During the tracking of the BVTT front, the nodes that need to be sprouted are flagged as invalid, and their descendants will be traversed in a depth-first manner to check for collisions. The nodes where traversal terminate will be inserted into the current BVTT front.

Due to the temporal coherence in a simulation, the computational load of updating each node is approximately balanced. Therefore, FBD is a good fine-grained task decomposition method. It overcomes the drawbacks of PHC, and can provide potentially scalable performance.

4.4. *Parallel Collision Query*

A collision query is preformed by scanning current BVTT front. For a BVTT node $\{N_a^i, N_b^i\}$ in the BVTT front, if both N_a^i and N_b^i are leaf nodes of the BVH, it is further checked for collision by performing exact elementary tests (Alg. 2).

Based on front tracking, the collisions are detected from the nodes that are contained in the front. In this case, we don't need to traverse the entire hierarchy from the BVTT's root node.

Algorithm 2 CollisionQuery(L): Collision query by scanning current BVTT front L.

```
1: for node {$N_a^i$, $N_b^i$} in $L$ do
2:     if IsLeaf($N_a^i$) AND IsLeaf($N_b^i$) then
3:         // Perform exact elementary tests.
4:         ElementaryTest($N_a^i$, $N_b^i$)
5:     end if
6: end for
```

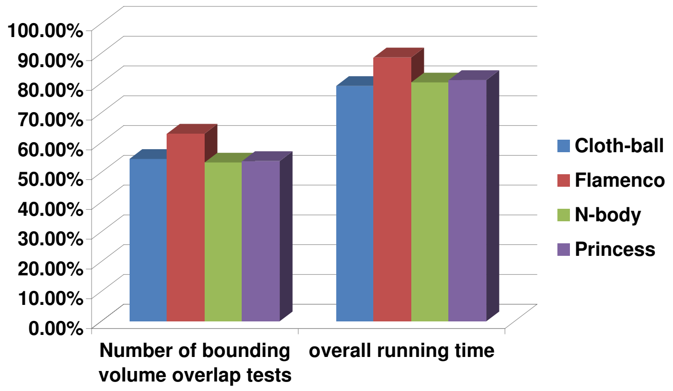

Figure 7: **Benefits of front tracking:** According to the experiments on a serial of benchmarks, the number of bounding volume test is reduced by 37% – 47%, and overall running time is reduced to 12% – 21% compared to [3] by front tracking.

This reduces the runtime overhead. In our benchmarks, we observed that the number of bounding volume tests can reduce by 37% – 47%, and overall running time decreases by 12% – 21% in a serial implantation, as shown in Fig. 7.

Another benefit of front tracking is that it provides a good approach to perform fine-grained task decomposition of the collision query over multiple cores. Since the nodes of a BVTT front are updated independently, it is easy to execute them in parallel on multiple cores.

In order to achieve high throughput, we need to fully utilize the computational power of all the cores. By using FDB, the task of collision query is broken into a set of front node updating sub-tasks which can be executed independently. As shown in Fig. 8, all the nodes of the BVTT front are organized as blocks and manage the threads using dynamic load balancing. Initially the blocks are send to the idle threads. When a thread finishes the computation on its block, a new block is assigned to the thread and thereby keeps the thread busy.

4.5. Adaptive Front Rebuilding

The initial BVTT front is built by traversing the BVTT in a top-down manner. The pseudo code description of the algorithms are give in Alg. 3 and Alg. 4. The building algorithm starts by checking for self-collision from the root node of the BVH (Alg. 3). During the traversal of the BVTT, all leaf node pairs containing triangle pairs are inserted into the front (Line 3 of Alg. 4). In terms of internal nodes of the BVTT, if their bounding volumes do not overlap, those node pairs are inserted into the front (Line 9 of Alg. 4). The front building algorithm proceeds recursively until the traversal is terminated.

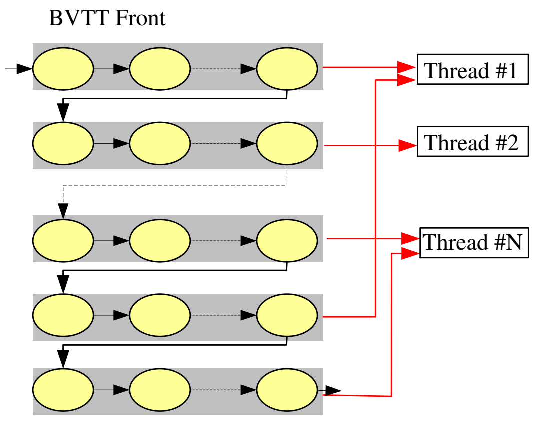

Figure 8: **Parallelized front tracking:** With FBD, a collision query is decomposed into a set of fine-grained tasks. Nodes of the BVTT front are organized into groups, and handled by different threads using dynamic load balancing.

Algorithm 3 FrontBuild(Node N^i): Building the front while checking self-collisions at a BVH node N^i.

```
1: if IsLeaf($N^i$) then
2:     return // Skip leaf nodes.
3: end if
4: // Check the descendants.
5: FrontBuild($N^i$ →left)
6: FrontBuild($N^i$ →right)
7: FrontBuild($N^i$ →left, $N^i$ →right)
```

After updating the BVTT front during a sequence of simulation time steps, the front contains many phantom nodes. These nodes that are flagged as invalid by sprouting operators and ideally these nodes should be removed by pruning operators. Although the collision query results are not affected by these phantom nodes, these nodes can consume high memory and result in unnecessary computations.

Our algorithm tends to estimate when does the BVTT front has a high number of phantom nodes. We use the following analysis to evaluate the quality of the BVTT front.

The process of front updating can be formulated as:

$$L_{i+1} = L_i + S_{i+1} - B_{i+1} \tag{1}$$

where:

- L_{i+1} and L_i are the BVTT fronts at current and previous simulation time steps respectively;
- S_{i+1} are the nodes inserted by the sprouting operator;
- B_{i+1} are the nodes flagged as invalid by the sprouting operator.

Let us assume that the front L_i updated for k simulation time steps. From Equation 1, the L_{i+k} can be expressed by:

$$L_{i+k} - L_i = \sum_{l=1}^{k} S_{i+l} - \sum_{l=1}^{k} B_{i+1} \tag{2}$$

Algorithm 4 FrontBuild(Node N^i_a, N^i_b): Building the front while checking for collisions between two BVH nodes N^i_a and N^i_b.

```
if IsLeaf($N^i_a$) AND IsLeaf($N^i_b$) then
    if $N^i_a$ not adjacent to $N^i_b$ then
        FrontAdd($N^i_a, N^i_b$) // Record into front.
        return
    end if
end if

if BoundingBoxTest($N^i_a, N^i_b$) == NoOverlap then
    FrontAdd($N^i_a, N^i_b$) // Record into front.
    return
end if


if IsLeaf($N^i_a$ ) then
    FrontBuild($N^i_a$, $N^i_b$ →left)
    FrontBuild($N^i_a$, $N^i_b$ →right)
else
    FrontBuild($N^i_a$ →left, $N^i_b$)
    FrontBuild($N^i_a$ →right, $N^i_b$)
end if
```

We evaluate the quality of the BVTT front using the following metrics:

$$Q_1(L_{i+k}) = \frac{\| \sum_{l=1}^{k} S_{i+l} \|}{\|L_{i+k}\|} \tag{3}$$

$$Q_2(L_{i+k}) = \frac{\| \sum_{l=1}^{k} B_{i+1} \|}{\|L_{i+k}\|} \tag{4}$$

where:

- $\| \sum_{l=1}^{k} B_{i+1} \|$ is the number of nodes flagged as invalid during the k simulation time steps;
- $\| \sum_{l=1}^{k} S_{i+l} \|$ is the number of nodes inserted during k simulation time steps;
- $\|L_{i+k}\|$ is the total number of front nodes.

When $Q_1(L_{i+k})$ or $Q_2(L_{i+k})$ are greater than specified thresholds, e.g., 30% and 45% respectively in our implementation, our algorithm rebuilds the BVTT front as opposed to updating it.

4.6. Orphan-set based Front Reduction

All the adjacent triangle pairs can not be culled by bounding volume tests, and they bring a large number of elementary tests.

The BVTT front L can be represented as:

$$L = A + B_a + B_{na} \tag{5}$$

where:

- A is the set of internal pairs $\{N_a, M_a\}$: N_a or M_a is not a leaf node of the BVH;
- B_a is the set of adjacent leaf pairs $\{N_b, M_b\}$: N_b and M_b are leaf nodes of the BVH, and N_b is adjacent to M_b.
- B_{na} is the set of non-adjacent leaf pairs $\{N_b, M_b\}$: N_b and M_b are leaf nodes of the BVH, and N_b is not adjacent to M_b.

All the adjacent leaf pairs that cannot be culled by bounding box overlap tests, so B_a tends to be a large subset of L. In our benchmarks, the ratio of the size of B_a to the size of L is typically between 20% to 40%.

Based on the property of the orphan set, following theorem is used to effectively cut down the size of BVTT front:

Orphan-set based Front Reduction Lemma : *By inserting only the non-adjacent pairs B_{na} into a reduced colliding front $\hat{L}$ and processing the orphan set separately, the BVTT front L in Equation 5 can be reduced to:*

$$\hat{L} = A + B_{na} + OF \tag{6}$$

where:

- *A and B_{na} are the internal pairs and non-adjacent leaf pairs of L respectively;*
- *OF is the orphan set of the models.*

Proof. All the nodes of B_a correspond to adjacent pairs. The orphan set OF includes all the elementary tests between adjacent pairs. So:

$$OF > B_a \tag{7}$$

By substituting this relation in Equation 5, we get:

$$\hat{L} = A + B_{na} + OF > L = A + B_a + B_{na} \tag{8}$$

As a result, all the collisions can be computed by only performing overlap tests on the reduced BVTT front $\hat{L}$, and it doesn't affect the accuracy of the algorithm. □

By not considering the node pairs that correspond to adjacent triangle pairs (B_a), we reduce the memory overhead of BVTT front, and lessen computation load. The size of BVTT front can shrink by about $20\% - 40\%$ of the original length, as shown in Fig. 9. This results in improved performance for both the serial as well as the parallel CCD computation algorithm. Specifically, we observed up to 17% improvement in the overall performance of our parallel algorithm.

4.7. Parallel BVH Updating

The entire scene is organized as a single BVH. We use the refitting algorithm to update the BVH as the objects undergo non-rigid motion. We use the following equation to evaluate the cost of refitting an internal node of the BVH:

$$T(I_u) = N_u * C_u \tag{9}$$

where:

- I_u is an internal node of the BVH;
- N_u is the number of nodes in the subtree rooted at I_u;

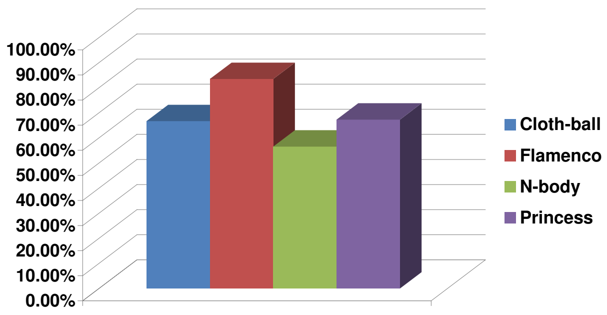

Figure 9: **Orphan-set based front reduction:** By removing all the nodes containing adjacent triangle pairs from the colliding front, the length of the colliding front shrinks by about 20% – 40% of the original length. We observed up to 17% improvement in the overall performance of our parallel algorithm.

- C_u is cost of recalculating the bounding volume for a single node. It is a constant value for all the nodes.

The values of N_u and C_u depend on the type of bounding volumes (BVs). For simple BVs, such as spheres or AABBs, C_u tends to be low. For more complex BVs, such as, OBBs or k-DOPs, the C_u is relatively more expensive.

Since the structure of the BVH is fixed during the various frames, N_u remains constant. Therefore, we perform static task decomposition at the preprocessing stage, and distribute the refitting computation evenly among different cores [48, 44].

4.8. *Cache-friendly BVTT Layouts*

High speed caches are used to cover the performance gap between the processor and memory. Multi-core processors enable an application to run multiple threads simultaneously. When large amount of data is processed, these threads have to compete for memory bandwidth and shared caches. As the number of threads increase, cache-accesses can become more expensive and can become a efficiency bottleneck [49]. Cache-efficient layouts of BVHs [50] have been proposed to get improved performance of collision detection between massive models. But there is no former work on the cache-competition problem in multi-core environment during the process of collision detection.

In our FBD based parallel CCD algorithm, BVHs and BVTT fronts are the primary data structures that are frequently accessed by multiple threads. We compute the memory layout and improve cache utilization based on following observations: The BVHs are frequently visited during the updating of BVTT fronts. For a node $\{N_a^i, N_a^j\}$ of current BVTT front, and sub-trees rooted at N_a^i and N_a^j are further traversed on spouting operator. When the front is updating in parallel by multiple threads, a good memory layout of BVHs and BVTT fronts is needed to fully utilize the high-speed cache by minimizing data exchange between cache and memory.

We store the BVH in depth-first order as a linear array (Fig. 10), and store the nodes of the BVTT front in an ordered list (Fig. 11). By rearranging the front nodes in this manner, the data locality is improved when a block of nodes is processed by the same core.

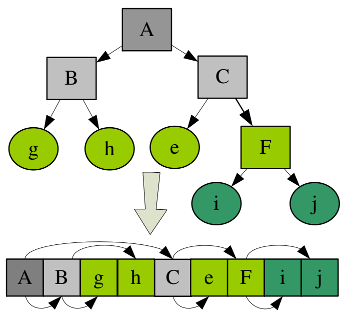

Figure 10: **Memory layout of the BVH:** The BVH in Fig. 2 is stored in depth-first order as a linear array.

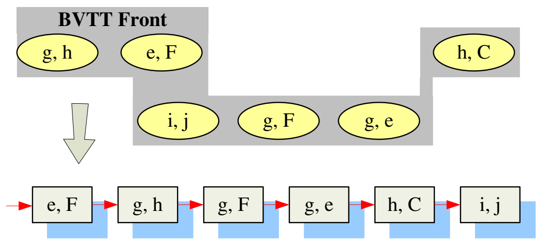

Figure 11: **Memory layout of the BVTT front:** The nodes of the BVTT front in Fig. 2 are stored in the order of its belonging BVH nodes.

As compared to a parallel CCD algorithm with a pointer-based BVH and unsorted BVTT fronts, these memory layouts further improves the performance by 7% – 14% without any other modification to the runtime code on a 16-core machine running 16 threads.

5. Analysis

We have presented two parallel CCD algorithms. These include the basic PHC algorithm in Sec. 3 and the FBD algorithm in Sec. 4. In this section, we analyze the scalability of these algorithms and show that FBD offers improved theoretical and empirical performance.

For the serial algorithm, its computation load T_s can be formulated as:

$$T_s = C_t + C_b + C_e \tag{10}$$

where:

- C_t is the cost of BVTT traversal;
- C_b is the cost of bounding volume overlap tests;
- C_e is the cost of elementary tests between the primitives.

The first task in PHC algorithm is traversal and decomposition of the problem into sub-tasks, and executing each sub-task in parallel. In this case, C_t and C_b are represented as: $C_t = \hat{C}_t + \check{C}_t$ and $C_b = \hat{C}_b + \check{C}_b$, where:

- $\hat{C}_t$ and $\hat{C}_b$ are the costs of BVTT traversal and bounding volume overlap test, respectively, in the first part correspond to traversal and collecting the sub-tasks;

- $\breve{C}_t$ and $\breve{C}_b$ are the costs of BVTT traversal and bounding volume overlap tests within all the sub-tasks;

The computation cost of PHC based algorithm is:

$$T_{PHC} = \hat{C}_t + \hat{C}_b + \frac{\breve{C}_t + \breve{C}_b + C_e}{N} \tag{11}$$

where N is the number of core[1].

By using FBD, the first step of traversing the BVTT from the root node is eliminated. Since all the nodes of the BVTT front are tracked in parallel, the computation load of FBD based algorithm can be formulated as:

$$T_{FBD} = \frac{S * C_b + C_e}{N} \tag{12}$$

where S is the ratio of reduced bounding volume overlap tests. In practices, S is always a constant less than 1 and its actual value depends on the benchmark. In our benchmarks, S is between 53% to 63%.

Computation Overhead of PHC and FBD Theorem: *Computation overhead of PHC (T_{PHC}) is always greater than of FBD (T_{FBD}), and T_{PHC} is directly proportional to T_{FBD} by a factor of N, i.e., $\frac{T_{PHC}}{T_{FBD}} = O(N)$, where N is the number of core.*

Proof. The computation load of PHC based and FBD based parallel CCD algorithm can be expressed as:

$$\begin{aligned}
\frac{T_{PHC}}{T_{FBD}} &= \frac{N * (\hat{C}_t + \hat{C}_b) + \breve{C}_t + \breve{C}_b + C_e}{S * C_b + C_e} \\
&= \frac{(N-1) * (\hat{C}_t + \hat{C}_b) + C_t + C_b + C_e}{S * C_b + C_e} \\
&= \frac{(N-1) * (\hat{C}_t + \hat{C}_b) + C_t + (1-S) * C_b}{S * C_b + C_e} + 1 \\
&= (N-1) * K_1 + K_2 + 1 \qquad (13)
\end{aligned}$$

where K_1 and K_2 are two constants depending on the benchmark:

$$\begin{aligned}
K_1 &= \frac{\hat{C}_t + \hat{C}_b}{S * C_b + C_e} \\
K_2 &= \frac{C_t + (1-S) * C_b}{S * C_b + C_e}
\end{aligned}$$

Equation 13 shows that the computation overhead of PHC always greater than of FBD ($\frac{T_{PHC}}{T_{FBD}} > 1$). And T_{PHC} is directly proportional to T_{FBD} by a factor of N ($\frac{T_{PHC}}{T_{FBD}} = O(N)$). □

In practice, we get results shown in Fig. 12. By using FBD, the running time of PHC is reduced by 40% with 8 cores, and reduced by at most 48% with 16 cores. As a result, PHC doesn't scale well with the number of core.

From Equation 11 and Equation 12, we can see the difference between their performance as the number of core increase: The speedup of PHC based algorithm is limited by the cost of task-collection and traversal, i.e., $\hat{C}_t$ and $\hat{C}_b$, while FBD based algorithm has better scalability by the number of core.

[1]The analysis assumes that there is no overhead for parallel execution with multiple cores.

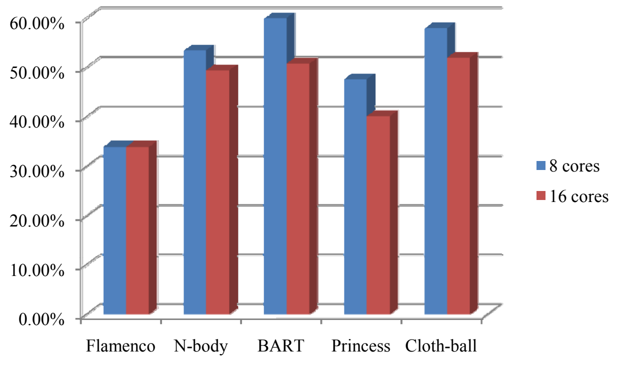

Figure 12: **Performance comparison between PHC and FBD:** The running time are shown as the comparison ratio between FBD and PHC. By using FBD, the running time of PHC is reduce by 40% with 8 cores, and reduce by 48% with 16 cores.

6. Implementation and Results

In this section, we describe our implementation and highlight the performance of our algorithm on multiple benchmarks.

6.1. Performance

We have implemented our algorithm on a Intel Xeon PC with four X7350 processors(quad-core CPU at 2.93 GHz), i.e., a total of 16 cores, and 16 GB of RAM using Visual Studio 2005. OpenMP is used as multi-threaded programming API. We use k-DOPs (specifically 18-DOPs) as bounding volumes for better culling efficiency comparing to AABBs or spheres. The BVHs are built in a top-down manner by recursive longest-axis spatial-median splitting.

6.2. Benchmarks

Five different benchmarks, which arise from different type of simulations and have varying characteristics, are used to measure the performance of our algorithm. These include:

- **Princess** (40K triangles, Fig. 14): A dancer with flowing skirt sits on the floor, resulting in many inter- and intra-object collisions.

- **Flamenco** (49K triangles, Fig. 15): A fiery flamenco dancer wearing colorful skirt with ruffles. This benchmark has a high number of self-collisions.

- **N-body** (34K triangles, Fig. 16): A scene with hundreds of spheres and cones that are colliding with each other.

- **Cloth-ball** (92K triangles, Fig. 17): A piece of cloth drops on top of a ball and curls around resulting in a high number of self-collisions.

- **BART** (4K triangles, Fig. 18): A set of triangles under mostly unstructured, random movement. Since it has high depth complexity and overlapping primitives, this scene is one of the worst cases for collision query as well as hierarchy updates. It is part of the BART animated ray tracing benchmark from [51].

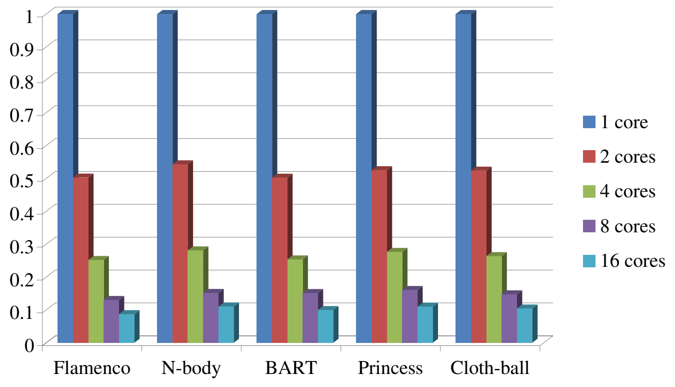

Figure 13: **Speedups of FBD based parallel CCD:** Almost linear accelerations are achieved when the number of core increase. The speedups are varying from 10.1X to 13X when all the 16 cores are used.

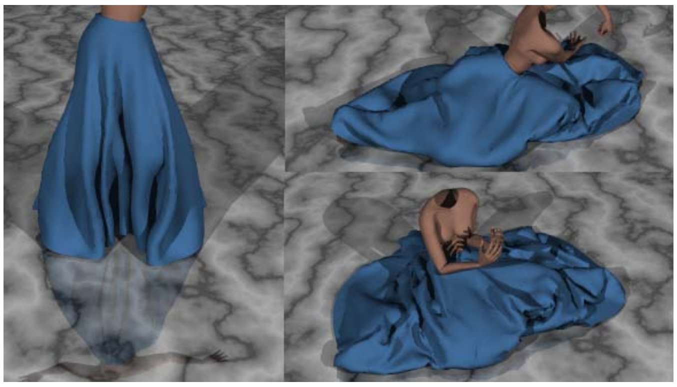

Figure 14: **Speedups on the Princess benchmark:** The speedups are 6.4X using 8 cores and 10.2X on the 16 cores. It takes 5.3ms per frame to compute all the collisions by using all the 16 cores.

All the benchmarks have multiple simulation steps. We perform CCD between discrete steps by using linearly interpolation motion between the vertices of the triangles and compute the first time-of-contact along the resulting trajectory.

6.3. FBD Performance

We observe considerable speedups in all our benchmarks by using multiple cores. Fig. 13 show the speedups of all the 5 benchmarks as the number of core vary. By fully utilizing all the 16 cores, the speedups of overall running time are varying from 10.1X to 13X. When 8 cores are used, the speedups are between 6.4X to 7.7X. In practice, almost linear speedups are achieved when the number of core increase. As the number of core increases to 16, the overhead of parallel execution, including thread synchronization, scheduling, cache competition, etc., increases and thereby results in a slightly sub-linear speedup. Fig.14 - Fig.18 show the detailed performance for each benchmark.

7. Comparison and Limitations

In this section, we compare our approach with prior techniques and highlight some of its limitations.

Figure 15: **Speedups on the Flamenco benchmark:** The speedups are 7.7X using 8 cores and 13X on the 16 cores. It takes 27ms per frame to compute all the collisions by using all the 16 cores.

Scenes	Query time of Our algorithm	Query time of [3]	Query time of [29]
Princess	5.3	55	500
Flamenco	27	350	N/A
N-body	11.6	117	N/A
Cloth-ball	32.5	340	800
BART	20.4	212	N/A

Table 2: **Performance and Speedup:** This table shows the average query time (ms) of our method, the serial implementation of [3], and the GPU-based technique of [29].

7.1. Comparison

By updating BVTT front incrementally and in parallel, our algorithm achieves 6.4X−7.7X speedups on 8 cores and 10.1X−13X speedups on 16 cores comparing to one of the fastest serial CCD algorithm reported [3]. Our former PHC based work [2] only achieves 4X − 6X speedups on 8 cores.

Some GPU algorithms has been design to improve the performance of CCD by performing occlusion queries [28] or computing 3D distance fields [29] on graphics hardware. In these algorithms, only some computing-intensive parts are executed in parallel on GPUs, and the performance is governed by the readbacks and occlusion queries, which don't scale well.

Our parallel algorithm shows better scalability by fully parallelizing all the stages of CCD, and appears to have a lower overhead by avoiding transferring data between GPUs and CPUs. Table 2 shows the average CCD time of our algorithm (when all 16 cores are used), the serial implementation [3], and GPU-based technique [29].

The notion of incrementally maintaining the colliding front has been used by many authors [45, 23, 12, 33, 27]. Most of the earlier algorithms are limited to handle rigid models and we extend it to self-collisions of deformable models. Our approach to handle self-collisions between adjacent pairs is different from that of [27]. We integrate self-collision detection into

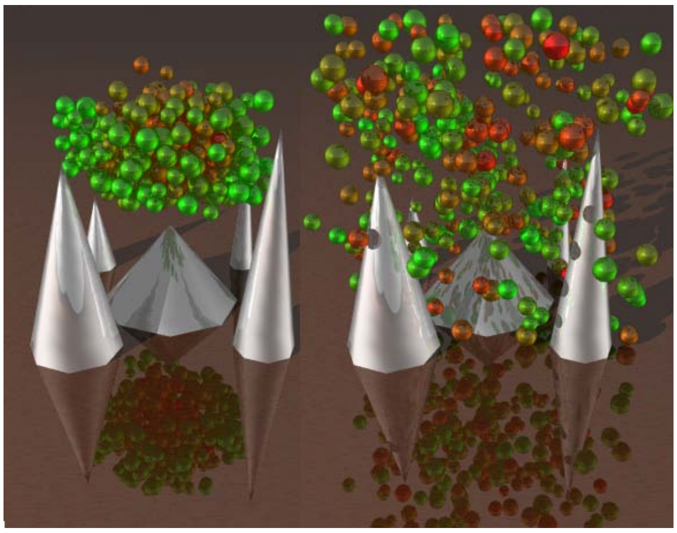
Figure 16: **Speedups on the N-body benchmark:** The speedups are 6.6X using 8 cores and 10.1X on the 16 cores. It takes 11.6ms per frame to compute all the collisions by using all the 16 cores.

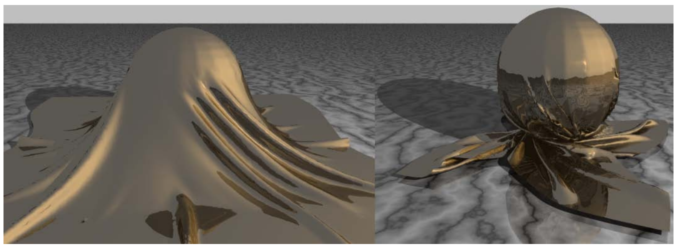
Figure 17: **Speedup on the Cloth-ball benchmark:** The speedups are 6.68X using 8 cores and 10.5X on the 16 cores. It takes 32.5ms per frame to compute all the collisions by using all the 16 cores.

the BVTT, perform adaptive rebuilding of the front, and significantly reduce the size of the BVTT front using orphan sets. As a result, our approach achieves considerable performance improvement over prior algorithms designed for cloth simulation or deformable collisions [41, 43, 42].

In terms of comparison to prior parallel collision detection algorithms [39, 40, 43, 44], our algorithm uses a different task decomposition approach. Specifically, we parallelize updating of the BVTT front. There are two major benefits of using such a front formulation. On one hand, the running time of bounding volume overlapping tests and BVH traversal is considerably reduced for deformable models. Secondly, it provides a good fine-grained task decomposition strategy for parallelization which is useful for parallel execution on modern multi-core/many-core processors.

7.2. Limitations

Our approach has many limitations. Firstly, the front formulation increases the memory overhead. Even for the BART benchmark with 4K triangles, it takes about 17MB to store the colliding front between successive frames. The length of the BVTT front can grow considerably if there are a high number of close proximities. Secondly, the speedup obtained by our

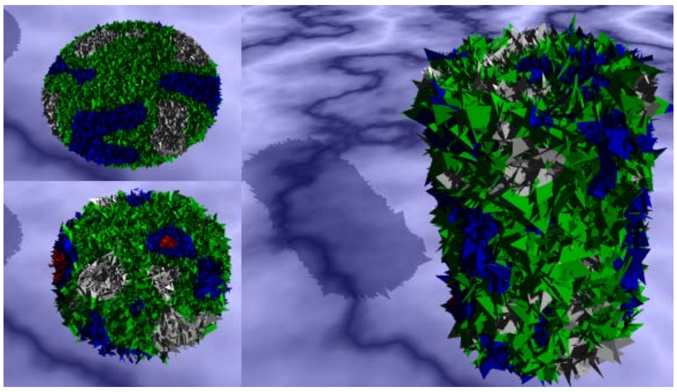
Figure 18: **Speedup on the BART benchmark:** The speedups are 6.6X using 8 cores and 10.4X on the 16 cores. It takes 20.4ms per frame to compute all the collisions by using all the 16 cores.

approach increases sub-linearly as the core number is above 10. When the core number is growing high, the overhead of parallel execution emerges in our current benchmarks and offsets most of the benefit of parallel execution. The overhead results in sub-linear improvement with the number of core. However, we expect almost linear speedups on more complex models with higher number of triangles.

8. Conclusion and Future Work

In this paper, we present a parallel CCD algorithm for deformable models. Our formulation maintains the colliding front which takes into account inter-object collisions as well as self-collision by performing adaptive rebuilding. Furthermore, we combine it with the orphan set formulation to reduce the number of elementary test. In practice, the FBD algorithm can results in considerable improvement in the performance of collision detection algorithm using multiple cores. We parallelize the algorithm over multiple cores. Our algorithm achieves quasi-linear acceleration on a number of benchmarks and we observe $6.4X - 7.7X$ speedups in the overall running times on 8 cores, and $10.1X - 13X$ speedups on 16 cores.

There are many avenues for future work. We would like to improve the performance and try to achieve close to linear speedups. This may involve strategies to overcome the parallelization penalty when the number of core increases. Secondly, we would like to use our algorithm for other applications that require real-time performance such as haptic rendering. Finally, we would like to extend this approach to other proximity queries, including distance computation.

8.1. Parallelization on GPU architectures

Our current algorithm is primarily designed for multi-core CPUs. It would be interesting to extend this approach to many-core GPUs. However, GPUs have smaller caches and they tend to hide the latency by using larger number of threads. As a result, we may have to modify the algorithm and take these features into account.

References

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

[2] M. Tang, D. Manocha, R. Tong, Multi-Core collision detection between deformable models, Tech. rep., Accepted by 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling. Department of Computer Science, UNC Chapel Hill (2009).

[3] M. Tang, S. Curtis, S.-E. Yoon, D. Manocha, ICCD: Interactive continuous collision detection between deformable models using connectivity-based culling, IEEE Transactions on Visualization and Computer Graphics 15 (4) (2009) 544–557.

[4] C. Ericson, Real-Time Collision Detection, Morgan Kaufmann, 2004.

[5] M. Lin, D. Manocha, Collision and proximity queries, in: Handbook of Discrete and Computational Geometry, 2003.

[6] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, W. Strasser, Collision detection for deformable objects, in: Eurographics State-of-the-Art Report (EG-STAR), Eurographics Association, Eurographics Association, 2004, pp. 119–139.

[7] P. M. Hubbard, Interactive collision detection, in: Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality, 1993.

[8] I. J. Palmer, R. L. Grimsdale, Collision detection for animation using sphere-trees, Computer Graphics Forum 14 (2) (1995) 105–116.

[9] G. Bradshaw, C. O'Sullivan, Adaptive medial-axis approximation for sphere-tree construction, ACM Trans. on Graphics 23 (1) (2004) 1–26.

[10] G. van den Bergen, Efficient collision detection of complex deformable models using AABB trees, Journal of Graphics Tools 2 (4) (1997) 1–14.

[11] S. Gottschalk, M. C. Lin, D. Manocha, Obbtree: a hierarchical structure for rapid interference detection, in: SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, 1996, pp. 171–180.

[12] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, K. Zikan, Efficient collision detection using bounding volume hierarchies of k-dops, IEEE Trans. on Visualization and Computer Graphics 4 (1) (1998) 21–37.

[13] C. Lauterbach, S. Yoon, D. Tuft, D. Manocha, RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs, IEEE Symposium on Interactive Ray Tracing (2006) 39–46.

[14] S. Yoon, S. Curtis, D. Manocha, Ray tracing dynamic scenes using selective restructuring, Proc. of Eurographics Symposium on Rendering.

[15] I. Wald, On fast Construction of SAH based Bounding Volume Hierarchies, in: Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing, 2007.

[16] S. Redon, A. Kheddar, S. Coquillart, Fast continuous collision detection between rigid bodies, Proc. of Eurographics (Computer Graphics Forum) 21 (3) (2002) 279–288.

[17] S. Redon, Y. J. Kim, M. C. Lin, D. Manocha, Fast continuous collision detection for articulated models, in: Proceedings of ACM Symposium on Solid Modeling and Applications, 2004, pp. 145–156.

[18] X. Zhang, S. Redon, M. Lee, Y. J. Kim, Continuous collision detection for articulated models using taylor models and temporal culling, ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26 (3) (2007) 15.

[19] M. Hutter, A. Fuhrmann, Optimized continuous collision detection for deformable triangle meshes, in: Proc. WSCG '07, 2007, pp. 25–32.

[20] S. Curtis, R. Tamstorf, D. Manocha, Fast collision detection for deformable models using representative-triangles, in: SI3D '08: Proceedings of the 2008 Symposium on Interactive 3D graphics and games, 2008, pp. 61–69.

[21] M. Lin, J. Canny, A fast algorithm for incremental distance calculation, Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on 2 (1991) 1008–1014.

[22] S. Cameron, Enhancing GJK: Computing minimum and penetration distance between convex polyhedra, IEEE International Conference on Robotics and Automation (1997) 3112–3117.

[23] S. A. Ehmann, M. C. Lin, Accurate and fast proximity queries between polyhedra using convex surface decomposition, in: of Eurographics2001, 2001, pp. 500–510.

[24] T.-Y. Li, J.-S. Chen, Incremental 3d collision detection with hierarchical data structures, in: VRST '98: Proceedings of the ACM symposium on Virtual reality software and technology, ACM, New York, NY, USA, 1998, pp. 139–144.

[25] O. Tropp, A. Tal, I. Shimshoni, D. P. Dobkin, Temporal coherence in bounding volume hierarchies for collision detection, International Journal of Shape Modeling 12 (2) (2006) 159–178.

[26] G. Zachmann, R. Weller, Kinetic bounding volume hierarchies for deforming objects, in: ACM Int'l Conf. on Virtual Reality Continuum and its Applications, 2006.

[27] R. Weller, G. Zachmann, Kinetic separation lists for continuous collision detection of deformable objects, in: Third Workshop in Virtual Reality Interactions and Physical Simulation (Vriphys), Madrid, Spain, 2006.

[28] N. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. Lin, D. Manocha, Interactive collision detection between deformable models using chromatic decomposition, ACM Trans. on Graphics (Proc. of ACM SIGGRAPH) 24 (3) (2005) 991–999.

[29] A. Sud, N. Govindaraju, R. Gayle, I. Kabul, D. Manocha, Fast proximity computation among deformable models using discrete voronoi diagrams, Proc. of ACM SIGGRAPH (2006) 1144–1153.

[30] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, P. Shirley, State of the art in ray tracing animated scenes, in: D. Schmalstieg, J. Bittner (Eds.), STAR Proceedings of Eurographics 2007, The Eurographics Association, 2007, pp. 89–116.

[31] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time KD-tree construction on graphics hardware, in: Proceedings of ACM SIGGRAPH Asia 2008, ACM, New York, NY, USA, 2008, pp. 1–11.

[32] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH construction on GPUs, Proc. of Eurographics (Computer Graphics Forum) 28 (2) (2009) 375–384.

[33] M. A. Otaduy, M. C. Lin, CLODs: Dual hierarchies for multiresolution collision detection, Eurographics Symposium on Geometry Processing (2003) 94–101.

[34] V. N. Rao, V. Kumar, Parallel depth-first search, part i: Implementation, International Journal of Parallel Programming 16 (1987) 6–479.

[35] V. Kumar, A. Y. Grama, Scalable load balancing techniques for parallel computers, Journal of Parallel and Distributed Computing 22 (1994) 60–79.

[36] A. Reinefeld, V. Schnecke, Work-load balancing in highly parallel depth-first search, in: In Scalable High Performance Computing Conference, 1994, pp. 773–780.

[37] Y. Kitamura, A. Smith, H. Takemura, F. Kishino, Parallel algorithms for real-time colliding face detection, Robot and Human Communication, 1995. RO-MAN'95 TOKYO, Proceedings., 4th IEEE International Workshop on (1995) 211–218.

[38] U. Assarsson, P. Stenström, A case study of load distribution in parallel view frustum culling and collision detection, in: Lecture Notes in Computer Science, 2001, p. 663.

[39] I. Grinberg, Y. Wiseman, Scalable parallel collision detection simulation, Proc. of Signal and Image Processing.

[40] Y.-K. Chen, J. Chhugani, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. Lin, A. D. Nguyen, E. Sifakis, M. Smelyanskiy, High-performance physical simulations on next-generation architecture with many cores, Intel Technology Journal 11 (3).

[41] B. Thomaszewski, W. Blochinger, Physically based simulation of cloth on distributed memory architectures, Parallel Comput. 33 (6) (2007) 377–390.

[42] A. Selle, J. Su, G. Irving, R. Fedkiw, Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction, IEEE Transactions on Visualization and Computer Graphics 99 (2).

[43] B. Thomaszewski, S. Pabst, W. Blochinger, Special section: Parallel graphics and visualization: Parallel techniques for physically based simulation on multi-core processor architectures, Comput. Graph. 32 (1) (2008) 25–40.

[44] D. Kim, J. Heo, S. Yoon, PCCD: Parallel continuous collision detection, Tech. rep., http://sglab.kaist.ac.kr/PCCD/, Korea Advanced Institute of Science and Technology, South Korea (2008).

[45] E. Larsen, S. Gottschalk, M. Lin, D. Manocha, Fast distance queries with rectangular swept sphere volumes, Tech. rep., Department of Computer Science, University of North Carolina (1999).

[46] O. Tropp, A. Tal, I. Shimshoni, A fast triangle to triangle intersection test for collision detection: Research articles, Comput. Animat. Virtual Worlds 17 (5) (2006) 527–535.

[47] M. Tang, S. Yoon, D. Manocha, Adjacency-based culling for continuous collision detection, The Visual Computer, Proc. of CGI08 (Computer

Graphics International 2008) 24 (7-9) (2008) 545–553.

[48] I. Wald, T. Ize, S. G. Parker, Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes, Computers and Graphics 32 (1) (2008) 3 – 13.

[49] D. Field, The impact of multi-core processors on application performance, http://www.hpcwire.com/offthewire/17910484.html (Mar. 2008).

[50] S.-E. Yoon, D. Manocha, Cache-efficient layouts of bounding volume hierarchies, Computer Graphics Forum 25 (3) (2006) 507–516.

[51] J. Lext, U. Assarsson, T. Möller, A benchmark for animated ray tracing, IEEE Comput. Graph. Appl. 21 (2) (2001) 22–31.