박사 학위논문
Ph. D. Dissertation

그래픽스 응용 프로그램을 위한 이종 연산장치 기반 근접질의 병렬처리 기술

Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications

김 덕 수 (金 德 洙 Kim, Duksu)
전산학과
Department of Computer Science

KAIST

2014

그래픽스 응용 프로그램을 위한 이종 연산장치 기반 근접질의 병렬처리 기술

Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications

Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications

Advisor : Professor Yoon, Sung-Eui

by

Kim, Duksu
Department of Computer Science
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2014. 5. 20.

Approved by

Professor Yoon, Sung-Eui

[Advisor]

---

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

그래픽스 응용 프로그램을 위한 이종 연산장치 기반 근접질의 병렬처리 기술

김 덕 수

위 논문은 한국과학기술원 박사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2014년 5월 20일

심사위원장 윤 성 의 (인)

심사위원 김 영 준 (인)

심사위원 박 진 아 (인)

심사위원 신 인 식 (인)

심사위원 허 재 혁 (인)

DCS
20107015

김 덕 수. Kim, Duksu. Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications. 그래픽스 응용 프로그램을 위한 이종 연산장치 기반 근접질의 병렬처리 기술. Department of Computer Science . 2014. 64p. Advisor Prof. Yoon, Sung-Eui. Text in English.

ABSTRACT

Proximity computation is one of the most fundamental geometric operations for various applications including physically-based simulations, computer graphics, robotics, Etc. Also proximity computation is one of the most time consuming parts in various applications. There have been numerous attempts to accelerate the queries like adopting an acceleration hierarchy to cull redundant computations. Even though these methods are general and improve the performance of various proximity queries by several orders of magnitude, there are ever growing demands for further improving the performance of proximity queries, since the model complexities are also ever growing. Recently, the number of cores on a single chip has continued to increase in order to achieve a higher computing power. Also, various heterogeneous computing architectures consisting of different types of parallel computing resources have been introduced. However, prior acceleration techniques such as using acceleration hierarchies gave less consideration for utilizing such parallel architectures and heterogeneous computing environments. Since we are increasingly seeing more heterogeneous computing environments, it is getting more important to utilize them for proximity queries, in an efficient and robust manner.

In this thesis, we employ heterogeneous parallel computing architectures to accelerate the performance of proximity computation for various applications. To efficiently utilize heterogeneous computing resources, we propose parallel computing systems and algorithms for proximity computation. We start with a specific proximity query and design a novel efficient parallel algorithm based on knowledge of the query and computing resources. Then we extend our method to various proximity queries and propose a general proximity computing framework. Also we improve the utilization efficiency of computing resources by designing optimization-based scheduling algorithm. With the proposed methods, an order of magnitude improvement is achieved on various quires by using up to two hexa-core CPUs and four different GPUs over using a single CPU core. In addition we propose an out-of-core proximity computation algorithm to handle a massive data that requires a larger memory space than the memory size of a computing resource in a heterogeneous computing system, especially for the particle-based fluid simulation. The proximity computing system using the out-of-core algorithm robustly works for a large-scale scene and achieves up to two order of magnitude performance improvement over a previous out-of-core approach. These results demonstrate the efficiency and robustness of approaches.

Contents

Abstract i
Contents ii
List of Tables v
List of Figures vi

Chapter 1. Introduction **1**
1.1 Proximity Queries 1
1.2 Parallel Computing Trend and Proximity Queries 1
1.3 Our approaches 3
1.4 List of related papers 4

Chapter 2. Background and Related Work **5**
2.1 Acceleration data structure 5
2.2 Parallel Proximity Query Computation 5
2.3 Lock-free Parallel Algorithms 6
2.4 Scheduling for Unrelated Parallel System 6
2.5 Performance Model 8
2.6 Scheduling for Parallel Proximity Computation 8
2.7 Out-of-Core GPU Algorithms 8

Chapter 3. Hybrid Parallel Continuous Collision Detection using CPUs and GPUs **10**
3.1 Overview 10
3.1.1 Continuous Collision Detection 10
3.1.2 Common BVH-based Collision Detection 11
3.1.3 Overview of our approach 12
3.2 Inter-CD based Parallel CD 12
3.2.1 Inter-CD based Decomposition 13
3.2.2 Initial Task Assignment 14
3.2.3 Dynamic Task Reassignment 14
3.2.4 Parallelizing an Inter-CD Task Unit 15
3.3 GPU-based Elementary Tests 16
3.3.1 Communication between CPUs and GPUs 16
3.4 Implementation and Results 18

3.4.1 Results 19
3.4.2 Analysis 21
3.4.3 Comparisons 23
3.5 Conclusion 23

Chapter 4. Scheduling in Heterogeneous Computing Environments for Proximity Queries 25
4.1 Overview 25
4.1.1 Hierarchy-based Proximity Computation 26
4.1.2 Our Hybrid Parallel Framework 26
4.2 LP-based Scheduling 27
4.2.1 Expected Running Time of Jobs 28
4.2.2 Constrained Optimization 29
4.2.3 Scheduling Algorithm 30
4.2.4 An example work-flow of our iterative LP solver 31
4.2.5 Analysis 32
4.3 Results and Discussions 33
4.3.1 Results and Analysis 35
4.3.2 Optimality 39
4.3.3 Near-Homogeneous Computing Systems 40
4.3.4 Comparison to a Manually Optimized Method 41
4.3.5 Work Stealing with Expected Running Time 42
4.4 Conclusion 43

Chapter 5. Out-of-Core Proximity Computation for Particle-based Fluid Simulations 44
5.1 Out-of-Core, Parallel ϵ-NN 45
5.1.1 System Overview 45
5.1.2 Work Distribution 46
5.1.3 Processing a Block in GPU 48
5.2 Expected Number of Neighbors 49
5.2.1 Problem Formulation 49
5.2.2 Validation and Error Handling 50
5.3 Results and Analysis 52
5.3.1 Results 53
5.3.2 Benefits of Our Memory Estimation Model 54
5.3.3 Benefits of Hierarchical Workload Distribution 55
5.4 Conclusion and Discussion 55

Chapter 6. Conclusion **57**
6.1 Discussions and Future Research Directions 58

References **60**

Summary (in Korean) **65**

List of Tables

3.1 Dynamic Benchmark Models 18

4.1 This table shows constants of our linear formulation computed for continuous collision detection. 29

4.2 An assignment result of the initial assignment step. 32

4.3 The assignment result of the first iteration. 32

4.4 This table shows the average number of iterations in the refinement step and the average time of an iteration. We also compare the quality of the iteratively refined solution (makespan, L_{fin}) with the initial solution (L_{init}) computed from initial assignment step. In this analysis, for each configuration of $|R|$, we run our algorithm for five hundred of randomly generated job sets with the constants in Table 4.1. We add four different GPUs to two hexa-core CPUs one by one as $|R|$ is increased. To focus more on showing benefits of our iterative solver, we turn off the time-out condition in the refinement step in this experiment. 33

4.5 The upper table shows three different machine configurations we use for various tests. The quad-core CPU is Intel i7 (3.2GHz) chip and each hexa-core CPU is Intel Xeon (2.93GHz) chip. The bottom table shows the throughput of each computing resource for the tested benchmarks. 34

4.6 This table shows the average portions of idle time of computing resources in our LP-based method with/without hierarchical scheduling at Machine 2. 40

5.1 This table shows different statistics of each benchmark. We show the average and maximum numbers of neighbors computed for each simulation frame, with the standard deviation. The max. data size is the maximum $s(B)$ among all frames, where B is the whole grid. 53

List of Figures

1.1 These figures show example applications of proximity queries. We accelerate proximity computations in the applications by using heterogeneous computing systems consisting of multi-core CPUs and GPUs. 2

3.1 These images show two frames of our cloth simulation benchmark consisting of 92 K triangles. In this benchmark, our method spends 23 ms for CCD including self-collisions on average and achieves 10.4 times performance improvement by using four CPU-cores and two GPUs over a serial CPU-based CCD method. 11

3.2 This figure shows the overall structure of the HPCCD method. It performs the BVH update and traversal at CPUs and the elementary tests at GPUs. 12

3.3 This figure shows high-level and low-level nodes given four available threads. The right portion of the figure shows an initial task assignment for the four threads. 14

3.4 These images are from the breaking dragon benchmark consisting of 252 K triangles, the most challenging benchmark in our test sets. Our method spends 54 ms for CCD including self-collisions and achieves 12.5 times improvement over a serial CPU-based method. . . 19

3.5 We measure the performance of our HPCCD with four different benchmarks in two machines as we vary the numbers of CPU threads and GPUs. We achieve 10 times to 13 times performance improvements by using the quad-core CPU machine with two GPUs. 20

3.6 This figure shows two frames during the N-body simulation benchmark with two different model complexities: 34 K and 146 K triangles. Our method spends 6.8 ms and 54 ms on average and achieves 11.4 times and 13.6 times performance improvements for two different model complexities. 21

3.7 This figure shows the performance improvement of our HPCCD method as a function of the number of CPU-cores without using GPUs over using a single CPU-core. The gray line, marked by triangles, shows an average performance improvement of the naive approach described in Sec. 3.1.2 with all the tested benchmarks. 22

4.1 The overview of our hybrid parallel framework 26

4.2 This figure shows observed processing time of two different job types on four different computing resources as a function of the number of jobs. 28

4.3 This figure shows the throughput, frames per second, of our hybrid parallel framework, as we add a CPU and two GPUs, in the tested benchmarks. (**C** = a quad-core CPU, **lG** = GTX285 (i.e. low-performance GPU), **hG** = GTX480 (i.e. high-performance GPU)) . . . 36

4.4 This figure shows the throughput, frames per second, of ours and prior scheduling algorithms, as we add more computing resources. (1C = a hexa-core CPU, 1G = GTX285, 2G = 1G+Tesla2075, 3G = 2G+GTX480, 4G = 3G+GTX580) 38

4.5 This figure shows frames per second of our LP-based scheduling and work stealing method with/without hierarchical scheduling on Machine 2. 39

4.6 This figure shows the performance of tested scheduling approaches on a near-homogeneous computing system consisting of two hexa-core CPUs and four identical GPUs (Machine 3 in Table 4.5). 41
4.7 This figure compares the performance of our method with HPCCD, which is optimized specifically for the tested application, continuous collision detection. The throughput, frames per second, includes hierarchy update time. 42

5.1 This figure shows an overall framework for processing ϵ-NNs in an out-of-core manner using heterogeneous computing resources. 45
5.2 The left figure shows a uniform grid with a few sub-grids; boundaries of these sub-grids, i.e., blocks, are shown in orange lines. The right figure shows an example of the workload tree with these blocks. 47
5.3 This plot shows the expected and observed number of neighbors in two configurations, $l = 2\epsilon$ (the left) and $l = \epsilon$ (the right), for the dam breaking benchmark. They show high correlations, 0.97 and 0.96, respectively. 50
5.4 These figures show two different large-scale fluid benchmarks. 51
5.5 These graphs show the processing time in the log scale for ϵ-NNs based on different out-of-core methods including ours. The measured time includes data communication time for sending input data and copying the results to the main memory. Starting from the dotted lines, our method runs in an out-of-core manner. 54
5.6 This graph compares the processing time (seconds) for ϵ-NNs with various space sizes and our estimation model for recording results within our method. 55

Chapter 1. Introduction

1.1 Proximity Queries

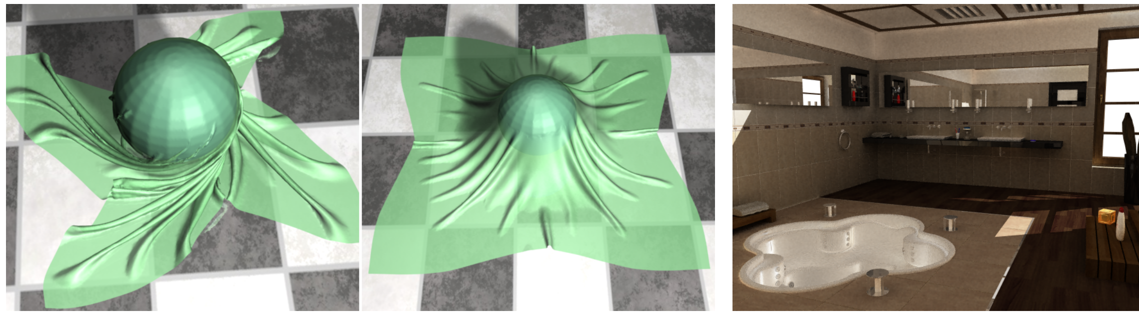
Proximity computation is one of the most fundamental geometric operations, and has been studied in the last two decades for various applications including games, physically-based simulations, ray tracing-based rendering, motion planning in robotics, etc. Some of the most common proximity queries include collision detection and distance query [1]. Collision detection aims to identify inter-collisions occurring between different objects or intra-collisions between different parts of a single object. Many different types of collision computation have been proposed, and their examples include discrete or continuous collision detection (CCD), ray-triangle intersection tests used for ray tracing, etc. On the other hand, distance query aim to compute different types of distances, e.g. minimum separation distance, Hausdorff distance, k-nearest neighbor computations, between a single or multiple objects [2]. Distance query also includes neighbor search queries that find k-nearest neighbors (k-NN) or neighbors within a specific distance (i.e. ϵ-nearest neighbor search, ϵ-NN) from a query point.

There have been numerous attempts to accelerate the queries. One of the most general approaches is adopting an acceleration hierarchy such as KD-trees [3] or bounding volume hierarchies (BVHs) [1, 4]. Even though this method is general and improves the performance of various proximity queries by several orders of magnitude, there are ever growing demands for further improving the performance of proximity queries, since the model complexities are also ever growing. For example, proximity queries employed in interactive applications such as games should provide real-time performance. However, it may not meet such requirement, especially for large-scale models that consist of hundreds of thousands of triangles.

1.2 Parallel Computing Trend and Proximity Queries

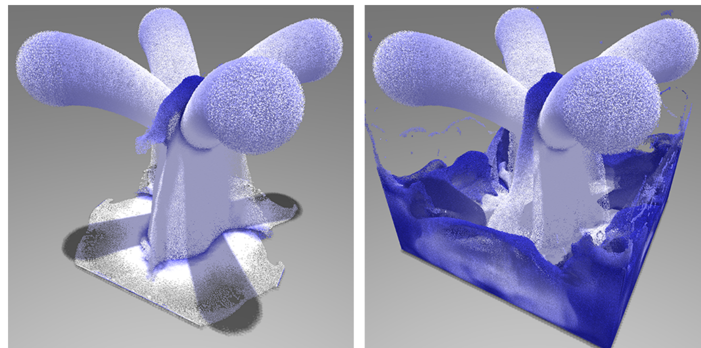
Recently, the number of cores on a single chip has continued to increase in order to achieve a higher computing power, instead of continuing to increase the clock frequency of a single core [5]. Currently commodity CPUs have up to eight cores and GPUs have hundreds of cores [6]. Another computing trend is that various heterogeneous multi-core architectures such as Sony's Cell, Intel Sandy Bridge, and AMD Fusion chips are available. The main common ground of these heterogeneous multi-core architectures is to combine CPU-like and GPU-like cores in a single chip. Such heterogeneity has been growing even more in the cloud computing environment, which is currently a wide-spreading trend in (high-performance computing) IT industry. Even though a cloud service can start with homogeneous computing nodes, the capacities of computing nodes vary a lot over time due to upgrade and replacement [7].

However, prior acceleration techniques such as using acceleration hierarchies do not consider utilizing such parallel architectures and heterogeneous computing environments. Since we are increasingly seeing more heterogeneous computing environments, it is getting more important to utilize them for various applications, including proximity queries, in an efficient and robust manner.



(a) Physically-based simulation (cloth benchmark) (b) Rendering (path tracing)

(c) Motion planning



(d) Particle-based fluid simulation

Figure 1.1: *These figures show example applications of proximity queries. We accelerate proximity computations in the applications by using heterogeneous computing systems consisting of multi-core CPUs and GPUs.*

1.3 Our approaches

Along the parallel computing trend, many parallel algorithms have been proposed and achieved high performance improvements. However they only utilize either multi-core CPUs or GPUs, and it is still not enough to provide interactive performance for large-scale models.

Different with those prior parallel algorithms, we utilize all available computing resources including multi-core CPUs and GPUs in heterogeneous computing systems to accelerate the proximity computation. Also we maximize the utilization efficiency of the heterogeneous systems by designing efficient heterogeneous parallel framework and workload distribution (i.e. scheduling) algorithms that considers the heterogeneity of the computing resources.

In this thesis we first focus on a specific queries, continuous collision detection and present a novel hybrid parallel continuous collision detection (HPCCD) method that utilizes both multi-core CPUs and GPUs (Chapter. 3). Our HPCCD method distributes jobs of collision detection to multi-core CPUs and GPUs based on the knowledge of the query and computing resources. In order to design highly scalable CPU-based parallel algorithm, we also propose a novel task decomposition that leads a synchronization-free parallel algorithm in the main loop of the collision detection process. With these approaches we improve the performance of continuous collision detection an order of magnitude using four CPU-cores and two GPUs compared with using a single CPU-core.

However there are generality and optimality issues. Since we design HPCCD for a specific query, it is unclear how this method works well with other queries. Also it uses manually defined workload distribution depending on the application-dependent heuristic and there is no guarantee to efficient utilization for computing resources.

We handle the generality issue and propose a general framework for various proximity queries in Chapter 4. We first factor out two common job types of proximity queries and present various proximity queries as the combination of the two common job types. Based on this unified representation of various proximity computations, we design heterogeneous computing framework that processing jobs of the two common job types. Then we propose an optimization-based scheduling algorithm and handle the optimality issue (Chapter 4.2). We represent our scheduling problem as a linear programming problem and compute an optimal job distribution that minimizes the running time of the computing system. We apply the proposed framework and scheduling algorithm to various proximity queries on various combinations of computing resources, and it achieves higher performance improvement compared with previous scheduling algorithms. These results demonstrate the generality and optimality of our approaches.

In above approaches, we assume that all computing resources have enough memory space and do not consider heterogeneity of computing resources in the perspective of memory space. However for a massive data, this assumption is not valid and we extend our attention to out-of-core proximity computation techniques.

Particle-based fluid simulation is one of the representative applications that requires handling a massive data set since a large number of particles are required to meet the higher realism. In Chapter. 5, we propose an out-of-core proximity computation method to handle a large scale particle-based fluid simulation that requires a larger memory space than the memory space of a computing resource (i.e. GPU) in heterogeneous computing systems. Our out-of-core technique includes memory footprint estimation model and hierarchical work distribution method that identifies the maximal size of work group the target computing resource can handle at once. As a result, our method robustly works for large-scale scenes while achieving an order of magnitude improvement compared with the case of using

a prior out-of-core technique.

Although we propose out-of-core proximity computation method only for an application (i.e., particle-based fluid simulation) in this thesis, it is also important issue for other proximity queries since the data size is continuously increased in various fields. We discuss about this issue in Chapter 6 to guide future research direction.

1.4 List of related papers

Some of parts of this thesis have been published. Following is a list of papers related to this thesis.

- **Duksu Kim**, Jae-Pil Heo, Jaehyuk Huh, John Kim, Sung-Eui Yoon, *HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs*, Computer Graphics Forum, vol. 28, no. 7, pp. 1791-1800, 2009
- **Duksu Kim**, Jinkyu Lee, Junghwan Lee, Insik Shin, John Kim, Sung-Eui Yoon, *Scheduling in Heterogeneous Computing Environments for Proximity Queries*, IEEE Transactions on Visualization and Computer Graphics, vol. 19, no. 9, pp. 1513-1525, 2013
- **Duksu Kim**, Myung-Bae Son, Young J. Kim, Jeong-Mo Hong, Sung-Eui Yoon, *Out-of-Core Proximity Computation for Particle-based Fluid Simulations*, Dept. of CS, KAIST, Technical Report CS-TR-2014-385, 2014

Chapter 2. Background and Related Work

2.1 Acceleration data structure

One of the most general approaches for accelerating proximity queries is employing an acceleration data structure such as KD-tree, bounding volume hierarchies (BVHs), and grid data structure [1, 3, 8]. Such acceleration data structure divide the space hierarchically (e.g., KD-tree and BVHs) or uniformly (e.g., uniform grid) and are used to narrow down the search space. In this thesis we design heterogeneous parallel algorithm on top of the acceleration data structure based proximity computation algorithms.

In Chapter 3 and 4 we use BVHs as the acceleration hierarchy for various proximity queries; KD-tree or other acceleration hierarchies can be used with our approaches as well. The Bounding volume (BV) is a simple volume enclosing one or more triangles of complex objects. BVHs are simply a tree of BVs and widely used technique to accelerate the performance for collision detection, ray tracing and visibility culling. There are different types of BVs such as sphere [9], axis-aligned bounding boxes (AABBs) [10], oriented bounding boxes (OBB) [11], k-DOPs [12] and etc. Most algorithms for deformable models typically use simple BVs such as spheres or AABBs and recompute the BVH during each frame [13]. Approaches to update dynamic BVHs include refitting these hierarchies [14, 15, 16], and performing dynamic or selective restructuring [17, 18, 19].

In Chapter. 5, we employ an uniform grid data structure, a commonly employed acceleration data structure for ϵ-nearest neighbor search in particle-based simulations [8]. To construct an uniform grid, we split the simulation space uniformly with a given grid resolution. Each divided space is called a cell and each point (or particle) is included in one of the cells depending on the position. Then we can localize the search space depending on the relative placement of cells.

2.2 Parallel Proximity Query Computation

Many parallel proximity query algorithms on multi-core CPUs have been proposed for various proximity queries. Lawler and Laxmikant [20] proposed a voxel-based collision detection 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 [21] designed a parallel collision detection algorithm for virtual prototype environment. This method is based on a culling method considering overlapping areas between BVs. However, it does not use acceleration hierarchies (e.g., BVHs) 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 were added. Lee et al. [22] proposed a simple load-balancing metric based on the penetration depth (for collision detection) and approximated Euclidean distance (for distance computation) between bounding volumes. They achieved high scalability (e.g., 5 and 9 times for collision detection and distance computation respectively) by using eight CPU cores compared with sing core computation. Tang et al. [23] proposed a front-based task decomposition method that utilizes multi-core processors for collision detection between deformable models. Their CPU-based parallel method achieves a high scalability by even using 16 CPU-cores. We will compare our method

with this CPU-based parallel method in Sec. 3.4.3.

There also have been considerable efforts to perform proximity query efficiently with GPUs [24, 25, 26, 27]. Govindaraju et al. [26] proposed an approach for fast collision detection between complex models using GPU-accelerated visibility queries. Kolb et al. [28] introduced an image-based collision detection algorithm for simulating a large particle system. There have been GPU-based algorithms for self-collision and cloth simulations [29, 30, 31] 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. Gress et al. [32] introduced a BVH-based GPU collision detection method for deformable parameterized surfaces. Also, several unified GPU-frameworks for various proximity queries have been proposed [33, 27].

For nearest neighbor search queries, parallel computing resources also have been actively used to improve the performance. Many prior parallel methods are designed for k-NN used for photon mapping [34, 35], 3D registration [36], etc. Parallel algorithms for ϵ-nearest neighbor (ϵ-NNs) search have been actively studied in the particle-based fluid simulation field. By utilizing the inherent parallel nature of many ϵ-NNs, efficient GPU-based SPH implementations have been proposed [37]. These methods distribute particles to threads and each thread finds the neighbors of the given particle. While these approaches are simple, they are not designed for out-of-core cases and the number of particles is limited to the video memory size, e.g., about 5 M for 1 GB video memory [38]. Ihmsen et al. [8] used multi-core CPUs in the whole process of SPH. They showed that a CPU-based parallel approach can handle a larger number of particle (e.g. 12 M) thanks to the large memory space (e.g., 128 GB) in the CPU side. To handle a large number of particles with GPU, a multi-GPU approach was also proposed [39]. This approach partitioned the simulation space into multiple GPUs and adopted a MPI based distributed computing among those GPUs. For example, they used four GPUs (GTX480 with 1.5 GB) to simulate 40 M particles. In Chapter 5, we propose a novel out-of-core techniques that can handle a large number particles even with a single GPU as long as main memory in CPU can hold all the data. Furthermore, our method runs much faster than the MPI-based distributed computing approach, more suitable for graphics simulations.

2.3 Lock-free Parallel Algorithms

The traditional synchronization based on locks can degrade the performance of parallel algorithms because of lock contention [40]. To address this problem, there have been many efforts to reduce or eliminate the use of locks by designing lock-free algorithms relying on atomic swap instructions [41]. However, these lock-free algorithms are based on the assumption that actual lock contentions are rare and thus reducing conflicting accesses to shared data structure is crucial.

On the other hand, we eliminates conflicting accesses to shared acceleration hierarchy (i.e., BVH) data in the main loop of PQ part based on our novel task decomposition method, although locks are only used in non-critical parts (Sec. 3.2.1).

2.4 Scheduling for Unrelated Parallel System

Scheduling concerns allocating jobs to resources for achieving a particular goal, and has been extensively studied [42]. We are interested in finding the optimal job distribution that maximizes the throughput of entire parallel system. This is known as minimizing the *makespan*. At high level, a

parallel system can be classified as identical, related, or unrelated. *Unrelated* parallel system consists of heterogeneous computing resources that have different characteristics and thus performance varies. *Related* or *identical* systems, on the other hand, are composed of resources that are similar or the exactly same in terms of characteristics and performance, respectively. Our scheduling method aims for most general parallel systems, such as unrelated parallel machines (i.e. heterogeneous computing systems).

In theoretical communities, minimizing the makespan for unrelated parallel machines is formulated as an integer programming (IP). Since solving IP is NP-hard, many approximate approaches with quality bounds have been proposed. These approximate methods achieve polynomial time complexity by using a linear programming (LP), which relaxes the integer constraint on the IP [43, 44, 45]. Lenstra et al. [44] proved that no LP-based polynomial algorithms guarantee an approximate bound of 50% or less to the optimal solution, unless $P = NP$. This theoretical result applies to our problem too.

In addition to theoretical results, many heuristic-based scheduling methods have been proposed for unrelated parallel machines [46, 47]. Al-Azzoni et al. [47] proposed approximated LP-based scheduling algorithms. Since gathering various information from all the resources incurs a significant overhead in their heterogeneous system, they considered information (e.g., the minimum completion time) from only a subset of resources. These techniques considered the scheduling problem in simulated environments or specific applications (e.g., an encryption algorithm). Moreover, they focused on studying theoretical or experimental quality bounds rather than a practical performance on working heterogeneous computing systems.

Topcuoglu et al. [48] proposed an insertion-based task scheduling method, Heterogeneous-Earliest-Finish-Time algorithm (HEFT), that accounts for the execution time of all tasks. Augonnet et al. [49] improved HEFT by considering the data transfer overhead. HEFT assigns tasks to processors one-by-one as it minimize the earliest finished time at each step. The algorithm provide high-quality scheduling results even if the types of jobs are different each other. However, the computational overhead increases as the number of jobs increases and it is unclear how much performance HEFT realizes for proximity query computations. This is mainly because we usually have hundreds of thousands of jobs at each scheduling time. In comparison, our scheduling algorithm finds how to distribute a set of jobs to multiple computing resources by solving optimization problem and its computational cost depends on the number of job types. Therefore, our method is well suit to the proximity computations that have a small number of job types.

Work stealing is a well-known workload balancing algorithm in parallel computing systems [50, 51]. Kim et al. [52] showed that the work stealing method achieved a near-linear performance improvement up to eight CPU cores for continuous collision detection. Hermann et al. [53] employed a work stealing approach to parallelize the time integration step of physics simulation with multi-GPUs and multi-CPUs. They compute an initial task distribution depending on a task graph to minimize inefficient inter-device communication caused by the work stealing. In case of proximity queries, however, it is hard to compute the task graph, since tasks are sporadically generated during processing prior tasks. To further improve the efficiency of work stealing methods, various knowledge about applications or computations can be utilized [54, 53, 55]. Hermann et al. [53] employed a priority-guided stealing approach and made GPUs steal time-consuming jobs first and CPUs take opposite strategy since small jobs decreases the efficiency of using GPUs. For fair comparison with those prior work, we also apply the priority-guided stealing approach to our basic work stealing implementation based on knowledge of proximity computations (Sec. 4.3). We compare our scheduling method with the basic work stealing approach and improve the efficiency of the working stealing method based on one of our contributions (Sec. 4.3.5).

2.5 Performance Model

For high quality scheduling results, scheduling algorithms commonly rely on performance models that predict how much computational time a resource takes to finish tasks. Literatures in the field of the scheduling theory often assume that there is a mathematical performance model for their scheduling problems [49]. Few works gave attention to modeling overheads such as data communication costs [46]. Performance models for GPUs and heterogeneous framework recently have been proposed [56, 57, 58, 59]. These architectural performance models can be very useful to accurately predict the computational time of jobs in a broad set of GPUs.

To design efficient scheduling algorithm, we use a simple performance model of jobs occurred in different proximity queries (Sec. 4.2.1). Our performance model can be efficiently computed with observed performance data and can be effectively incorporated within our LP-based scheduling method.

2.6 Scheduling for Parallel Proximity Computation

Many scheduling methods have been also proposed for various parallel computations for proximity queries. Tang et al. [23] group jobs into blocks and assign a block to each idle CPU thread in a round robin fashion. Lauterbach et al. [27] check the workload balance among cores on a GPU and perform a workload balancing (i.e. distributing jobs evenly) when the level of workload balance is low.

Only a few works have proposed utilizing heterogeneous multi-core architectures. Budge et al. [60] designed an out-of-core data management method for ray tracing on hybrid resources including CPUs and GPUs. They prioritize different jobs in ray tracing and assign them into either a CPU or a GPU, by checking which processor the jobs prefer and where the required data is stored. In our HPCCD method (Chap. 3), we decompose continuous collision detection into two different task types and manually dedicate all the tasks of each job type into one of the CPU or GPU.

It is unclear, however, how well these techniques can be applied to a wide variety of proximity queries, since they use manually specified rules for a specific application, or rely on application-dependent heuristics. Furthermore, their approaches do not rely on optimization frameworks to maximally utilize available computing resources. Departing from these prior application-dependent scheduling methods, our optimization-based scheduling method (Chap. 4) takes a general and robust approach in order to minimize the makespan of a wide variety of proximity queries with hybrid resources. We compare our method with these prior methods in Sec. 4.3.

2.7 Out-of-Core GPU Algorithms

The limited video memory space raises various challenges for handling a large data set in GPU. The out-of-core issue has been well studied for rendering [61]. Nonetheless, it has not been actively studied for different parts of particle-based fluid simulations.

Abstracting distributed memory space of CPU and GPU into a logical memory is a general approach for handling massive data with GPU. Nvidia's CUDA supports a memory space mapping method that maps pinned-memory space into the address space of GPU [62]. While it is convenient to use, it can be inefficient, unless minimizing expensive I/O operations effectively. We compare this mapped memory based out-of-core approach with ours in Sec. 5.3.

Different out-of-core techniques have been proposed for k-NN used in ray tracing and photon mapping. In particular, Budge et al. [60] designed an out-of-core data management system for path tracing with kd-trees constructed over polygonal meshes. This approach adopted a pre-defined task assignment policy to distribute different jobs to CPU or GPU. Recently, Kim et al. [63] used separate, decoupled data representations designed for meshes to fit large-scale data in the video memory. Unfortunately, it is unclear how these techniques designed for k-NNs can be applied to other proximity queries.

Chapter 3. Hybrid Parallel Continuous Collision Detection using CPUs and GPUs

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

3.1 Overview

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

3.1.1 Continuous Collision Detection

Collision detection is classified as two categories: discrete and continuous methods. *Discrete collision detection* (DCD) finds intersecting primitives at discrete time steps. DCD can be performed quite efficiently, but may miss colliding primitives that occur between two discrete time steps. This issue can be quite problematic in physically-based simulations, CAD/CAM, etc. On the other hand, *continuous collision detection* (CCD) identifies intersecting primitives at the first time-of-contact (ToC) during a time interval between two discrete time steps.

In order to find intersecting primitives at the first ToC during a time interval between two discrete time steps, CCD methods model continuous motions of primitives by linearly interpolating positions of primitives between two discrete time steps. We also use this linear continuous motion. There are two types of contacts: *inter-collisions* between two different models and *intra-collisions*, i.e., *self-collisions*, within a model. Both contacts arise in two contact configurations, vertex-face (VF) case and edge-edge (EE) cases. These two cases are detected by performing VF and EE elementary tests, which reduce to solving cubic equations given the linear continuous motion between two discrete time steps [64].

BVHs are widely used to accelerate the performance of CCD methods. We use the AABB representation because of its fast update method, simplicity, and wide acceptance in various collision detection methods [13]. Given a BV node n of a BVH, we use notations of $L(n)$ and $R(n)$ to indicate the left child and right child nodes of the node n. As models are deforming, we have to update BVHs of such deforming models. We update BVHs based on a selective restructuring method, which reconstructs small portions

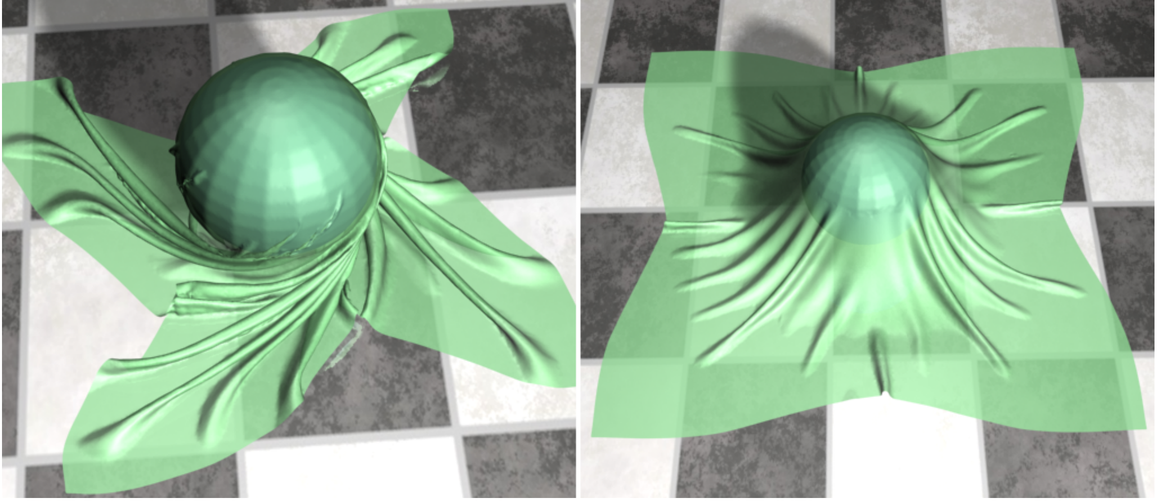


Figure 3.1: *These images show two frames of our cloth simulation benchmark consisting of 92 K triangles. In this benchmark, our method spends 23 ms for CCD including self-collisions on average and achieves 10.4 times performance improvement by using four CPU-cores and two GPUs over a serial CPU-based CCD method.*

of BVHs that may have poor culling efficiency and refits the rest of portions of BVHs by traversing the BVH in a bottom-up manner [13]. We also combine the selective restructuring method with a lazy BVH construction method [19]. In the lazy BVH construction, only small parts (e.g., one or two levels) of subtree of a node is constructed when it firstly visited. This can reduce hierarchy update overhead since some parts of a BVH are not visited if they are culled.

3.1.2 Common BVH-based Collision Detection

For BVHs of deforming models, we merge these BVHs into a BVH and then perform our CCD method with the merged BVH. In this case, inter-collisions among multiple objects and self-collisions within each object can be computed by performing self-collision detection with the merged BVH [13]. We initially perform collision detection between two child nodes of the root node of the BVH. To do that, we create a *collision test pair* consisting of these two nodes. Then, we push the pair into a queue, called *collision test pair queue*. In the main loop of the CD algorithm, we dequeue a pair (n, m) consisting of BV nodes n and m from the queue and perform a BV overlap test between two BV nodes, n and m, of the pair. If there is an overlap, we refine two BV nodes with their two child BV nodes and create four different collision pairs, $(L(n), L(m))$, $(L(n), R(m))$, $(R(n), L(m))$, and $(R(n), R(m))$. If we have to find self-collisions within nodes n and m for dynamically deforming models, we also create two additional collision pairs, $(L(n), R(n))$ and $(L(m), R(m))$. When we reach leaf nodes during the BVH traversal, we perform the VF and EE elementary tests between features (e.g., vertex, edges, and faces) associated with the leaf nodes. We continue this process until there is no more collision pairs in the queue.

Issues of parallelizing the BVH-based CD: Parallelizing the BVH-based CD is rather straightforward. One naive approach is to divide the pairs stored in the collision test pair queue into available threads. Then, each thread performs the BVH traversal and adds collision test pairs into its own queue without any locking. However, threads have to use a locking mechanism for reconstructing a BV of a node. We found that this naive method shows poor scalability (Fig. 3.7). Two issues cause this low

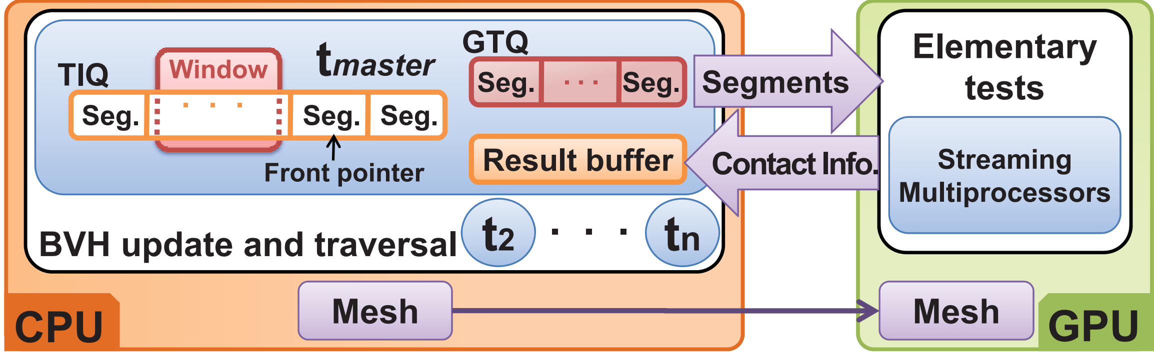


Figure 3.2: *This figure shows the overall structure of the HPCCD method. It performs the BVH update and traversal at CPUs and the elementary tests at GPUs.*

performance. The first one is that contacts among objects occur in localized regions of objects and processing a pair may generates a high number of additional pairs or may terminate soon after. This high variance of the computational workload associated with each pair requires frequent redistributions of computational workload for a load-balancing among threads and results in a high overhead. The second one is that using locks to avoid simultaneous reconstructions on the same node serializes these multiple threads and hinders the maximum utilization of multi-core architectures. In this chapter, we propose a scalable parallel CCD method that addresses these issues.

3.1.3 Overview of our approach

At a high level, our HPCCD method consists of two parts: 1) CPU-based BVH update and traversal with lazy BV reconstructions and 2) GPU-based VF and EE elementary tests (see Fig. 3.2).

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

3.2 Inter-CD based Parallel CD

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

Terminologies: We define a few terminologies to describe our method. We define an *inter-collision test pair set*, $ICT_{PS}(n)$, to denote all the collision test pairs generated in order to find inter-collisions between two child nodes of a node n. We define two nodes to have a *parent-child relationship* if one node is in the sub-tree rooted at the other.

3.2.1 Inter-CD based Decomposition

Our task decomposition method for the parallel CPU-based BVH traversal is based on task units of an inter-collision detection, *inter-CD*. Each inter-CD task unit processes collision test pairs represented by $ICT_{PS}(n)$ of a node n. To assign task units to each CPU thread, we push a node n into a *task unit queue* and associate the queue with the thread. Inter-CD task unit has two phases: 1) setup phase and 2) the BVH traversal phase performing BV overlap tests. In the setup phase of an inter-CD task, we first fetch a node, n_S, from its task unit queue and refine the node into its two child nodes $L(n_S)$ and $R(n_S)$. If we have to perform the self-collision detection, we push those two child nodes that will generate other inter-CD task units into the task unit queue. We also create a *scheduling queue* for dynamic task reassignment for a load-balancing, which will be explained in Sec. 3.2.3.

After the setup phase, we perform the BVH traversal phase. During the BVH traversal phase, we use a collision test pair queue as used in the common BVH-based traversal explained in Sec. 3.1.2. We assign a collision test pair $(L(n_S), R(n_S))$ into the collision test pair queue. Then, we fetch a pair consisting of two nodes n and m from the collision test pair queue and perform a BV overlap test between two BV nodes n and m of the pair. If there is a BV overlap, we refine both of those two nodes into $L(n)$, $R(n)$, $L(m)$, and $R(m)$. Then, we construct four collision test pairs, $(L(n), L(m))$, $(L(n), R(m))$, $(R(n), L(m))$, and $(R(n), R(m))$, and push them in the collision test pair queue. We continue this process until we reach leaf nodes. If we reach leaf nodes, we perform exact VF and EE elementary tests between features associated with the leaf nodes by using GPUs. If there is any collision, we put the collision result into a *result buffer*.

Disjoint property of inter-CD task units: During processing an inter-CD task unit of a node n, we create and test various collision test pairs, $ICT_{PS}(n)$, of nodes that are in the sub-tree rooted at the node n. If there is no parent-child relationship between two nodes, say n and m, we can easily show that a set of accessed nodes during performing $ICT_{PS}(n)$ is disjoint from another set of accessed nodes during performing $ICT_{PS}(m)$. We will utilize this disjoint property to design an efficient parallel CCD algorithm (Sec. 3.2.2 and Sec. 3.2.3).

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

Note that our serial CCD algorithm based on inter-CD task units is constructed by simply reordering the processing order of collision test pairs of typical BVH-based CD methods explained in Sec. 3.1.2.

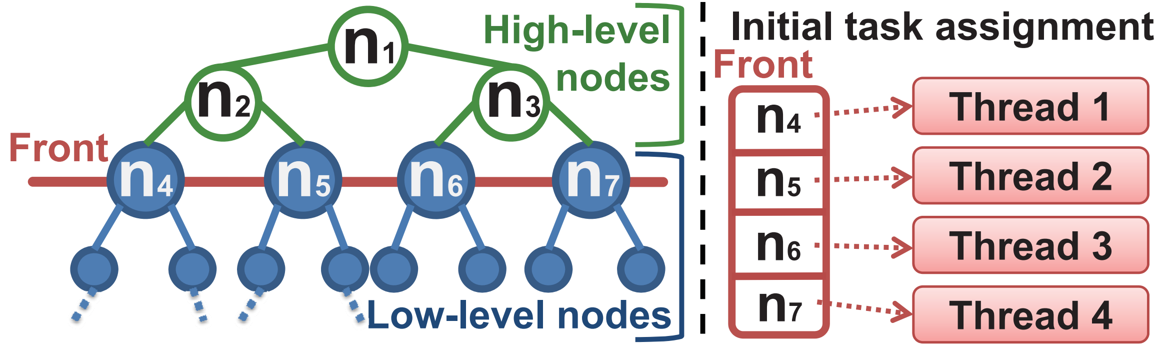


Figure 3.3: *This figure shows high-level and low-level nodes given four available threads. The right portion of the figure shows an initial task assignment for the four threads.*

3.2.2 Initial Task Assignment

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

To guarantee that nodes assigned to threads do not have any parent-child relationship and thus satisfy the disjoint property, we compute such nodes by maintaining a front while we traverse the BVH from its root node in the breadth-first order (see Fig. 3.3). If the size of the front is same as the number of available threads, we stop the BVH traversal and assign each node in the front to each thread. We refer to those nodes and all the nodes in the sub-trees rooted at those nodes as *low-level nodes* while all the other nodes are referred to as *high-level nodes*. An example of low-level and high-level nodes for four threads is shown in Fig. 3.3.

In the same manner with the serial CCD method described in Sec. 3.2.1, each thread finds collisions that occur in a subtree of the node that is initially assigned to the thread. Once each thread finishes its computation, we process inter-CD task units of high-level nodes. First, we process parent nodes, n_2 and n_3 in the case of Fig. 3.3, of initially assigned nodes to each thread. We wait until the processing of inter-CD task units of two nodes n_2 and n_3 finishes and then process their parent node, n_1. While processing high-level nodes, we do not add child nodes of those nodes to the task unit queue since we already processed inter-CD tasks of those child nodes.

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

3.2.3 Dynamic Task Reassignment

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

unit queue among threads. Then, the distribution thread t_{dist} gives the front node in its task unit queue to the requesting thread $t_{request}$.

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

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

In each thread, we check whether its own scheduling queue is empty or not by looking at its size right after finishing all the collision test pairs and before performing another inter-CD task unit. If there are no requests in the queue, the thread continues to process another inter-CD task unit by fetching a node from its task unit queue. If there are requests in the scheduling queue, we distribute its computational workload stored in the task unit queue to the requesting threads. After distributing the computational workload, the distribution thread t_{dist} sends wake-up messages with the partitioned nodes to the requesting threads. Once a requesting thread $t_{request}$ receives the wake-up message, the thread pushes the received node into its task unit queue and resumes its computation by performing inter-CD task units. Pseudo-code of our parallel CCD algorithm based on inter-CD task units is shown in Listing 3.1. Note that we do not perform any synchronization nor locking in the main collision detection loop.

3.2.4 Parallelizing an Inter-CD Task Unit

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

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

Listing 3.1: Pseudocode of HPCCD method

```
Perform_Self_CD (node n) {
        TaskUnit_Q <- n;
        while (! TaskUint_Q.Empty ()) {
                n <- TaskUint_Q.Dequeue ();
                if (n has child nodes) {
                        TaskUint_Q <- L(n) and R(n);
                        Pair_Q <- (L(n),R(n));
                }
                while (! Pair_Q.Empty ()) { // Main CD loop
                        Pair <- Pair_Q.Dequeue () ;
                        Perform lazy reconstruction for nodes of
                         Pair;

                        if (IsOverlap (Pair)){
                                if (IsLeaf (Pair) )
                                        Perform elementary tests;
                                else
                                        Pair_Q <- Refine (Pair);
                        }
                }
                if (! SchedulingQ.Empty ())
                        Distribute its work to the requesting
                         thread;
        }
        Request to a distribution thread and sleep;
}
```

3.3 GPU-based Elementary Tests

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

3.3.1 Communication between CPUs and GPUs

In our HPCCD framework, multiple CPU threads generate elementary tests simultaneously. For sending data from CPUs to GPUs, an easy solution would be to let each CPU working thread send two triangle indices to GPUs. However, we found that this approach requires a high overhead since GPUs

have to maintain individual device contexts for each CPU thread [6]. Instead, we use a master CPU-GPU communication thread, t_{master} (see Fig. 3.2). Each CPU thread requests the master thread to send the data to GPUs. The overall communication interface between the CPUs and GPUs is shown in Fig. 3.2.

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

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

Procedure of the master thread: The master CPU-GPU communication thread performs the following steps in its main loop. The first step is to look at the consecutive segments in the TIQ's window. If there are multiple consecutive "full" segments, we send these consecutive "full" segments to GPUs with one data sending API call and push them in one element of the GTQ. The reason why we send consecutive segments is to reduce the number of calls of data sending API, which has a high kernel call overhead [6]. We then update the window position by sliding it right after the transmitted consecutive segments in the TIQ. Note that any "partial" segments are not sent and will be checked again for the transmission when the window contains these segments later. When a GPU finishes processing all the elementary tests constructed from the segments received, the GPU sets the state variable of the element of the GTQ to be "data ready" and stores all the colliding primitives in the output buffer of the queue element. As a next step, the master thread goes over all the elements in the GTQ and remove elements that have the "data ready" state. We also update the result buffer that contains the contact information at the first ToC with output buffers of these removed elements. Then, we make segments of these "data ready" elements to have the "empty" state in order to be reused for a next request of "empty" segments. The final step of the master thread is to process requests of "empty" segments from CPU threads.

Load balancing between CPUs and GPUs: When we use a low-end GPU and high-end CPUs, we found that the GPU may not process all the elementary tests generated from these CPU-cores, thus requiring additional GPUs to load-balance and achieve a further performance improvement. Without having additional GPUs, we can load-balance the elementary tests across both the CPUs and GPUs to

Model	Tri. (K)	Image	Avg. CCD time (ms)
Cloth simulation	92	Fig. 3.1	23.2
Breaking dragon	252	Fig. 3.4	53.6
LR N-body simulation	32	Fig. 3.6	6.8
HR N-body simulation	146	Fig. 3.6	53.8

Table 3.1: *Dynamic Benchmark Models*

achieve an additional performance improvement by using CPUs to process the elementary tests instead of generating and sending the elementary tests to the GPUs. To do this, a CPU working thread checks whether the GTQ's size is bigger than the number of GPU cores, right before the CPU thread finishes to fill a segment and requests an "empty" segment. If the GTQ's size is bigger than the number of GPU cores, we assume that the GPUs are busy and cannot process all the segments produced by CPUs. In this case, the CPU thread processes the half of the elementary tests of the "full" segment and continues to performs the BVH traversal until it fills the half-empty segment. Once the segment becomes full again, the CPU thread checks the queue size of GTQ and performs the same procedure again.

3.4 Implementation and Results

We have implemented our HPCCD method and tested it with two different machines. The first one is an Intel Xeon desktop machine with two 2.83 GHz quad-core CPUs and a GeForce GTX285 GPU. The second one is an Intel i7 desktop machine with one 3.2 GHz quad-core CPU and two Geforce GTX285 GPUs. The Intel i7 processor supports the hyper-threading [40]. We will call these two machines 8C1G (8 CPU-cores and 1 GPU) and 4C2G (4 CPU-cores and 2 GPUs) machines for the rest of the chapter. We use the *OpenMP* library [65] for the CPU-based BVH traversal and CUDA [6] for the GPU-based elementary tests. We also use a feature-based BVH using *Representative-Triangles* [66] in order to avoid any redundant VF and EE elementary tests.

Parallel BVH update: Before we perform the CCD using a BVH, we first refit BVs of the BVH. Our algorithm traverses the BVH in a bottom-up manner and refits the BVs. To design a parallel BVH refitting method utilizing k threads, we compute $2k$ nodes in a front as we did for the initial task assignment of inter-CD task units to threads in Sec 3.2.2. Then, we assign the first k nodes stored in the front to each thread and then each thread performs the BV refitting to the 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. For the thread finishing its refitting, we assign the next available node in the front to the thread. During the BVH traversal, we identify and selectively restructure BVs with low culling efficiency. To do that, we use a heuristic metric proposed by Larsson and Akenine-Möller [14]. We perform a lazy BV reconstruction by using a simple median-based partitioning of triangles associated with the node.

Benchmarks: We test our method with three types of dynamic scenes (see Table 3.1). The first benchmark is a cloth simulation, where a cloth drapes on a ball and then the ball is spinning (Fig. 3.1). This benchmark consists of 92 K triangles and undergoes severe non-rigid deformations. In our second benchmark, a bunny collides with a dragon model. Then, the dragon model breaks into numerous pieces (Fig. 3.4). This model has 252 K triangles. Our final benchmark is a N-body simulation consisting of

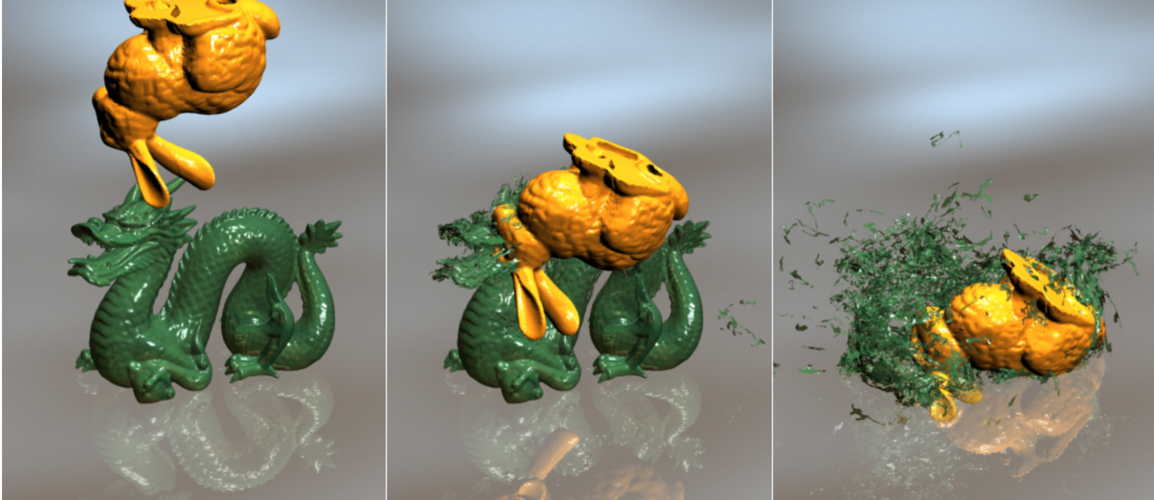


Figure 3.4: *These images are from the breaking dragon benchmark consisting of 252 K triangles, the most challenging benchmark in our test sets. Our method spends 54 ms for CCD including self-collisions and achieves 12.5 times improvement over a serial CPU-based method.*

multiple moving objects (Fig. 3.6). We compute two different versions with different model complexities of this benchmark: a high-resolution (HR) version has 146 K triangles and a low-resolution (LR) has 32 K triangles. Each object in this benchmark may undergo a rigid or deformable motion and objects collide with each other and the ground. These models have different model complexities and characteristics. As a result, they are well suited for testing the performance of our algorithm.

3.4.1 Results

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

In the 4C2G machine, the HPCCD method spends 23.2 milliseconds (ms), 53.6 ms, 6.8 ms, and 53.8 ms on average for the cloth simulation, the breaking dragon, and LR/HR N-body simulations respectively; we will report results in this benchmark order for the rest of the tests. These computation times translate to about 43, 19, 148 and 19 frame per seconds (fps) for the four benchmarks respectively. Compared to using a single CPU-core, we achieve 10.4, 12.5, 11.4, and 13.6 times performance improvements. We also show the performance of HPCCD with different numbers of CPU threads and GPUs with the two different machines (see Fig. 3.5).

We measure the scalability of our CPU-based BVH update and traversals of our HPCCD method as a function of CPU threads (e.g., 1, 2, 4, and 8 threads) without using GPUs in the 8C1G machine (see Fig. 3.7) with all the benchmarks. The CPU part of our HPCCD method shows 6.5, 6.5, 6.4, and 7.1 times performance improvements by using 8 CPU-cores over a single CPU-core version in the four benchmarks respectively. We achieve a stable and high scalability near the ideal linear speedup across our benchmarks that have different model complexities and characteristics. This high scalability is due to the lock-free parallel algorithm used in the main loop of the collision detection.

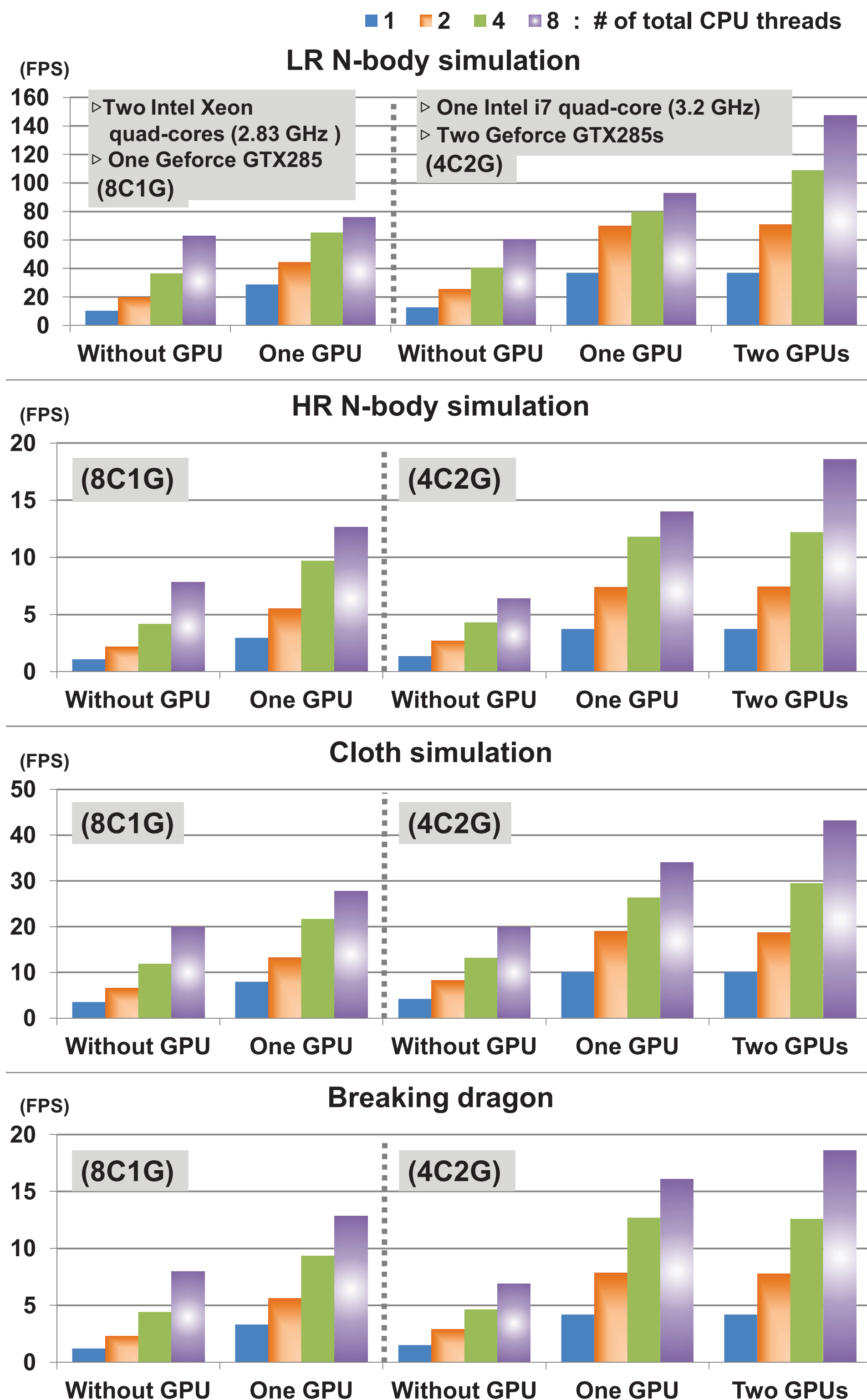


Figure 3.5: *We measure the performance of our HPCCD with four different benchmarks in two machines as we vary the numbers of CPU threads and GPUs. We achieve 10 times to 13 times performance improvements by using the quad-core CPU machine with two GPUs.*

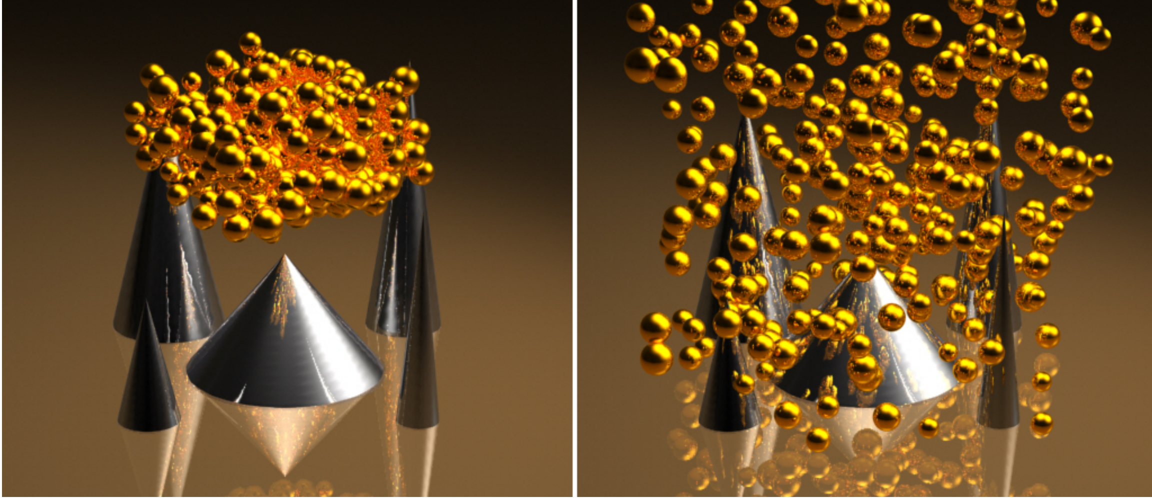


Figure 3.6: *This figure shows two frames during the N-body simulation benchmark with two different model complexities: 34 K and 146 K triangles. Our method spends 6.8 ms and 54 ms on average and achieves 11.4 times and 13.6 times performance improvements for two different model complexities.*

3.4.2 Analysis

Our HPCCD method consists of four components; 1) BV refitting, 2) parallel CCD with low-level and high-level nodes, 3) performing elementary tests, and 4) other serial components and miscellaneous parts (e.g., setup of threads). In the serial version of the HPCCD using only one CPU-core, these four components take about 3%, 19%, 77%, and 1% on average in our tested benchmarks respectively; the results will be reported in the order each component appeared above for the rest of the section. By offloading the elementary tests to GPUs and parallelizing other components except for the serial part, we were able to achieve up to a 13.6 times performance improvement. By using the 4C2G machine, the four components take about 12%, 81%, not-available, and 6.46% respectively. Since the time spent on performing elementary tests in GPUs are overlapped with the CPU-based BVH traversal, we could not exactly measure that component. However, that component takes less than or equal to that of the CPU-based BVH traversal.

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

We also measure the scalability of the third component of performing elementary tests as a function of the number of GPUs used in the 4C2G machine. We perform the CCD by using only a single CPU-core and measure the time, T_e, spent on performing elementary tests after serializing the components of the HPCCD. We then measure how much the overall CCD time is reduced from using a single CPU-core to

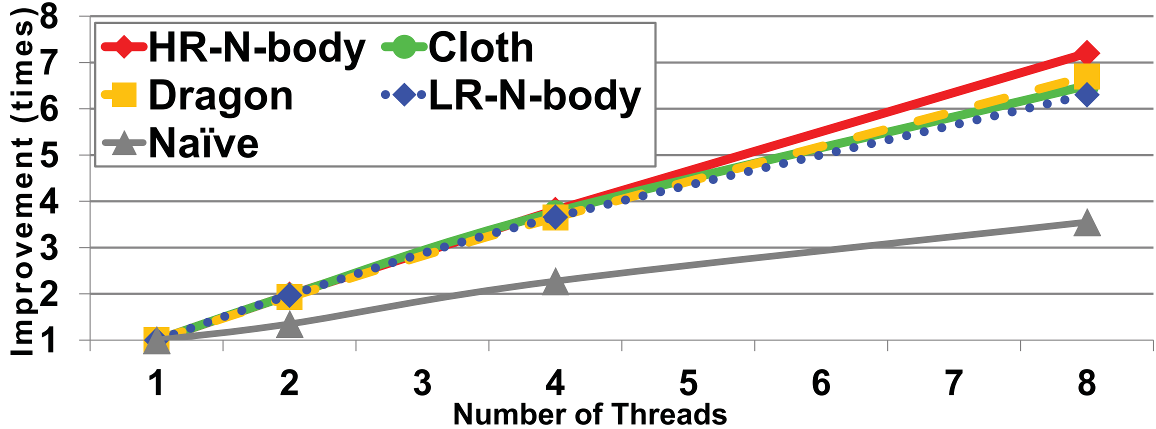


Figure 3.7: *This figure shows the performance improvement of our HPCCD method as a function of the number of CPU-cores without using GPUs over using a single CPU-core. The gray line, marked by triangles, shows an average performance improvement of the naive approach described in Sec. 3.1.2 with all the tested benchmarks.*

using GPUs for the elementary tests. We refer to the reduced time as T_r. The scalability of the third component can be computed by a simple equation, $T_e/(T_e - T_r)$. According to this equation, we achieve 2.8 times and 4.6 times performance improvements for the third component by using one GPU and two GPUs respectively on average in the benchmarks. However, we expect to achieve a higher scalability when using multiple CPUs, since we can generate more elementary tests efficiently and thus utilize GPUs better.

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

Limitation: Our algorithm has certain limitations. Note that the serial part of the HPCCD method takes 6.46% with the 4C2G machine. According to the Amdahl's law [40], we can achieve only 15 times more improvement in addition to the 13 times performance improvement we have achieved by using the 4C2G machine, although we would use unlimited resource of CPUs and GPUs. Also, our method can detect a case when GPUs do not keep up with CPUs and use CPUs to perform elementary tests to achieve a higher performance. However, our current algorithm does not attempt to achieve a higher performance when GPUs are idle. We can implement processing inter-CD task units in GPUs using the CUDA and perform the inter-CD tasks in GPUs when GPUs are idle. However, it requires further research to map the hierarchical traversal well in the streaming GPU architectures. Also, we mainly focus on the efficient handling of large deforming models with consisting of tens or hundreds of thousands of triangles. If the model complexity is small or we have to handles models consisting of a small number of rigid bodies, our method may not get a high scalability since there are not many inter-CD task units that we can parallelize.

3.4.3 Comparisons

It is very hard to directly compare the performance of our method over prior methods. However, most prior approaches use either GPUs [26, 25, 24, 33] or CPUs [20, 21, 23] to accelerate the performance of CCD. One distinct feature of our method over prior methods is that it maps CPUs for the BVH traversal and GPUs for performing elementary tests. Since these two different components, the traversal and elementary tests, are more suitable to CPUs and GPUs respectively, we decompose the computation of CCD in such a way that it can fully exploit the multi-core architectures of CPUs and GPUs. Therefore, our method achieved more than an order of magnitude performance improvement over using a single CPU-core and showed interactive performance for large-scale deforming models.

We compare our method with the current state of the art technique proposed by Sud et al. [33]. This method supports the general polygonal models and CCD including self-collision. We contacted authors of this technique, but we were not able to get the binary of this method. Therefore, we compare results of our method with their results reported in their paper. Note that this comparison is rather unfair, since they used a GeForce 7800, old graphics card. Since their tested benchmarks are different from ours, we measure an average ratio of model complexities of tested benchmarks to the CCD times spent for processing those benchmarks. The ratio of our method is 107 times higher than that of their method. This means that our method can process 107 times bigger model complexity given a unit time or run 107 times faster given a unit model complexity than their method. Also, according to the GPU performance growth data from the GeFroce 7800 to GeForce GTX 280 for 3 years [6], the performance has been improved about 6 times. Therefore, based on this information, we conjecture that our method is about 5 times to 10 times faster than their method even though they would use two GeForce GTX 285 GPUs that our method was tested. Moreover, they reported that the performance of their method is limited by the data read-back performance. Although the performance of their method would have been improved by using a recent GPU, data read-backs has not been much improved in past years (e.g., about 3 times improvement in terms of data bandwidth for three years in the past [6]). They reported that the data read-back from a GPU and other constant costs even for small models span between 50 ms and 60 ms at least, which is even higher than or comparable to the whole computation time of our HPCCD method tested with large-scale deforming models.

We also compare our method with a CPU-based parallel CCD method proposed by Tang et al. [23]. This method also achieved a high performance improvement by using 16 CPU-cores. However, our method achieves about 50% and 80% higher performance with the same tested benchmarks: the cloth simulation and LR N-body simulation respectively. Since the portion of elementary tests is larger in the LR N-body simulation, our hybrid method achieves a higher performance improvement with the LR N-body simulation. Also, according to the Google Product Search [1] and its reported lowest prices for CPUs and GPUs, the price of the 16 CPU-cores (USD 7200) used in [23] is 4.4 times higher than that of the CPU (USD 995) and two GPUs (USD 640) of our 4C2G machine. Therefore, our method achieves about 7 times higher performance per unit cost.

3.5 Conclusion

We have presented a novel, hybrid parallel continuous collision detection method utilizing the multi-core CPU and GPU architectures. We use CPUs to perform the BVH traversal and culling since CPUs

[1] `http://www.google.com/products`, 2009

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

In next chapter, we extend our hybrid parallel framework designed for CCD to other proximity queries with more CPUs and GPUs.

Chapter 4. Scheduling in Heterogeneous Computing Environments for Proximity Queries

In this chapter, we present a novel, Linear Programming (LP) based scheduling algorithm that minimizes the running time of a given proximity query, while exploiting heterogeneous multi-core architectures such as CPUs and GPUs. We first factor out two common and major job types of various proximity queries: hierarchy traversal and leaf-level computation. We then describe a general, hybrid parallel framework, where our scheduler distributes the common proximity computations to the heterogeneous computing resources (Sec. 4.1). In order to represent the performance relationship between jobs and computing resources, we model the expected running time of those computations on the computing resource, by considering setup costs, data transfer overheads, and the amount of jobs. We then formulate our scheduling problem, minimizing the largest time spent among computing resources, as an optimization problem and present an iterative LP-based scheduling algorithm (Sec. 4.2), which shows high-quality scheduling results with a small computational overhead. We further reduce the overhead of our scheduling method by employing a hierarchical scheduling technique, to handle a larger number of independent computing resources.

We have applied our hybrid parallel framework and scheduling algorithm to a wide variety of applications (Sec. 4.3). In the tested benchmarks, we use various combinations of a quad-core CPU, two hexa-core CPUs, and four different types of GPUs. In various machine configurations, our method achieves up to an order of magnitude performance improvement over using a multi-core CPU only. While other tested prior scheduling methods show even lower performance as we add more computing resources, our method continually improves the performance of proximity queries. Moreover, we show that our method achieves throughput that are close (e.g., 75%) to a conservative upper bound of the ideal throughput. In addition, we employ our expected running time model to optimize a work stealing method, that is a general workload balancing approach used in parallel computing systems. The work stealing method optimized by our expected running time model achieves performance improvement (18% on average) over the basic one in the tested benchmarks, while eliminating one of the manually tuned parameters that strongly affect its performance (Sec. 4.3.5).

Our method is a practical, optimization-based scheduling algorithm that aims proximity computation in heterogeneous computing systems. For various proximity quires, our method robustly provides performance improvement with additional computing resources. To the best of our knowledge, such results have been not acquired by prior scheduling algorithms designed for parallel proximity computations. We wish that our work takes a step towards better utilization of current and future heterogeneous computing systems for proximity computations.

4.1 Overview

We give a background on hierarchy-based proximity computation and then describe our hybrid parallel framework.

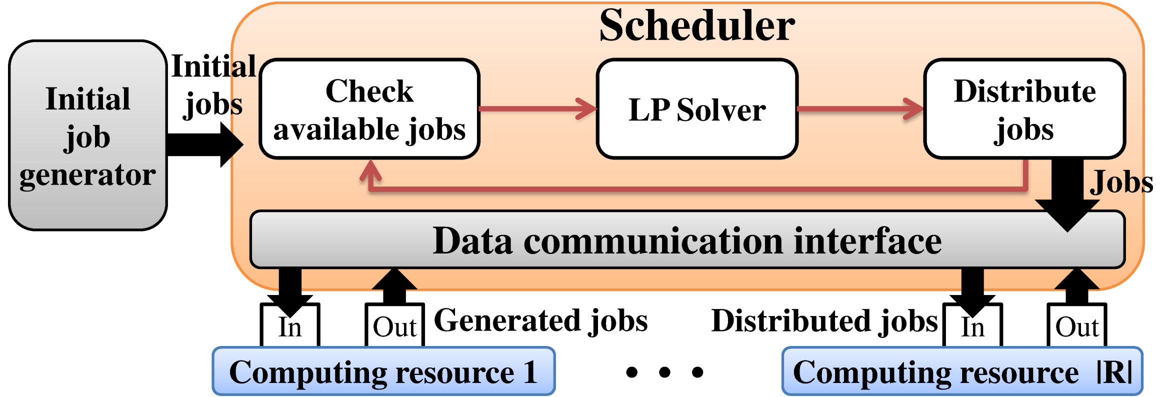


Figure 4.1: *The overview of our hybrid parallel framework*

4.1.1 Hierarchy-based Proximity Computation

Proximity queries are commonly accelerated by using an acceleration hierarchy (e.g., BVHs or kd-trees) constructed from a mesh. For simplicity, we assume that we use BVHs as the acceleration hierarchy for various proximity queries in this chapter; kd-trees or other acceleration hierarchies can be used with our method as well.

To perform proximity queries, we traverse a BVH starting from the root node of the BVH. For each intermediate node, we perform computations based on the bounding volume (BV) associated with the node. Depending on the result of the computations, we traverse the left node, the right node, or both of the nodes in a recursive manner. Once we reach leaf nodes, we also perform other computations based on geometric primitives (e.g., triangles) associated with the leaf nodes. These two different types of computations, *hierarchical traversal* and *leaf-level computation*, are common jobs that can be found in many hierarchy-based proximity computations.

These two components have different characteristics from a computational point of view. The *hierarchical traversal* component generates many computational branches and thus requires random memory access on the mesh and the BVH. Moreover, its workload can vary a lot depending on the geometric configuration between BVs of intermediate nodes. On the contrary, the *leaf-level computation* follows mostly a fixed work flow and its memory access pattern is almost regular. Because of these different characteristics, we differentiate computations of various proximity queries into these two types of jobs. This job differentiation is also critical for modeling an accurate performance model and finding an optimal workload distribution algorithm in heterogeneous computing systems (Sec. 4.2).

4.1.2 Our Hybrid Parallel Framework

Fig. 4.1 shows our overall hybrid parallel framework for various proximity queries. Our hybrid parallel framework consists of four main components: 1) initial job generator, 2) computing resources, 3) scheduler, and 4) data communication interface. Before performing a proximity query, we first share basic data structures (e.g., meshes and their BVHs) among different computing resources, to reduce the data transfer time during the process of the proximity query. Then the *initial job generator* computes a set of initial jobs and feeds it into the scheduler. The *scheduler* distributes initial jobs into *computing resources* (e.g., CPUs or GPUs). The scheduler runs asynchronously in a separate, dedicated CPU thread, while computing resources process their jobs. Data transfer among the scheduler and computing

resources is performed through the *data communication interface*. Each computing resource has two separate queues, incoming and outgoing job queues, as the data communication interface. The scheduler places distributed jobs into the incoming job queues. Then, each computing resource fetches jobs from its incoming job queue and starts to process the fetched jobs. If a computing resource generates additional jobs while processing them, it places new jobs into its outgoing job queue; a hierarchical traversal job dynamically creates multiple jobs of the leaf-level computation depending on geometric configurations. Once there is no job in the incoming job queue of a computing resource, it notifies the scheduler. At each time the scheduler gets such a notification, it collects available jobs from all the outgoing job queues of the computing resources and distributes them into incoming job queues. The main goal of our scheduler is to compute a job distribution that minimizes the makespan.

Implementation details: When a job transfer between different computing resources is required, the transfer is performed through main memory in the machine. If one of the source and the target computing resources is CPU, it is done by only one transaction between main memory and GPU's video memory (or global memory). For communication between two GPUs, jobs are moved to main memory first, and then are transferred to the target GPU's memory, since direct communication between GPUs is only supported under some specific conditions and GPU architectures [67].

In our implementation, we dedicate a single CPU thread, *GPU manager*, to all of GPU computing resources regardless of the number of GPUs. The GPU manager launches kernels and checks states (e.g., idle or busy) of GPU computing resources. The communication between main memory and GPU's global memory is controlled by the scheduler and the GPU manger. The GPU manager has its own communication queue, and the scheduler pushes a request on the communication queue when a data transfer between main memory and GPU memories is required. When the queue has requests, the GPU manager launches data transfer kernels asynchronously through a separate stream that is different from a stream for a job processing kernel [67]. As a result, we can overlap the data transfer and computations and it can hide the communication overhead. Instead of fetching a single job from the incoming queue of a computing resource and processing it, we dequeue multiple jobs and process them with a single invocation of the job processing routine. Specifically, we dequeue jobs that have the same job type. In each job processing routine, we divide the fetched jobs into multiple groups, and launch multiple kernels on different GPU streams for each group at once to overlap data transfer and computation. Thus, we can further minimize the data transfer overhead and optimize the utilization of GPUs.

4.2 LP-based Scheduling

In this section, we describe our formulations of the problem and present our scheduling algorithm.

Notations: We define a *job* to be an atomic unit that our scheduling method handles. To denote each computing resource (e.g., CPU or GPU), we use R_i, where i is the resource ID. R is a set of all R_i and we use $|R|$ to denote the number of available resources. To indicate the type of each job (e.g., hierarchical traversal and leaf-level computation), we use J_j, where j is the job type ID. J is a set of all J_j and $|J|$ refers to the number of different job types. We use n_j to denote the number of jobs that have a particular job type J_j, while we use n_{ij} to denote the number of jobs of J_j allocated to R_i. We also use the term *makespan* to denote the largest running time spent among all the computing resources.

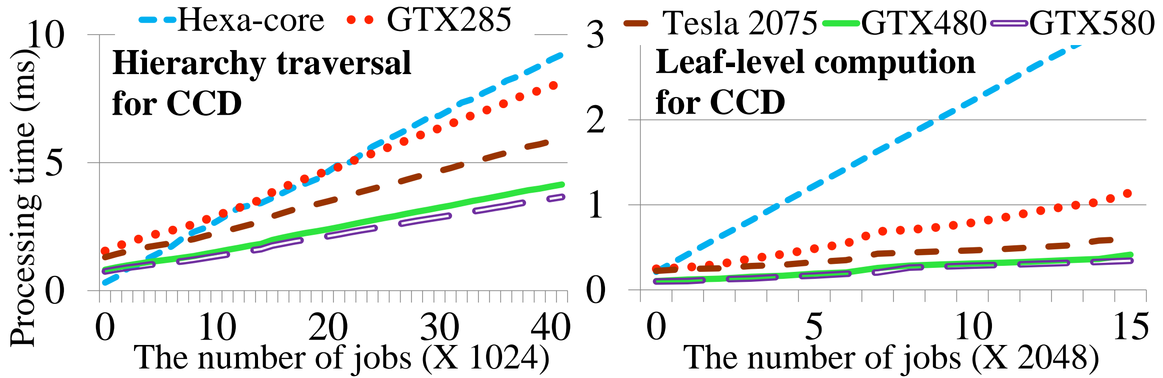


Figure 4.2: *This figure shows observed processing time of two different job types on four different computing resources as a function of the number of jobs.*

4.2.1 Expected Running Time of Jobs

Since many factors on both computing resources and computations of proximity queries influence the performance, it is hard to consider all the factors separately when we decide a job distribution. To abstract such complex performance relationship and to model our scheduling problem as a mathematical optimization problem, we propose to formulate an expected running time of processing jobs on a particular computing resource.

To formulate the expected running time of jobs, we measure how much time it takes to process jobs on different computing resources. As shown in Fig. 4.2, an overall trend is that the processing time of jobs on each computing resource linearly increases as the number of jobs increases. However, the performance difference among computing resources varies depending on characteristics of each job type. For leaf-level computation a GPU shows much higher performance than a multi-core CPU, while a multi-core CPU shows a little lower or similar performance with a GPU for hierarchical traversal. To efficiently utilize heterogeneous computing resources, we need to consider these relationship between job types and computing resources. Another interesting observation is that a certain amount of setup cost, especially higher for GPUs, is required to launch a module (or kernel for GPU) that processes at least a single job on computing resources.

To fully utilize the available computing power, it is also important to minimize communication overheads between computing resources. For example, two identical computing resources show different processing throughput for a same job depending on which computing resource generates the job since another one needs to wait for data transfer. In order to accommodate such data transfer overhead, we differentiate the types of jobs depending on which computing resource those jobs were created at, even though they perform the same procedure (e.g., the hierarchical traversal created from a CPU or a GPU).

We reflect these observations in our formulation of the expected running time of jobs. More specifically, given n_{ij} jobs with a job type J_j that are created at a computing resource R_k, the expected time, $T(k \rightarrow i, j, n_{ij})$, of completing those jobs on a computing resource R_i is defined as the following:

$$T(k \rightarrow i, j, n_{ij}) = \begin{cases} 0, & \text{if } n_{ij} \text{ is } 0 \\ T_{setup}(i,j) + T_{proc}(i,j) \times n_{ij} & \\ +T_{trans}(k \rightarrow i, j) \times n_{ij}, & \text{otherwise.} \end{cases} \quad (4.1)$$

$T_{setup}(i,j)$ represents the minimum time, i.e. *setup cost*, required to launch a module that processes

unit (μs)	Hexa-core CPU		GTX285		Tesla2075		GTX480		GTX580		T_{trans}	
	T_{setup}	T_{proc}	T_{setup}	T_{proc}	T_{setup}	T_{proc}	T_{setup}	T_{proc}	T_{setup}	T_{proc}	**C↔G**	**G↔G**
Traversal	11.07	0.22	260.5	0.19	242.2	0.14	135.6	0.10	125.0	0.09	0.003	0.01
Leaf-level	2.77	0.10	81.9	0.03	136.3	0.013	43.0	0.01	42.02	0.01	0.002	0.003

Table 4.1: *This table shows constants of our linear formulation computed for continuous collision detection.*

the job type J_j on the computing resource R_i. $T_{proc}(i, j)$ is the expected processing time for a single job of J_j on the R_i, while $T_{trans}(k \rightarrow i, j)$ is the transfer time of the data about a single job of J_j from R_k to R_i. In the rest of this thesis, we simply use $T(i, j, n_{ij})$ instead of $T(k \rightarrow i, j, n_{ij})$ since we differentiate the types of jobs depending on the producer resources and the job type J_j inherently contains the information.

Measuring constants: In order to accurately measure constants $T_{setup}(\cdot)$, $T_{proc}(\cdot)$, and $T_{trans}(\cdot)$ of Eq. (4.1), we measure the running time of performing jobs in each computing resource and the transfer time between computing resources, as we vary the number of jobs. We then fit our linear formulation with the observed data, and compute the three constants for each job type on each computing resource by using sample jobs. This process is performed at a pre-processing stage and takes a minor time (e.g., a few seconds). Computed constants for our linear formulation in one of our tested machines are in Table 4.1. The constants are measured for each proximity query, not for each experiment.

Our formulation of the expected running time shows linear correlations, which range from 0.81 to 0.98 (0.91 on average) with the observed data. This high correlation validates our linear formulation for the expected running time of jobs.

4.2.2 Constrained Optimization

There are many different ways of distributing jobs into available computing resources. Our goal is to find a job distribution that minimizes the makespan.

We run our scheduler when no more jobs are left in the incoming job queue of a computing resource. When the scheduler attempts to distribute unassigned jobs, some computing resources may be busy with processing already assigned jobs. Therefore, our scheduler considers how much time each computing resource would spend more to finish all jobs allocated to the resource. We use $T_{rest}(i)$ to denote such time for each computing resource R_i. We estimate $T_{rest}(i)$ as the difference between the expected running time of the jobs on R_i computed based on Eq. (4.1) and the time spent on processing the job so far; if we already have spent more time than its expected running time to process the jobs, we re-compute the expected running time of the computing resource with remaining jobs.

We formulate the problem of minimizing the makespan for performing a proximity query, as the following constrained optimization:

$$Minimize\ L,$$

$$subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i, j, n_{ij}) \leq L, \forall i \in R \quad (4.2)$$

$$\Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J \quad (4.3)$$

$$n_{ij} \in \mathbb{Z}^{+} (zero\ or\ positive\ integers). \quad (4.4)$$

This optimization formulation leads to find values of n_{ij} that minimize L under the three constraints, from Eq. (4.2) to Eq. (4.4). The first constraint (Eq. (4.2)) defines L as the makespan. The second constraint of Eq. (4.3) makes sure that there is neither missing nor redundant jobs. Finally, the third constraint (Eq. (4.4)) ensures that the result values of n_{ij} are restricted to zero or positive integer numbers.

4.2.3 Scheduling Algorithm

Our optimization formulation falls into the category of minimizing the makespan, which is known as NP-hard. To design an efficient scheduling algorithm, we first remove the integer constraint (Eq. (4.4)) for the values of n_{ij}. Instead, we allow a floating value for n_{ij} and choose an integer value that is closest to the floating value. We found that this relaxation affects very little (less than 1%) to the quality of scheduling results, since we have hundreds of thousands of jobs on average across all the tested benchmarks. With this relaxation and if we do not consider setup costs, we can solve the optimization problem in a polynomial time by using linear programming (LP) [68]. When we consider setup costs, our optimization formulation becomes a piece-wise linear function. If $n_{ij} = 0$, the setup cost $T_{setup}(i, j)$ should be zero. Otherwise, the setup cost can have a non-zero value. Thus, our formulation becomes a piece-wise LP problem, which has been known as NP-hard as well [69].

Instead of checking all the possible cases ($2^{|R||J|}$) of distribution of job types into computing resources, we present an iterative LP solver that checks only $|R||J|$ distribution cases. Our scheduling algorithm has the following two main steps: 1) initial assignment and 2) refinement steps.

Initial assignment step: Assume that we always have setup costs in the expected running time formulation (Eq. (4.1)). By running the LP solver, we compute a job distribution that gives a smallest L, given the assumption. However, we observe that the initial assignment can have larger L than optimal solution because of the relaxation to the piece-wise condition of setup costs. This can result in inefficient utilization of computing resources.

As a simple example, assume that we have only two job types (J_1, J_2) and the same number of jobs for both job types, i.e. $n_1 = n_2 = 100$. Also, we have two computing resources (R_1, R_2) that have identical capacities and show the same performance for both job types, i.e. $T_{proc}(i, 1) = T_{proc}(i, 2) = 0.01s$, but setup costs are different:

$$\begin{aligned} R_1: &\quad T_{setup}(1,1) = 2s, \quad T_{setup}(1,2) = 0s \\ R_2: &\quad T_{setup}(2,1) = 0s, \quad T_{setup}(2,2) = 2s. \end{aligned}$$

In the initial assignment step, the LP solver assumes that all the computing resources have setup costs for all the job types irrespective of the number of jobs. The LP solver, therefore, considers that the setup cost is same (i.e. two seconds) for both computing resources, and distributes the same number of jobs to both computing resources regardless of job types. As a result, the example parallel system consisting of R_1 and R_2 takes more than two seconds, since each computing resource already takes two seconds for its setup cost. However, if we allocate all the jobs of J_2 to R_1 and all the jobs of J_1 to R_2, both computing resources do not incur setup costs and thus we can complete all the jobs in one second.

Refinement step with an application-independent heuristic: To address the under-utilization issue of computing resources in the initial assignment step, we iteratively improve assignment solutions in refinement steps. Since we relax the piece-wise condition of setup costs in the initial assignment, we

consider its negative effects and re-assign jobs to reduce such negative effects. To perform this strategy, we define a *job-to-resource ratio* (n_{ij}/n_j) for each job type. Given a job type, this ratio describes the portion of jobs that are being processed on the resource R_i. The ratio can be an approximate indicator of the benefit obtained by using R_i to process jobs of J_j, since the overhead of the setup cost is constant and thus amortized when the number of jobs increases.

We treat a computing resource that has the smallest job-to-resource ratio to be most under-utilized given a job type. If there are multiple candidates, we choose the one that has a larger setup cost than the others. We thus re-assign jobs of the job type assigned to the computing resource to other computing resources. To implement this heuristic within our LP-based scheduler, we set $T_{setup}(i,j)$ as zero and $T_{proc}(i,j)$ as ∞ for the R_i that has the smallest job-to-resource for j_j. Note that even though R_i does not get any jobs given the job type J_j, it can get more jobs of other job types. As a result, it can better utilize different capacities of heterogeneous computing resources.

We perform the refinement step until one of the following three conditions are met: 1) the LP solver cannot find a solution, 2) we cannot further reduce the L value, the makespan, or 3) the LP-solver takes more time than the smallest value of $T_{rest}(i)$, the expected running time for completing tasks that are under processing in each resource among all the resources, to prevent a long idle time of computing resources (i.e. *time-out condition*). Through this iterative refinement steps, we can achieve better assignment results that are close to the optimal solution.

Note that this is an application-independent heuristic, which can be used in various different proximity queries. In addition to this heuristic, we can also have query-dependent heuristics for a particular proximity query.

4.2.4 An example work-flow of our iterative LP solver

We show an example work-flow of our scheduling method, which is based on an iterative LP solver.

Assume that we have only two job types (J_1, J_2) and have two times more jobs for J_1 than jobs of J_2 (i.e. $n_1 = 200$, $n_2 = 100$). Suppose also that we have three computing resources (R_1, R_2, R_3) that have identical capacities and show the same performance for both types of jobs, i.e. $T_{proc}(i,1) = T_{proc}(i,2) = 0.01s$ where i is a computing resource index, but their setup costs are different:

$$\begin{aligned}
R_1: &\quad T_{setup}(1,1) = 1s, \quad T_{setup}(1,2) = 0s \\
R_2: &\quad T_{setup}(2,1) = 0s, \quad T_{setup}(2,2) = 1s. \\
R_3: &\quad T_{setup}(3,1) = 0s, \quad T_{setup}(3,2) = 4s.
\end{aligned}$$

In the initial assignment step, the LP solver assumes that all the computing resources have setup costs for all the job types irrespective of the number of jobs. The LP solver, therefore, considers that the setup cost is same (i.e. one second) for two computing resource, R_1 and R_2, while the setup cost for R_3 is four seconds. Since the setup cost of R_3 (i.e. four seconds) is larger than the cost of processing all the jobs in other computing resources, the LP solver does not assign any jobs to R_3. Instead, the LP solver distributes the same number of jobs to R_1 and R_2 regardless of job types (Table 4.2).

However, since R_3 does not get any jobs, it actually does not take any setup cost and will be idle, even though other computing resource are busy with processing assigned jobs. Also, R_1 and R_2 will take more than two seconds, since each computing resource already takes one second for its setup cost and

Res.	n_{i1} (n_{i1}/n_1)	n_{i2} (n_{i2}/n_2)	Expected running time
R_1	100 (0.5)	50 (0.5)	2.5 sec.
R_2	100 (0.5)	50 (0.5)	2.5 sec.
R_3	0 (0.0)	0 (0.0)	0.0 sec.

Table 4.2: *An assignment result of the initial assignment step.*

consumes one and half seconds for processing jobs. However, all the jobs can be done in two seconds, if we allocate all the jobs of J_2 to R_1 and J_1 to R_2 respectively.

In the first iteration of the refinement step, our iterative LP solver chooses R_3 to re-assign its jobs of J_2, since its job-to-resource ratio is zero and its setup cost (four seconds) is larger than others. We then re-run the LP solver under the constraint that no jobs of J_2 can be given to R_3, and an assignment result shown in Table 4.3 can be achieved. Even though R_3 does not waste its capacities, R_1 and R_2 processes jobs inefficiently, because of its incorrect calculations of setup costs. Since we achieve lower *makespan,* L (1.67 seconds) than the initial solution (2.5 seconds), we invoke one more iteration.

Res.	n_{i1} (n_{i1}/n_1)	n_{i2} (n_{i2}/n_2)	Expected running time
R_1	16 (0.08)	50 (0.5)	1.66 sec.
R_2	17 (0.085)	50 (0.5)	1.67 sec.
R_3	167 (0.685)	-	1.67 sec.

Table 4.3: *The assignment result of the first iteration.*

4.2.5 Analysis

At the worst case, our iterative solver can perform up to $O(|R||J|)$ iterations, since it can assign all the jobs into only one computing resource. In practice, however, our LP-based iterative scheduling algorithm takes only a few iterations. When $|R|$ and $|J|$ are 6 and 12 respectively, our methods runs 7.5 iterations on average in our experiments. Each iteration of our LP-based scheduling method takes only 0.3 ms on average. Also, the expected running time is reduced up to 59% (19% on average) by the refinement step over the initial solution of the initial assignment step. Table 4.4 shows the benefit of our iterative solver. As we will see later our method achieves higher improvement from the initial solution through refinement iterations, as a heterogeneous level of computing resources increases.

We have also studied the quality of our scheduler by comparing its quality over the optimal result computed with the exhaustive method. In the exhaustive method, we check all the possible ($2^{|R||J|}$) assignments of job types into computing resources and find the job distribution that gives the smallest expected running time. We run the exhaustive method only for the configuration of $|R| = 3$ because of the high computational overhead. We found that our scheduler has a minor computational overhead (e.g. less than 1 ms) and compute a job distribution that achieves a near-optimal expected running time, which is on average within 6% from the optimal expected running time computed with the exhaustive method that takes 30 sec.

Hierarchical scheduling: Even though the computational overhead of our LP solver is low with a small number of computing resources, it increases with $O(|R|^2|J|^3)$ theoretically in the worst case [70]. However, we found that it increases almost linearly as a function of the number of resources in practice.

	With hierarchical				Without hierarchical			
# of Res. ($\|R\|$)	**3**	**4**	**5**	**6**	**13**	**14**	**15**	**16**
# of job types	6	8	10	12	26	28	30	32
# of iterations	4.3	5.5	6.6	7.5	29.9	35.0	35.5	38.5
Time/Iter. (ms)	0.16	0.20	0.25	0.30	0.93	0.99	1.04	1.40
Avg. L_{fin}/L_{init}	0.94	0.90	0.91	0.81	0.96	0.92	0.90	0.88
Min. L_{fin}/L_{init}	0.53	0.60	0.62	0.41	0.65	0.63	0.71	0.74

Table 4.4: *This table shows the average number of iterations in the refinement step and the average time of an iteration. We also compare the quality of the iteratively refined solution (makespan, L_{fin}) with the initial solution (L_{init}) computed from initial assignment step. In this analysis, for each configuration of $|R|$, we run our algorithm for five hundred of randomly generated job sets with the constants in Table 4.1. We add four different GPUs to two hexa-core CPUs one by one as $|R|$ is increased. To focus more on showing benefits of our iterative solver, we turn off the time-out condition in the refinement step in this experiment.*

Nonetheless, the overhead of our scheduling algorithm becomes a non-negligible cost in interactive applications when we employ many resources, since the number of iterations is also increased linearly. For example, if we do not terminate the refinement step by the time-out condition, the overhead becomes 29 ms on average, when we have sixteen computing resources (Machine 2 in Table 4.5) for a collision detection application.

Interestingly, we found that simple workload balancing methods designed for identical parallel systems comparably work well or even better than our LP-based method for identical cores in a device. It is due to simplicity of the methods and low inter-core communication cost under shared memory systems. Multiple cores in a multi-core CPU is a typical example. Based on this observation, we group computing units in a device (e.g., cores in a single multi-core CPU) as a big computing resource and treat it as one computing resource for our LP-based scheduling; we measure the expected running time of different job types with the big computing resource and use that information for our LP-based scheduling. Once tasks are allocated to the one big resource, we run a simple scheduling (e.g., work stealing) method. We found that this two-level hierarchical scheduling method improves the performance of proximity queries in tested benchmarks 38% on average over running without the hierarchical scheduling in Machine 2 (Table 4.5).

4.3 Results and Discussions

We have implemented our hybrid parallel framework and scheduling algorithm (*Ours(Exp.+LP)*) in three different machine configurations (Table 4.5). As we discussed in the hierarchical scheduling method (Sec. 4.2.5), we treat the identical computing units in a device as a single computing resource. We then use simple work stealing method [52] within a multi-core CPU, and even distribution method [27] within a GPU for tasks allocated to the single computing resource. In the case of using two hexa-core CPUs and four GPUs together, we have six different computing resources (i.e. $|R| = 6$). Initially, we have two different job types (hierarchy traversal and leaf-level computation) for all the tested proximity queries. We differentiate these two job types depending on which computing resource generates such type of jobs (Sec. 4.2.1). Therefore, $|J|$ becomes 12.

Res.	Machine 1	Machine 2	Machine 3
1	Quad-core CPU	Hexa-core CPU	Hexa-core CPU
2	GeForce GTX285	Hexa-core CPU	Hexa-core CPU
3	GeForce GTX480	GeForce GTX285	GeForce GTX480
4		Tesla 2075	GeForce GTX480
5		GeForce GTX480	GeForce GTX480
6		GeForce GTX580	GeForce GTX480

unit (FPS)	**Cloth** (Fig. 1.1(a))	**N-body** (Fig. 3.6)	**Fract.** (Fig. 3.4)	**Path tracing** (Fig. 1.1(b))	**Motion** (Fig. 1.1(c))
Quad-CPU	9.75	3.80	3.60	0.004	0.25
Hexa-CPU	10.32	3.96	3.59	0.005	0.39
GTX285	21.91	11.17	8.23	0.006	0.12
Tesla2075	39.06	19.93	14.39	0.010	0.42
GTX480	58.76	29.39	21.43	0.015	0.59
GTX580	64.64	32.41	23.63	0.020	0.71

Table 4.5: *The upper table shows three different machine configurations we use for various tests. The quad-core CPU is Intel i7 (3.2GHz) chip and each hexa-core CPU is Intel Xeon (2.93GHz) chip. The bottom table shows the throughput of each computing resource for the tested benchmarks.*

We use the axis-aligned bounding box as bounding volumes (BVs) for BVHs because of its simplicity and fast update performance. We construct a BVH for each benchmark in pre-computation time in a top-down manner [1]. For dynamic scenes, we simply refit BVs at the beginning of every new frame [19]. The hierarchy refit operation takes a minor portion (e.g., about 1%) of the whole proximity query computation. Therefore, we just use the fastest computing resource in the machine to update BVHs rather than parallelizing the operation. We have implemented CPU and GPU versions of hierarchical traversals and leaf-level computations based on prior CPU and GPU implementations [27, 52]. We use the OpenMP library [65] and CUDA [6] to implement CPU- and GPU-based parallel proximity queries respectively. Also, we use the LINDO LP[1] solver for our LP-based scheduling algorithm.

For comparison, we have implemented three prior scheduling methods designed for proximity queries. The first one (*Lau10*) is a scheduling algorithm proposed by Lauterbach et al. [27]. When the level of workload balance among computing resources is low, this scheduling method distributes available jobs into computing resources evenly in terms of the number of jobs. The second method is the block-based round-robin method (*Tan09*), proposed by Tang et al. [23]. Also, we have implemented a work stealing (*WS*) algorithm [52] as the third method. In *WS*, once a computing resource becomes idle, it steals a portion (e.g., half) of the remaining jobs from a busy computing resource (victim). To further optimize the implementation of *WS*, we allocate initial jobs according to the relative capacity of different computing resources computed by our expected running time formulation. We also employ the priority based stealing strategy. We give high priority to leaf-level jobs for GPUs and hierarchical traversal jobs for Multi-core CPUs while accounting the characteristics of jobs and computing resources (Sec. 4.2.1). For *Tan09* and *WS* methods, we found that the block size and stealing granularity are strongly related with the performance. We have tested various block sizes (e.g., from 1K jobs to 20K jobs) for *Tan09* and

[1]LINDO systems (http://www.lindo.com)

stealing granularity (e.g., 30-70% of remaining jobs of the victim) for *WS*. Then, we reported the best result among them for each benchmark on each resource combination. Note that while the best results of these techniques are manually acquired, we do not perform any manual tuning for our method.

Benchmarks: We have applied our hybrid parallel framework and its scheduling algorithm into the following five benchmarks that have different characteristics and use different proximity queries. Three of our benchmarks are well-known standard benchmarks[2]: cloth simulation (92 K triangles, Fig. 1.1(a)), N-body simulation (146 K triangles, Fig. 3.6), and fracturing simulation (252 K triangles, Fig. 3.4). For these benchmarks, we perform continuous collision detection and find all the inter- and intra-collisions. Our fourth benchmark is a roadmap-based motion planning problem (137 K triangles, Fig. 1.1(c)) [71], where we attempt to get a sofa out of a living room. We use discrete collision detection for the benchmark. However, this query does not need to identify all the contacts, and thus terminates right away when we find a single collision. We generate 50 K random samples in its configuration space. Our final benchmark is a path tracing where a living room (436 K triangles, Fig. 1.1(b)) is ray traced with incoherent rays generated by a Monte Carlo path tracer [72]. For the benchmark, we generate 80 million rays in total.

Work granularity: Atomic scheduling granularities for leaf-level computation are a triangle-triangle overlap test for the first four benchmarks and a ray-triangle intersection test for the path tracing benchmark. These leaf-level jobs are dynamically generated, when we reach leaf nodes during the hierarchical traversal. For collision queries, a pair of nodes of BVHs is an atomic unit for the hierarchical traversal job. For a pair of two nodes, we perform a sub-tree traversal that starts with performing a bounding volume (BV) overlap test between the two nodes and traverse hierarchies recursively depending on the overlap test result until we reach leaf nodes. In the motion planning benchmark, we need to check whether a randomly generated configuration sample is in a free space or not. This operation is essentially the same as collision detection, and thus we use similar traversal jobs to collision detection. For the path tracing benchmark, an atomic traversal job consists of a ray and the root of a BVH. We perform ray-BV intersection tests in a recursive way until reaching a leaf node. In all the tested benchmarks, each leaf node has a single primitive (e.g., a triangle). If we have multiple primitives at a leaf node, a pair of leaf nodes generates multiple atomic leaf-level jobs. Since our framework and scheduling algorithm are independent to the number of generated jobs, our methods are also compatible with other types of hierarchies that have multiple primitives at leaf nodes.

Initial jobs: To generate initial jobs for our scheduling methods, we identify jobs that are independent and thus can be parallelized naively. These initial jobs can be constructed in an application-dependent manner. In the motion planning and path tracing benchmarks, independent rays and samples are set as initial jobs. For all the other benchmarks, we traverse the BVH of each benchmark in a breadth-first manner and set independent collision detection pairs as initial jobs, as suggested by prior parallel methods [27]. This step takes less than 1 ms in the tested benchmarks.

4.3.1 Results and Analysis

Fig. 4.3 and Fig. 4.4 show the performance of various proximity queries that are parallelized with different scheduling methods in two machine configurations (Machine 1 and 2 in Table 4.5). We use the line plot instead of bar graph in order to highlight the performance trend as we add more computing

[2]UNC dynamic benchmarks (http://gamma.cs.unc.edu/DYNAMICB)

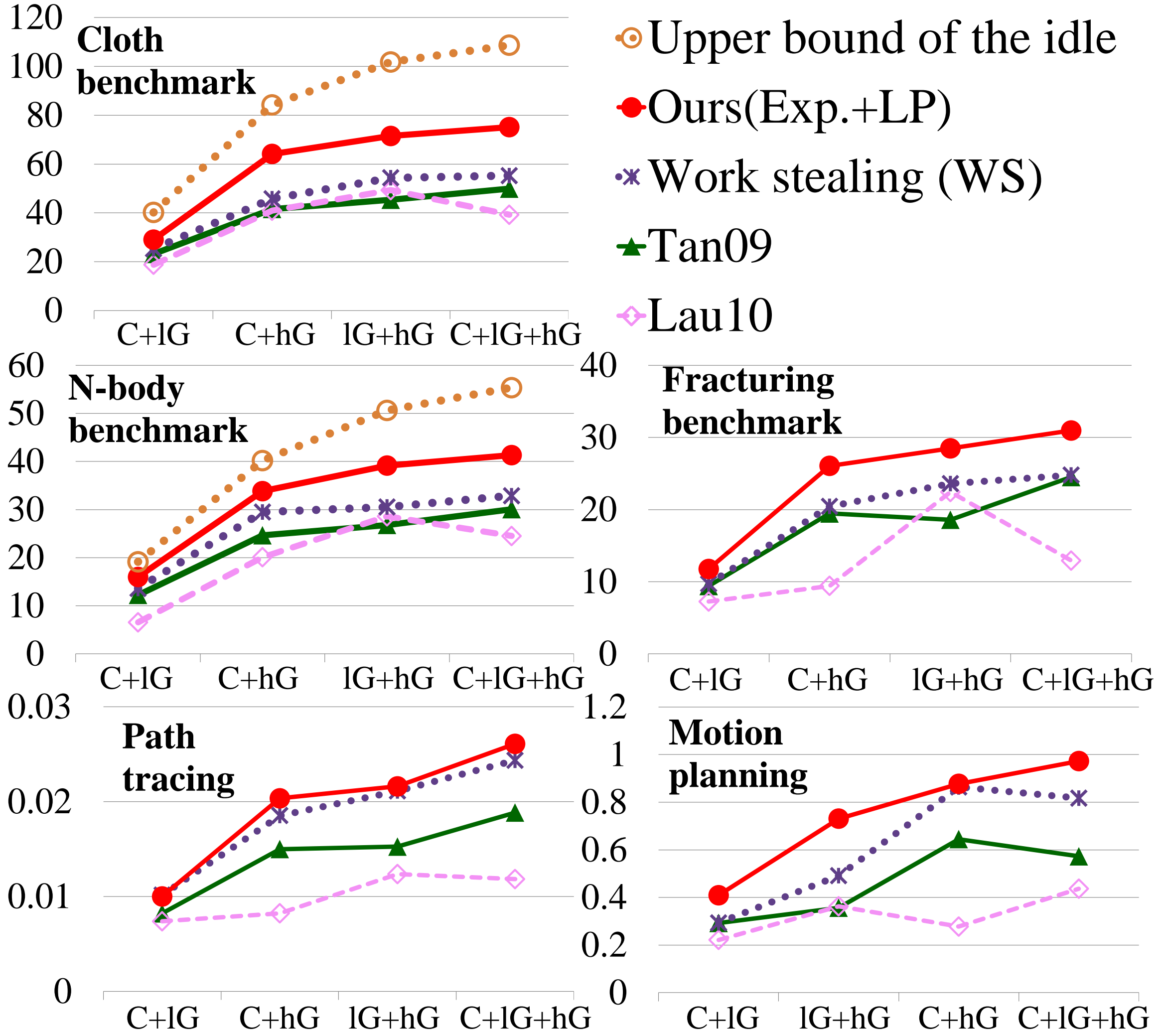


Figure 4.3: *This figure shows the throughput, frames per second, of our hybrid parallel framework, as we add a CPU and two GPUs, in the tested benchmarks. (**C** = a quad-core CPU, **lG** = GTX285 (i.e. low-performance GPU), **hG** = GTX480 (i.e. high-performance GPU))*

resources. For the graph, we measure processing throughput (i.e. frames per second). In order to see how the performance of each query behaves with various combinations of computing resources, we measure the capacity of each computing resource (Table 4.5), and combinations of these computing resources in various ways.

On average, our scheduling method shows higher improvement over all the other prior methods. A more interesting result is that as we add more computing resources, our LP-based method continually improves the performance across all the tested benchmarks. This demonstrates the robustness of our method. Compared to the result achieved by using only a hexa-core CPU, our method achieves up to 19 times improvement by using two hexa-core CPU and four different GPUs (Fig. 4.4).

On the other hand, all the other methods often show even lower performance for additional computing resources, especially when we add lower capacity computing resources. For example, *Lau10* shows significantly lower performance (42%), when we use an additional quad-core CPU (**C**) to GTX285 and GTX 480 (**lG+hG**) system for the fracturing benchmark (Fig. 4.3). In this case, **C** has relatively lower capacity than other GPUs. However, *Lau10* does not consider the relative capacity difference and as-

signs jobs evenly, which leads to a lower performance. Surprisingly, *Tan09*, an efficient version of the round-robin scheduling that naturally considers different capacities of computing resources, also shows lower a performance, when we add **lG** to **C+hG** in the motion planning benchmark (Fig. 4.3). This lower performance is mainly caused by their lack of mechanism for considering different running times of jobs.

The *WS* shows a relatively stable performance compared with other two prior approaches. However, it also gets a lower performance (6%) when we use an additional **lG** to **C+hG** for the motion planning benchmark (Fig. 4.3). On average, our LP-based algorithm shows 22% and 36% higher performance than *WS* in Fig. 4.3 and Fig. 4.4 respectively. We found that since *WS* does not consider the relative capacity of heterogeneous computing resources, stealing operations occur more frequently than in homogeneous computing systems. In addition, communication cost in distributed memory systems is much higher than in shared memory systems [73]. Such a large number of stealing operations and high data communication overhead lower the utilization of heterogeneous computing systems. In our continuous collision detection benchmarks, the work stealing method launches 11 times more data transfer operations on average than our LP-based algorithm when we use six computing resources. Interestingly, for the path tracing benchmark, *WS* shows comparable performance with *Ours(Exp.+LP)*. It is due to the fact that the communication cost is relatively smaller than the large computation time of proximity query for the benchmark. Nonetheless, *Ours(Exp.+LP)* achieves better performance over *WS*, even though we manually calibrate the stealing granularity of *WS* for each combination of resource configuration and benchmark.

Scalability: In Fig. 4.4, all the scheduling methods generally achieve higher throughputs with more computing resources, since we intentionally add more powerful resources. Nonetheless, *Ours(Exp.+LP)* shows the highest throughput among all the other prior scheduling methods in heterogeneous computing systems. Also, the performance gap between ours and prior methods becomes larger as the computing system has higher heterogeneity. Note that heterogeneity is increased as we employ more computing resources, since added computing resources have different capacity from those of prior computing resources. On the other hand, our LP-based method adapts well to high heterogeneity, since it naturally considers the capacity difference of computing resources.

Benefits of LP-based scheduling: To measure benefits caused only by our LP-based scheduling formulation, we implemented a simple scheduling method based on the expected running time of jobs. This simple algorithm assigns jobs according to the relative capacity of different computing resources, while considering our expected running time of jobs. For example, if R_1 is two times faster than R_2 given a particular job type J_i in terms of their expected running time, we then assign two times more jobs to R_1 than R_2. On average, our LP-based scheduling method (*Ours(Exp.+LP)*) shows 26% higher performance over the simple method for the tested benchmarks in Machine 1 and 2. Since the simple method considers only the relative performance of computing resources for each job type and does not look for more optimized job distributions among all the job types, it shows lower performance than the LP-based algorithm. We also measure benefits of considering setup costs (T_{setup}). When we ignore setup costs in our LP-based scheduler, it shows up to 38% (9% on average) performance degradation compared to considering the setup costs in the tested benchmarks.

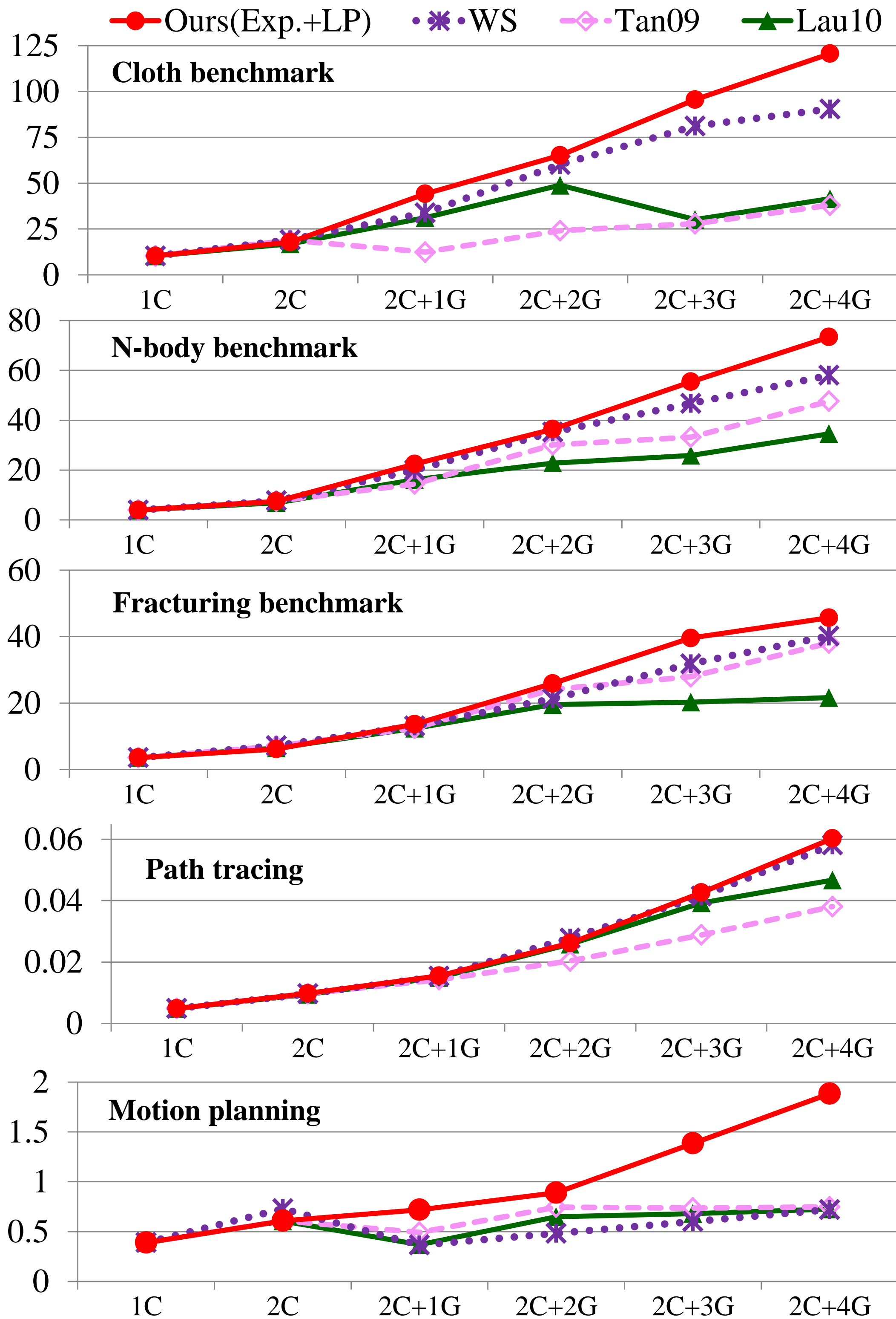


Figure 4.4: *This figure shows the throughput, frames per second, of ours and prior scheduling algorithms, as we add more computing resources. (1C = a hexa-core CPU, 1G = GTX285, 2G = 1G+Tesla2075, 3G = 2G+GTX480, 4G = 3G+GTX580)*

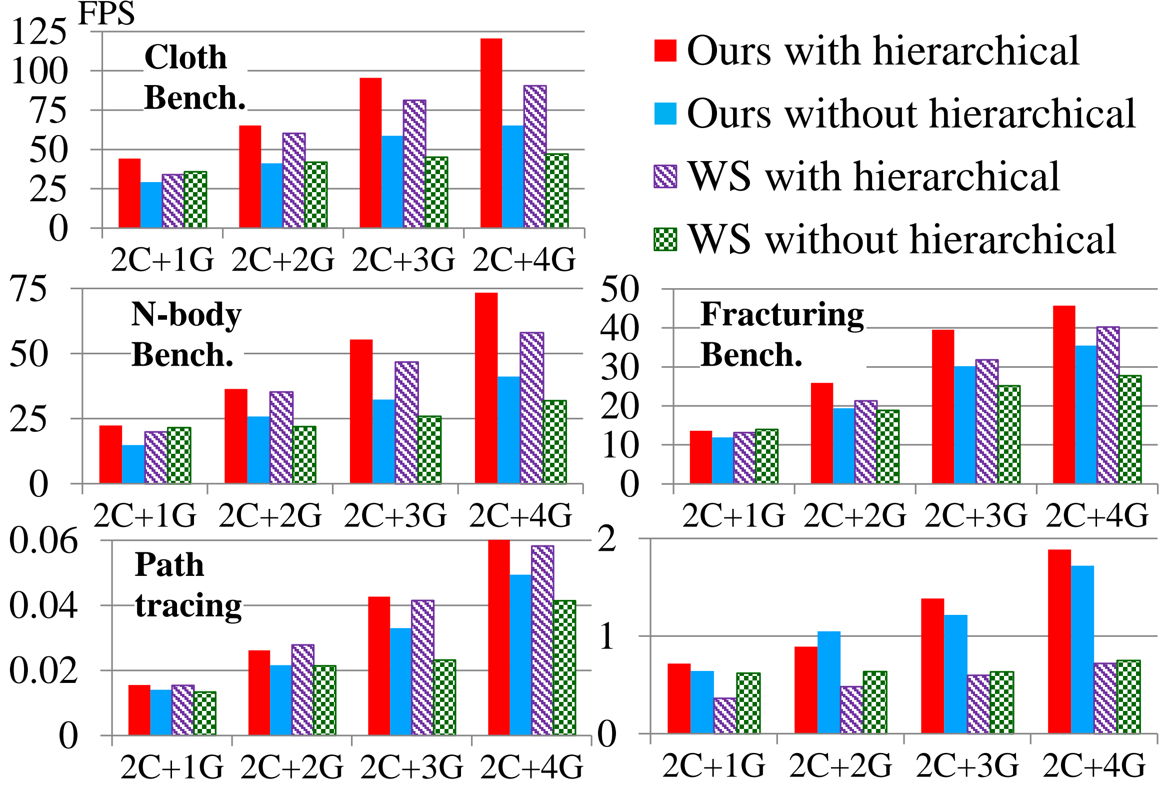


Figure 4.5: *This figure shows frames per second of our LP-based scheduling and work stealing method with/without hierarchical scheduling on Machine 2.*

Benefits of hierarchical scheduling: Fig. 4.5 shows the benefits of our hierarchical scheduling method. By incorporating hierarchical scheduling, our method achieves 35% improvement on average over the one without using hierarchical scheduling. This improvement is caused mainly by two factors. Firstly, the hierarchical approach lowers down the number of resources that we need to consider, and reduces the computational overhead for scheduling. As the computational time for scheduling is decreased, the idle time of computing resources spent on waiting for jobs is reduced and thus we achieve a higher utilization of the computing system. For example, at **2C+4G** our LP-based scheduling takes 1.1 ms for each iteration without hierarchical scheduling ($|R|$=16). It reduces to 0.3 ms when we use hierarchical scheduling ($|R|$=6) and the average portion of idle time of computing resources is decreased by 11% (Table 4.6). Secondly, hierarchical scheduling reduces the data transfer overhead. In our tested benchmarks, the number of data transfer operations is decreased by 30% with our hierarchical scheduling method. Interestingly the hierarchical approach also improves the efficiency of *WS* by 29% on average. Nonetheless, our LP-based scheduling combined with hierarchical scheduling shows a even higher performance.

4.3.2 Optimality

In order to look into the optimality of our method, we compute an upper bound of the ideal scheduling result in terms of real (not expected) running time that we can achieve for the tested proximity queries; it goes beyond the scope of this thesis to derive the ideal scheduling result for our problem, where jobs are dynamically created depending on results of other jobs.

We compute an upper bound of the ideal throughput that can be achieved with multiple heterogeneous computing resources in the following manner. While running a proximity query we gather and

Idle ratio		Cloth	N-body	Fract.	Path	Motion
With Hier.	**2C+1G**	0.133	0.187	0.134	0.001	0.019
	2C+2G	0.161	0.177	0.092	0.010	0.165
	2C+3G	0.205	0.166	0.213	0.007	0.144
	2C+4G	0.227	0.206	0.181	0.021	0.202
W/O Hier.	**2C+1G**	0.084	0.055	0.048	0.059	0.145
	2C+2G	0.222	0.111	0.133	0.059	0.114
	2C+3G	0.350	0.294	0.263	0.077	0.201
	2C+4G	0.433	0.286	0.305	0.128	0.224

Table 4.6: *This table shows the average portions of idle time of computing resources in our LP-based method with/without hierarchical scheduling at Machine 2.*

dump all the generated jobs. Then with all the gathered jobs, we compute the highest throughput by considering all the possible job distributions in an off-line manner. For computing the highest throughput, we ignore all the dependencies among jobs and computational overheads (e.g., data communication and scheduling time); we have computed this upper bound only for the cloth and N-body benchmarks, since computing the upper bound for a benchmark takes a few weeks with our tested machine. Note that it is infeasible for a scheduling method to achieve such upper bound, since it is impossible to exactly predict which jobs will be generated at runtime and we assume that there are no job dependencies to derive the upper bound. As a result, this upper bound is rather loose.

The computed upper bounds are shown for the cloth and N-body benchmarks in Fig. 4.3. For both benchmarks, our method shows throughputs that are within 75% of the performance provided by the upper bounds of ideal throughputs on average. On the other hand, *Lau10*, *Tan09*, and *WS* show within only 45%, 54%, and 61% of the ideal throughput on average, respectively. Note that no prior methods discussed this kind of optimality, and our work is the first to look into this issue and achieves such high throughput close to the ideal throughput.

To see if our method can be improved further, we investigated the under-utilization of each computing resource with our LP-based algorithm. We measured how long computing resource stays in the idle status; a computing resource is defined as idle when it completes all assigned jobs or waits for required data during assigned jobs. We found that the idle time takes a small portion (13% on average) of total running time when we use our LP-based algorithm with hierarchical scheduling (Table 4.6). We also found that the idle time due to data waiting takes less than 10% of whole idle time since we overlap the data transfer and computations. To check the overhead of our scheduler, we measured how much time our scheduler running on the CPU takes, compared to other working threads running on the same CPU; the measured time includes not only the time for solving LP but also communication cost for checking the size of output queues and dispatching the scheduling results to computing resources. It takes about 7% of total running time of those working threads. This indicates that our scheduling method has low computational overhead.

4.3.3 Near-Homogeneous Computing Systems

Although our method is designed mainly for heterogeneous computing systems, we can apply our method for homogeneous computing systems. To check usefulness of our approach even in these sys-

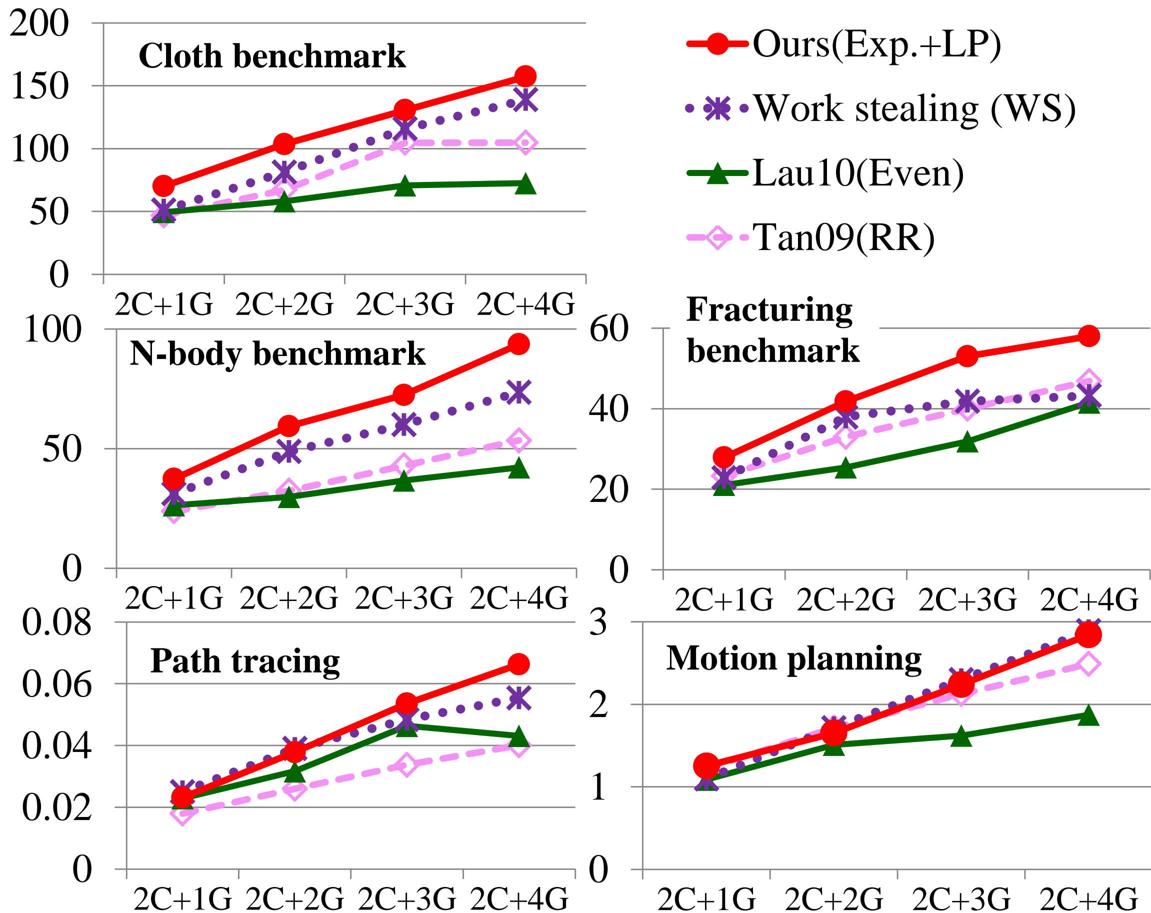


Figure 4.6: *This figure shows the performance of tested scheduling approaches on a near-homogeneous computing system consisting of two hexa-core CPUs and four identical GPUs (Machine 3 in Table 4.5).*

tems, we compare ours and prior approaches in a near-homogeneous system consisting of two hexa-core CPUs and four identical GPUs. Fig. 4.6 shows throughputs with different scheduling algorithms in the near-homogeneous computing system for our tested benchmarks. Prior approaches show better scalability in the near-homogeneous system over in heterogeneous computing configurations. *Tan09*, *Lau10*, and *WS* methods on the near-homogeneous system show improved performance by 11%, 5%, and 10% over the heterogeneous computing respectively, in terms of a relative throughput compared with *Ours(Exp.+LP)*. Nonetheless, our approach still achieves higher throughputs than the prior methods. On average *Ours(Exp.+LP)* shows 39%, 54%, and 12% higher throughputs over the three prior methods, respectively in the tested benchmarks. This result demonstrates the generality and robustness of our LP-based algorithm.

4.3.4 Comparison to a Manually Optimized Method

Only a few works [60] including HPCCD method (Chapter 3) have been proposed to utilize heterogeneous multi-core architectures, such as CPUs and GPUs, in the field of computer graphics. It is very hard to directly compare ours against them in a fair ground. However, these techniques are designed specifically for particular applications (e.g., continuous collision detection or ray tracing). They also assign jobs into either CPUs or GPUs according to manually defined rules (i.e. application-dependent heuristics) that are only valid for a specific application. Unlike these prior works, our method formulates

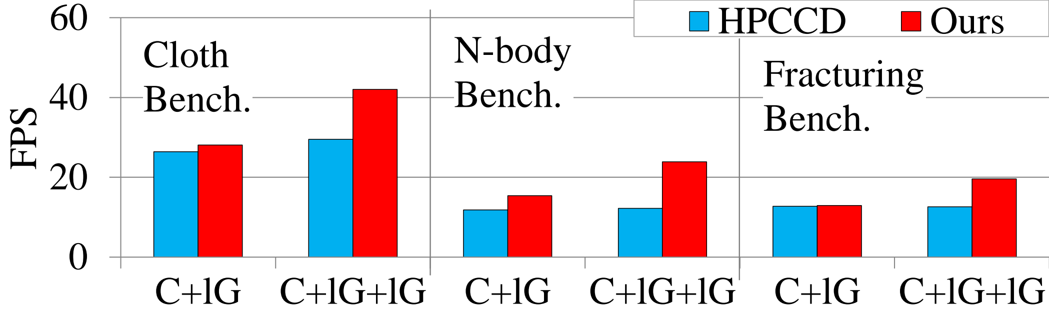


Figure 4.7: *This figure compares the performance of our method with HPCCD, which is optimized specifically for the tested application, continuous collision detection. The throughput, frames per second, includes hierarchy update time.*

the scheduling problem as an optimization problem based on common components of various proximity queries, to achieve wide applicability. Although we have not explored in this thesis, we can also adopt application-dependent heuristics of these prior methods in the refinement step of our LP-based scheduling algorithm, to further improve the performance for a specific application.

We compared the performance of our method over the hybrid parallel continuous collision detection (HPCCD). In Chapter 3, HPCCD is designed specifically for continuous collision detection, by manually assigning jobs to more suitable computing resources (e.g., primitive tests for GPUs). For a fair comparison, we have used the same benchmarks and machine configurations (i.e. a quad-core CPU and two GTX285s) used in their paper. Our method–iterative LP scheduling method without any modification to the application–shows similar or a slightly higher (e.g., 1.3 times higher) performance when we use a GPU with a quad-core CPU. However, when we add one more GPU, our algorithm achieves much higher (e.g., 2 times) performance than HPCCD (Fig. 4.7). This is mainly because our LP-scheduling method considers different capabilities of computing resources and achieves a better distribution result than that computed by HPCCD's application-dependent heuristic. This result further demonstrates the efficiency and robustness of our algorithm, since we achieve even higher performance than the method specifically designed for the application, even though ours is not optimized at all for the application.

4.3.5 Work Stealing with Expected Running Time

In heterogeneous computing systems, the work stealing method requires a large number of stealing operations and high communication overhead as we discussed in Sec. 4.3. It is therefore hard to achieve a high performance with work stealing methods in heterogeneous computing systems.

If each computing resource steals an appropriate amount of jobs from a victim, we can reduce the number of stealing operations and improve the utilization of the heterogeneous computing systems. We found that we can employ one of our contributions, the expected running time formulation, to determine the suitable stealing granularity automatically. In our version of work stealing method, we first calculate the relative capacity among computing resources based on our expected time model for each job type. We then normalize the relative capacities to a range between 0 and 1. Finally, we assign different stealing granularities to computing resources by scaling a basic granularity (e.g., half of remaining jobs in the victim) with the normalized values. At run-time, each computing resource steals jobs from a victim according the assigned stealing granularity.

We found that our method decreases the number of data transfer by 71% on average compared with

the basic work stealing method when we use six computing resources. As a result, our work stealing method achieves 11%, 20%, and 23% higher performance on average in Machine 1, 2, and 3 (Table 4.5) respectively over the basic work stealing method. Also, in the near-homogeneous computing system (Machine 3) it shows compatible performance (0.6% higher on average) with our LP-based method. This result shows the generality and a wider applicability of our expected running time formulation. Nonetheless, in heterogeneous computing systems (Machine 1 and 2), our LP-based method achieves up to 45% (12% on average) higher performance than our version of work stealing method.

4.4 Conclusion

In this chapter, we have presented a novel, LP-based scheduling method, in order to maximally utilize more widely available heterogeneous multi-core architectures. To achieve wide applicability, we factored out common jobs of various proximity queries and formulate an optimization problem that minimizes the largest time spent on computing resources. We have designed a novel, iterative LP solver that has a minor computational overhead and computes a job distribution that achieves near-optimal expected running time. We then have further improved the efficiency of our scheduling method with hierarchical scheduling to handle a larger number of resources. To demonstrate the benefits of our method, we have applied our hybrid parallel framework and scheduling algorithm into five different applications. With two hexa-core CPUs and four different GPUs, we were able to achieve an order of magnitude performance improvement over using a hexa-core CPU. Furthermore, we have shown that our method robustly improves the performance in all the tested benchmarks, as we add more computing resources. In addition, we improved a basic work stealing method with our expected running time model and it shows 18% higher performance on average in the tested benchmarks.

Our method currently assumes that all the data (e.g., geometry and BVH) is in each computing resource. For large data sets that cannot fit into a device memory, we need to consider a better data management across different computing resources. As a first step, in the next chapter, we propose an out-of-core proximity computation technique for a proximity query, ϵ-nearest neighbor search, in particle-based fluid simulation.

Chapter 5. Out-of-Core Proximity Computation for Particle-based Fluid Simulations

Thanks to ever growing demands for higher realism and the advances of particle-based fluid simulation techniques, large scale simulations are getting increasingly popular across different graphics applications including movie special effects and computer games. This trend poses numerous technical challenges related to an excessive amount of computations and memory requirements.

In this chapter, we are mainly interested in handling nearest neighbor search used for particle-based fluid simulations. Nearest neighbor search is performed for each particle in the simulation and is a performance bottleneck of the simulation in practice. In our SPH simulation based on the method of Becker and Teschner [74], we found that neighbor search can take more than 50% of the overall simulation time, when we use a single CPU core. Most particle-based fluid simulations use ϵ-Nearest Neighbor, ϵ-NN, for a query particle, which identifies all the particles that are located within a search sphere, whose center is at the query particle and radius is set to ϵ. To handle such a high computational cost of ϵ-NN, many parallel techniques based on multi-core CPUs [8] or GPUs [38, 37] have been proposed. Thanks to these recent works, we can achieve high performance improvement (e.g., $20\times$ to $40\times$) by using a GPU for processing ϵ-NN queries over a single CPU-core based serial method.

Unfortunately, it has not been actively studied to handle massive-scale ϵ-NNs for data sets that do not fit in the GPU memory (video memory) for particle-based fluid simulations. For example, Harada et al. [38] reported that GPU can handle about 5 M particles per 1 GB video memory and most commodity-level GPUs have from one to three GBs of the video memory. As a result, large-scale particle-based fluid simulations consisting of more than 10 M or more particles have to be processed in a much less performance in the CPU side that can have much larger memory space than GPU. This is mainly because prior GPU-based parallel techniques were neither designed for the out-of-core case nor directly applicable to such cases.

In this chapter, we propose an out-of-core technique utilizing heterogeneous computing resources for processing ϵ-NNs used in a large-scale particle-based fluid simulation consisting of tens of millions of particles. In particular, we handle the out-of-core problem where the video memory of GPUs cannot hold all the necessary data of ϵ-NN, while main memory of CPU is large enough to hold such data. We use a uniform grid, a commonly employed acceleration data structure for particle-based simulations. Given this context, we use the granularity of a block containing a sub-grid of the uniform grid as a main work unit, to streamline various computation and memory transfers between CPU and GPU. Once GPU receives a block from CPU, the GPU performs ϵ-NNs with the particles contained in the block (Sec. 5.1.1). Our main problem is then reduced to identifying a maximal work unit that can fit into the video memory. To estimate the memory requirement of processing a block, we present a novel, memory estimation method based on the expected number of neighbors for a query particle (Sec. 5.2). To efficiently compute a maximal block for each GPU, we also propose a simple, hierarchical work distribution method (Sec. 5.1.2).

To demonstrate the benefits of our method, we have tested our method with three large-scale particle-based fluid simulation benchmarks consisting of up to 65 M particles (Fig. 1.1 and Fig. 5.4).

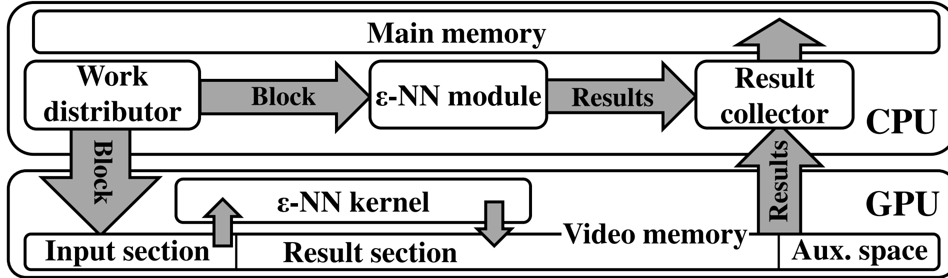


Figure 5.1: *This figure shows an overall framework for processing ϵ-NNs in an out-of-core manner using heterogeneous computing resources.*

These benchmarks require up to 16 GB memory space for processing ϵ-NNs. Our out-of-core method for ϵ-NNs can process these benchmarks with a GPU that has only 3 GB video memory. Overall, our method can perform up to 15 M ϵ-NNs per second. We have also implemented an alternative, GPU-based out-of-core approach based on an Nvidia's mapped memory method [62]. Compared to this alternative, our method shows up to 26 $\times$ performance improvement. These results are mainly thanks to the efficiency of our out-of-core method and the high accuracy of our memory estimation model that shows up to 0.97 linear correlation with respect to the observed number of neighbors. Also, compared to our base method, an in-core CPU version using only those two hexa-core CPUs and the large main memory space holding all the data, our method achieves up to 6.3 $\times$ improvement using an additional GPU. This result is 51$\times$ higher performance compared to using a single CPU core.

5.1 Out-of-Core, Parallel ϵ-NN

In this chapter, we target mainly for handling large-scale ϵ-NN used for particle-based fluid simulation both in out-of-core and parallel manners. Theoretically, achieving the optimal performance in this context is non-trivial and thus has been studied only for particular problems such as sorting and FFTs [75] on the shared memory model with the same parallel cores. Instead, we propose a simple, hierarchical approach, tailored to our particular problem, that simultaneously computes a job unit that can fit into the video memory of a GPU, while utilizing heterogeneous parallel computing resources.

5.1.1 System Overview

The main goal of our system is to efficiently find and store the neighborhood information for a massive amount of particles that cannot be handled at once by a GPU. We assume that the CPU memory is large enough to hold all those information. This assumption is valid for tens or hundreds of millions of particles, since current PCs can have hundreds of gigabytes up to 4 TB memory.

Fig. 5.1 shows an overview of our system. We use a uniform grid that are commonly used for accelerating ϵ-NN, while determining cell indexes with Z-curve to exploit spatial locality [8]. The simulation space is split uniformly into *cells* so that the length of each cell is equal to ϵ or 2ϵ. Then, neighboring particles for a query particle with the ϵ-NN are located in the cell enclosing the query particle or its adjacent cells. Initially, the uniform grid is stored in main memory. As a result, we need to send those cells and their particles from CPU to GPU to perform the ϵ-NN in the GPU side. We use the term *processing cells* to denote the process of performing ϵ-NN for particles in the cells.

In the CPU side, *work distributor* divides the uniform grid into sub-grids dynamically and assigns them to available computing resources based on our hierarchical work distribution method (Sec. 5.1.2). To process data in a cache-coherent mammer, we divide the uniform grid in the form of a cubic sub-grid. A *block* represents a sub-grid and contains an *active cell list* that includes the indexes of cells contained in the sub-grid. The term *processing block* denotes processing cells in the active cell list of the block.

The *ϵ-NN module* in CPU and *ϵ-NN kernel* in GPU receive a block from the work distributor when they are idle. Once ϵ-NN module or ϵ-NN kernel finishes to process the block, it pushes the results back to the result collector. Finally, the *result collector* takes the results, stores them in main memory, and returns them to the particle-based fluid simulator.

5.1.2 Work Distribution

To process ϵ-NN for a massive number of particles while fully utilizing high performance GPUs in an out-of-core manner, we divide the grid such that the size of the working set of each block should be smaller than the size of video memory.

Processing a block requires to access particles and to write information of their identified neighbor particles. Therefore, the required memory size, $s(B)$, for processing a block, B, can be determined mainly by the number of particles, n_B, stored in the block and the number of neighbors for each particle, n_{p_i}, as the following:

$$s(B) = n_B s_p + s_n \sum_{p_i \in B} n_{p_i}, \tag{5.1}$$

where s_p and s_n are the data sizes of storing a particle and a neighbor particle, respectively. i indicates the i-th particle stored in the block B. Typically, s_n is 8 bytes required for encoding an index of a particle and the distance between the query and its neighbor particle that is used for computing forces in the simulation part. s_p requires 12 bytes to encode the particles position. In addition we need a minor space to store sub-grid data, active cell list, etc.

Evaluating the required size of processing a block is straightforward except the number of neighbors, n_{p_i}. Unfortunately, we cannot know the exact number of neighbors until we actually perform the query, a common chicken-and-egg problem. A naive approach is a two-pass algorithm: it measures the required memory space without writing result at the first pass and re-processes the queries while storing the result in the second pass. Although we can exactly measure n_{p_i}, but this approach has redundant operations. Another alternative is to use a general vector-like data structure that adaptively grows according to identified neighbors [76]. This data structure, unfortunately, is not designed for the out-of-core case and thus fails, when all the sizes of these vectors grow even bigger than the available video memory size.

Instead of these approaches, we estimate the number of neighbors as the expected number of neighbors, and reserve the memory space based on the estimation result (Sec. 5.2). When n'_{p_i} is the expected number of neighbors for the i-th particle, $s(B)$ becomes as following:

$$s(B) = n_B s_p + s_n \sum_{p_i \in B} n'_{p_i} + s_{Aux}. \tag{5.2}$$

In the above equation, we introduce a new term, s_{Aux}, the size of an auxiliary space, which is an additional space to handle the error related to our estimation process, and its size is computed depending on n_B and the estimation error; its details are given in Sec. 5.2. For example, a block of 32^3 has up to about 1.3 M particles, and 18 expected neighbors per particle on average across our tested benchmarks.

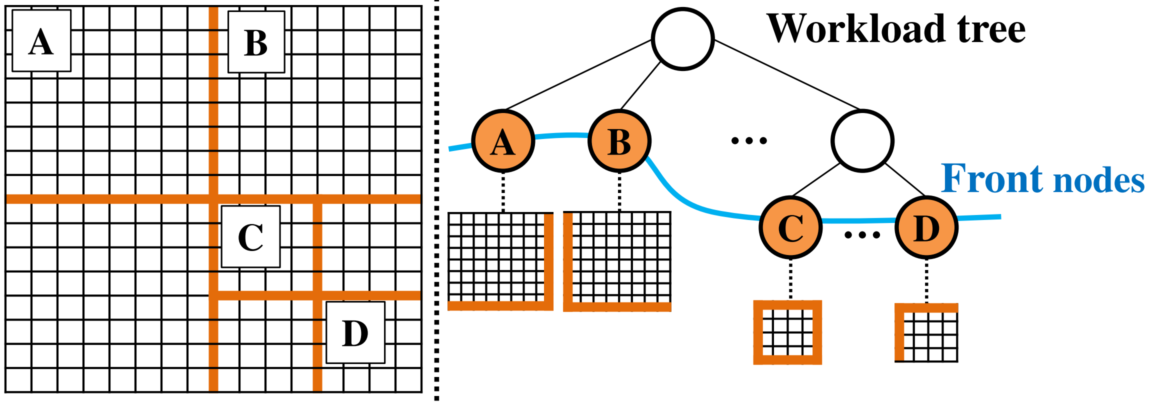


Figure 5.2: *The left figure shows a uniform grid with a few sub-grids; boundaries of these sub-grids, i.e., blocks, are shown in orange lines. The right figure shows an example of the workload tree with these blocks.*

$n_B s_p$, $s_n \sum_{p_i \in B} n'_{p_i}$, and s_{Aux} then take 15.6 MB, 187 MB, and 38.5 MB, respectively. Thanks to this memory estimation process, we can efficiently find a block that fits in the video memory and perform ϵ-NNs for the block in GPU without any memory I/O thrashing.

Once we know the required memory space for processing a block, the next task is to divide the uniform grid into blocks. Simply, we can use a small fixed size (e.g., 8^3) of blocks so that each block requires less memory space than the size of the video memory. We found that a larger block size (e.g., 64^3) shows better performance, but it is hard to find a maximal block size for each simulation in practice, since the maximal block size that GPU can handle varies depending on the simulation state and regions of the simulation space (Sec. 5.3.3).

Hierarchical work distribution. To efficiently construct such a maximal block, we use a *workload tree*, which is an octree built on top of the grid (Fig. 5.2). Each node of the workload tree represents a block, and also contains the number of particles included in the block and the expected number of neighbors of those particles. Its child nodes are computed by dividing the sub-grid of the parent node in all the dimensions. As a result, each leaf node of the tree represents a cell, while the root includes the whole uniform grid.

The work distributor running on a CPU thread finds the maximal blocks that the GPU video memory can hold by traversing the workload tree, and give those blocks to GPU threads. The detail work flow of our hierarchical work distribution is summarized in Algorithm 1. The work distributor maintains a *front node queue* containing blocks that are candidates for maximal blocks.

Initially, the front node queue contains the root node of the workload tree. When a GPU has available video memory space, the work distributor takes the front node and checks whether the available space is larger than the required memory size for processing the block in the node. If we have enough available space in the GPU, the work distributor assigns the block to the GPU. Otherwise, we also check whether the block size is bigger than the maximum size of the GPU video memory. If the required memory size is larger than the maximum size of video memory, we have to process its child blocks and thus the distributor enqueues its eight child nodes to the queue. If the required memory size of the block is smaller than the maximal video memory size, we decide not to process it at the current time, and thus push the node back to the queue, which will be processed later when the GPU has enough space for

Algorithm 1 Our hierarchical work distribution algorithm

$Queue_{node} \leftarrow$ push the root node of the workload tree
Repeat until $Queue_{node}$ is not empty
if $(G_{remain} \leftarrow the\ remaining\ size\ of\ video\ memory) \geq 0$ **then**
  $B \leftarrow$ pop a node from $Queue_{node}$
  if $(s(B) \leftarrow required\ memory\ size\ to\ process\ B) \leq G_{remain}$ **then**
    Give B to the GPU
  else
    if $s(B) > (G_{max} \leftarrow the\ maximal\ video\ memory\ size)$ **then**
      $Queue_{node} \leftarrow$ push the eight children of B
    else
      $Queue_{node} \leftarrow$ push B
    end if
  end if
end if

the block. Based on this simple, hierarchical process, we efficiently identify maximal blocks that can be processed simultaneously in the GPU, while utilizing the available video memory.

Boundary handling. There are two types of cells in a sub-grid; the cells at the boundary of a block (*boundary cells*) and other cells are *inner cells* (Fig. 5.2). Usually, the number of boundary cells is much smaller than that of inner cells, because boundary and inner cells exist in 2D and 3D space, respectively. To find neighbors of boundary cells of a block we need to access cells in its adjacency block and it requires a larger working set over handling inner cells, resulting in a lower locality. As a result, we let the CPU cores to process those boundary cells, since main memory already has all data and CPU can efficiently handle random memory access thanks to the well-organized caches. On the other hand, we let a high-performance GPU to focus on processing a large number of inner cells.

5.1.3 Processing a Block in GPU

When a block is given to a GPU, we create a new stream to perform the data transfer for concurrently processing other blocks and hiding the data transfer overhead. We then configure work space in the video memory. The work space is decomposed into input, result, and auxiliary space sections (Fig. 5.1). Each section reserves space based on each term of Eq. 5.2: $n_B s_p$ for the input section, $s_n \sum_{p_i \in B} n'_{p_i}$ for the result section, and s_{Aux} for the auxiliary space, respectively. Each query particle then receives the estimated, fixed amount memory space, and each GPU thread processes an ϵ-NN for the query particle in the block. When a GPU thread identifies neighbors for the query particle, it stores them in its pre-defined, corresponding space in the result section without any synchronization.

We may need further memory spaces than the pre-defined result section, due to the inaccuracy of our estimation process. In this overflow case, we write such results into the auxiliary space. Multiple GPU threads can access the auxiliary space simultaneously and thus we need to perform synchronization. Fortunately, we have found that this happens rarely (i.e., 3% of all identified neighbors on average), thanks to the high accuracy of our estimation model. Even when the auxiliary space becomes full, we can also use an additional space in main memory in the CPU side. This operation accessing main memory

from the GPU side is very expensive, and never happened in our method with the tested benchmarks. Our approach of handling these overflows can be seen as designing an effective out-of-core vector data structure, whose initial size is determined by our memory estimation model, while reducing the expensive synchronizations.

5.2 Expected Number of Neighbors

We have described so far that it is critical to compute and reserve an appropriate amount of memory space for processing blocks in an out-of-core manner. The main unknown factor for computing the required memory space (Eq. 5.2) for processing blocks is to compute the expected number of neighbors, n'_{p_i}, of a particle. We estimate it based on the particle distribution in the simulation space, while considering the relationship between the search radius and cell size.

5.2.1 Problem Formulation

ϵ-NN for a particle, p, is to find neighbor particles, which are located within a search sphere, $S(p, \epsilon)$, whose center is at p and radius is ϵ. In general, particle distributions over the uniform grid covering the simulation space is not uniform. In many cells, however, particle distributions tend to show local uniformity around each cell in particle-based fluid simulations. This is mainly because designing high-quality SPH techniques is related to reduce the density variation over time [77, 74] and therefore particles tend to have a similar movement with nearby particles, while maintaining a specific distance with them. Based on this observation, we assume a local uniform distribution, i.e., particles are uniformly distributed in each cell.

Assuming the local uniform distribution, the number of neighbors is then proportional to the overlap volume between the search sphere $S(p, \epsilon)$ and cells weighted by their associated particles. Specifically, the expected number of neighbors, $E(p_{x,y,z})$, for a particle p located at (x, y, z) is defined as the following:

$$E(p_{x,y,z}) = \sum_i n(C_i) \frac{Overlap(S(p_{x,y,z}, \epsilon), C_i)}{V(C_i)}, \tag{5.3}$$

where C_i indicates the cell containing $p_{x,y,z}$ and its adjacent cells that have any overlap between the search sphere and the bounding box of the cell C_i. $n(C_i)$ is the number of particles contained in the cell C_i, and $V(C_i)$ represents the volume of the cell. $Overlap(S(p_{x,y,z}, \epsilon), C_i)$ represents the overlap volume between $S(p_{x,y,z})$ and C_i.

This equation requires us to compute $Overlap(S(p_{x,y,z}, \epsilon), C_i)$ for each query particle in a cell, and thus causes a high computational overhead overall, since many particles can exist in each cell (e.g., 10 to 30 on average). Instead, we compute the average, expected number of neighbors for particles of a cell, C_q, and use the value, $E(C_q)$, for all the particles, as their expected number of neighbors. The average, expected number of neighbors of particles $E(C_q)$ in a cell C_q is then defined as:

$$\begin{aligned} E(C_q) &= \frac{1}{V(C_q)} \int_0^l \int_0^l \int_0^l E(p_{u,v,w}) \, \mathrm{d}u\mathrm{d}v\mathrm{d}w \\ &= \frac{1}{V(C_q)} \sum_i n(C_i) \frac{D(C_q, C_i)}{V(C_i)}, \end{aligned} \tag{5.4}$$

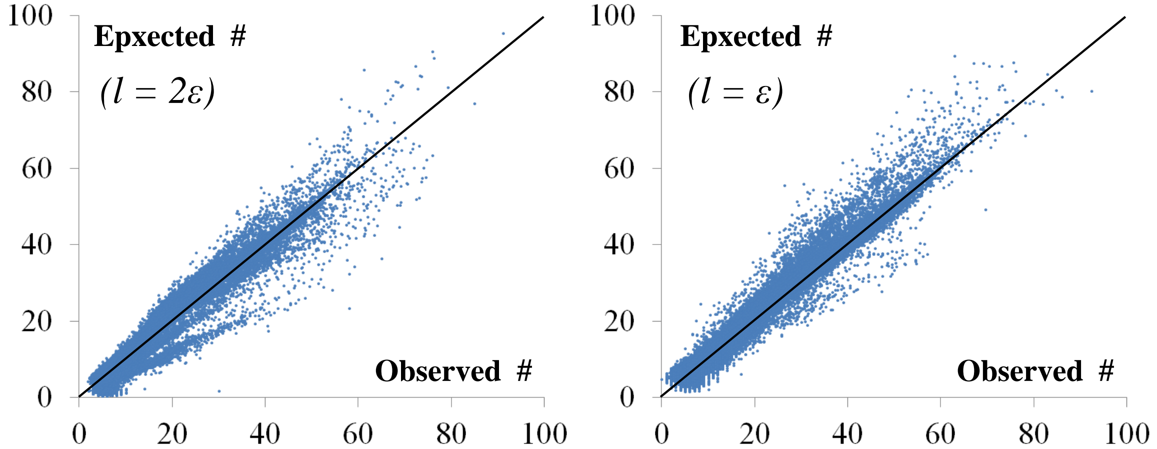


Figure 5.3: *This plot shows the expected and observed number of neighbors in two configurations, $l = 2\epsilon$ (the left) and $l = \epsilon$ (the right), for the dam breaking benchmark. They show high correlations, 0.97 and 0.96, respectively.*

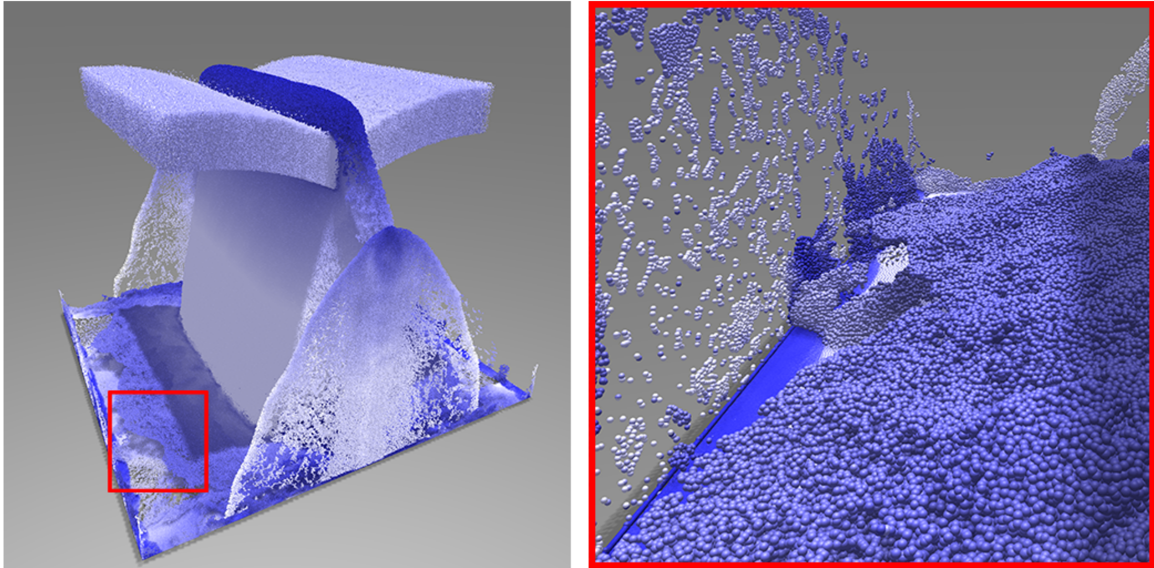
where $D(C_q, C_i) = \int_0^l \int_0^l \int_0^l Overlap(S(p_{u,v,w}, \epsilon), C_i) \mathrm{d}u\mathrm{d}v\mathrm{d}w$, l is the length of a cell along each dimension, and $p_{u,v,w}$ is a particle p positioned at (u, v, w) on a local coordinate space in C_q. Given the uniform grid with l and ϵ values, which are not frequently changed by users, $D(C_q, C_i)$ can be pre-computed. As a result, we pre-compute these values in an offline manner (taking a few seconds). Specially we use the Monte Carlo method, which achieves high accuracy as we generate many samples (e.g., 1 M).

At runtime, we evaluate $E(C_q)$ of Eq. 5.4 by considering $n(C_i)$ and looking up pre-computed $D(C_q, C_i)$ values, which are stored at a less than 1 KB sized look-up table. Overall, this runtime evaluation is done in a constant time. All the expectation computation combined with traversing the workload tree takes 100 ms to 500 ms at each frame on average with our tested benchmarks consisting of up to 65 M particles.

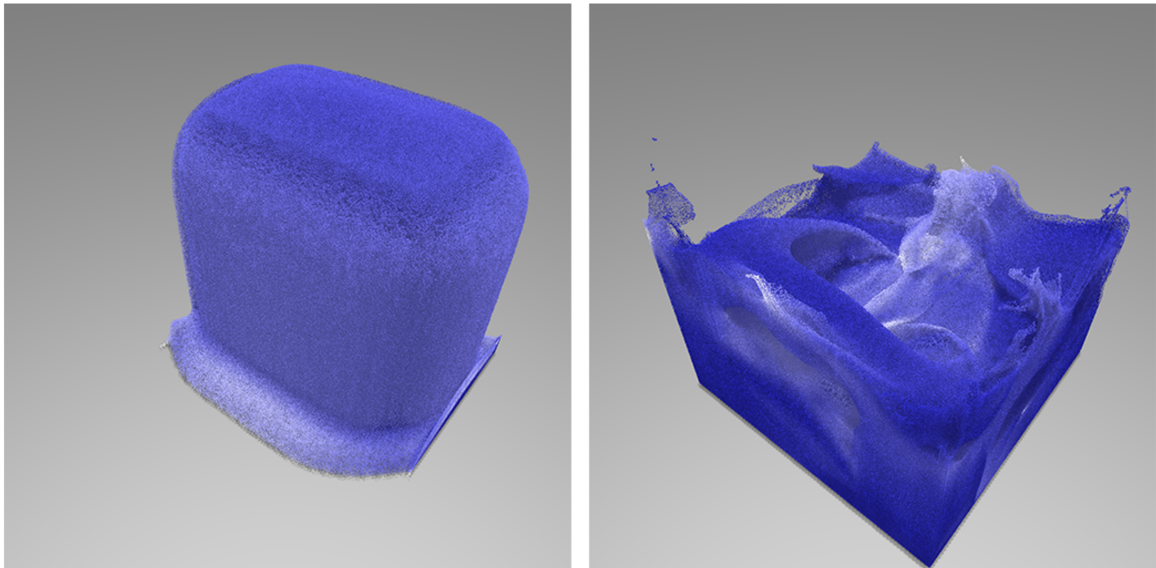
5.2.2 Validation and Error Handling

We have measured the accuracy of our expectation model with our tested benchmarks. We use two most common configuration, $l = 2\epsilon$ and $l = \epsilon$, for computing the uniform grid, and compare the expected number of neighbors $E(C_q)$ with the actual number of neighbors that are computed after finishing ϵ-NN for each particle. Fig. 5.3 shows the scatter plots between the expected and observed results. They show high correlations (e.g., 0.97) and we achieve such a high accuracy by considering the number of particles in cells, while assuming the local uniform distribution of particles. We have observed similar results with other tested models.

Additionally, we have measured the root mean square error (RMSE) between the estimated and observed ones. The RMSE is computed as 3.7 for the four source benchmark, and indicates that our estimated number of neighbors can be higher or lower by 3.7 on average to the actual number of neighbors. This information is useful for estimating the required space for the auxiliary space. At the worst case where we underestimate 3.7 neighbors for each query particle in a block, we need to access the auxiliary space to accommodate such underestimation. RMSEs of other tested models are lower than 3.7. Based on this analysis, we set the size of the auxiliary space s_{Aux} in Eq. 5.2 as $3.7 * (n_B s_n)$ to cover the worst case.



(a) Two sources (65.6 M particles)



(b) Dam breaking (15.8 M particles)

Figure 5.4: *These figures show two different large-scale fluid benchmarks.*

5.3 Results and Analysis

We have implemented and tested our out-of-core parallel ϵ-NN method in a machine consisting of a GPU (Nvidia Geforce GTX 780, 3 GB) and two Intel Xeon hexa-core CPUs (2.93GHz) with 192 GB main memory. We use 2.8 GB of 3 GB video memory for all tests, since some of GPU memory (e.g., 200 MB) is reserved by GPU drivers for display and running CUDA kernels (e.g., thread local memory). We have implemented ϵ-NN module on multi-core CPU based on a prior method [8] and use 12 threads for the module. For ϵ-NN kernel in GPU, we have implemented a locality-aware GPU algorithm based on a prior in-core GPU algorithm [37]. We have implemented a vector data structure for the auxiliary space in GPU by using the atomic operation [76].

Our simulation method [74] has been implemented on multi-core CPUs based on Ihmsen et al. [8]. We first perform ϵ-NNs and pass their results to the simulation solver, which moves particles based on the computed neighbor search results. We set the cell length of the uniform grid as the two time of search radius, i.e., $l = 2\epsilon$, since it is one of commonly adopted choices in practice [78]. The grid resolutions of our benchmarks are then 128^3 for dam breaking and four source benchmarks and 256^3 for the two source benchmark.

To compare the efficiency and robustness of our out-of-core system, we have implemented two alternative methods:

- *IC-GPU* has been implemented by removing all out-of-core features from our method. This method stores all results in a vector structure designed for GPU [76] like the auxiliary space in our system. All available GPU memory is used as a memory pool for the vector structure.
- *Map-GPU* method uses Nvidia's mapped memory techniques. A sufficiently large space (e.g., 50 GB) is reserved in main memory and is then mapped into the GPU memory address space for writing ϵ-NN search results.

We have also implemented a simple workload distribution method to measure the benefit of our hierarchical approach:

- *Fixed-Block* method divides the uniform grid into the fixed size (e.g., 16^3) of blocks and assigns blocks to GPU, while using multiple streams to hide the data transfer overhead. This method is also tested by reserving a maximal memory space (*Fixed-Block(Max)*), or by reserving the memory space based on our memory estimation method (*Fixed-Block(Exp)*).

Benchmarks: We have tested different methods against three different benchmarks (Table 5.1). The first benchmark, Dam, is a well-known dam breaking benchmark that has a fixed number of particles, 15.8 M particles, throughout the simulation (Fig. 5.4(b)). The other two benchmarks, four and two sources benchmarks, have four or two sources emitting particles up to 32.7 M and 65.6 M particles, respectively (Fig. 1.1(d) and Fig. 5.4(a)). These benchmarks are available at our project webpage, which is unprovided now for the anonymous submission.

The average number of neighbors for each particle in three different benchmarks ranges 11 to 26, while their maximum reaches up to 489 neighbor particles. When we attempt to process the whole uniform grid in a single block, these three benchmarks require 6 GB to 16 GB memory space to contain all the required data given the configuration. The space is used for holding particle positions, grid structures, and recording neighbors identified for ϵ-NNs.

	Dam (Fig. 5.4(b))	Four src. (Fig. 1.1(d))	Two src. (Fig. 5.4(a))
Max. # of pts.	15.8 M	32.7 M	65.6 M
Max. data size	5.7 GB	15.5 GB	13.0 GB
Avg. n_{p_i} / Max. n_{p_i}	15.4 / 184	26.1 / 489	11.0 / 327
Avg. $\sigma(n_{p_i})$	9.6	26.9	10.8

Table 5.1: *This table shows different statistics of each benchmark. We show the average and maximum numbers of neighbors computed for each simulation frame, with the standard deviation. The max. data size is the maximum $s(B)$ among all frames, where B is the whole grid.*

5.3.1 Results

Fig. 5.5 shows the performance of different methods on each benchmark. For the four and two sources benchmarks, we draw the graph as a function of the number of particles, to see the overhead and benefits of our out-of-core approach over the alternative methods.

As long as all the data fit into the video memory of a GPU, we can use the in-core method to perform all ϵ-NN queries within the GPU. Based on our memory estimation model, our approach determines maximal blocks that fit into the GPU video memory and thus uses the in-core ϵ-NN processing algorithm with those blocks. When we have the small number of particles (e.g., 5 M), our method shows 60% lower performance on average than *IC-GPU*, since our method generates the workload tree to estimate the memory footprint; it takes from 100 ms to 500 ms in CPU and it is relatively a high overhead in the case with the small number of particles. This overhead, however, is a small price to pay for handling the out-of-core case.

At a specific point (e.g., 12 M particles for the four sources benchmark), the required memory size exceeds the size of video memory. Our method continues to process larger particles in an out-of-core manner, while *IC-GPU* fails to perform ϵ-NNs.

We have compared our method with *Map-GPU* to see the efficiency of our out-of-core approach. Our method achieves higher performances: 26× for the Dam, 18× for the four sources, and 5× for the two sources benchmark over *Map-GPU* on average. Detailed implementation for the mapped memory feature used for *Map-GPU* in the GPU driver is not available, but L2 cache in the GPU side is used for the mapped memory. On the other hand, we specifically use the global memory in GPU for caching data (i.e., particles and cells) and reserving the memory space with our memory estimation model. Thanks to them, our method achieves such high performance improvements over *Map-GPU*.

Fig. 5.5 also compares the performance of ours and *Fixed-Block(Max)*, which reserves a maximal memory space, i.e., 250 neighbors, for each particle, and uses 16^3 blocks for all the benchmarks. When the number of neighbors exceed the maximum, it uses the auxiliary space in the video memory. Our method achieves 6× for the Dam., 4× for the four sources, and 4× for two sources benchmark over *Fixed-Block(Max)* on average. This result demonstrates the benefits of our hierarchical work distribution method based on our memory estimation model.

Comparison with CPU-based ϵ-NN. One could simply use the CPU with a large main memory space to handle our tested benchmarks. We have implemented the state-of-the-art parallel ϵ-NN for particle-based simulation [8] and compared the performance with ours. Compared with using a single CPU-core, the parallel CPU algorithm using 12 cores shows 8.4× higher performance on average. With

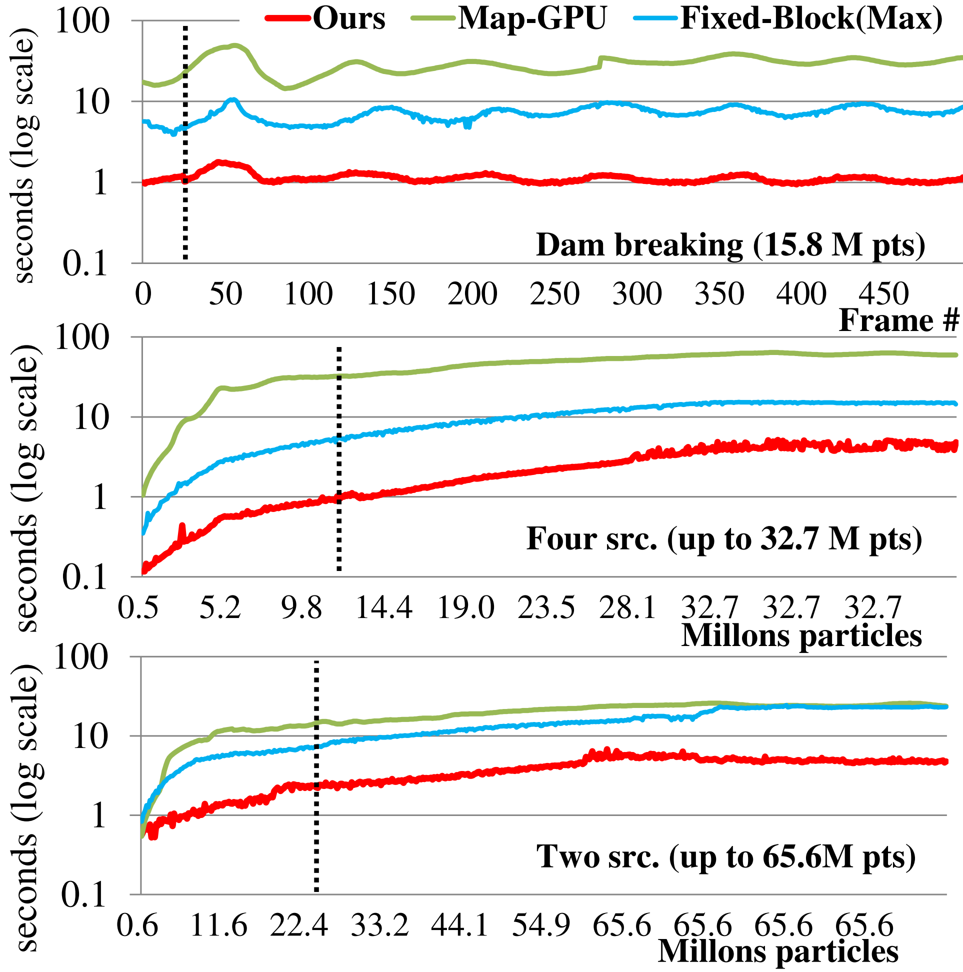


Figure 5.5: *These graphs show the processing time in the log scale for ϵ-NNs based on different out-of-core methods including ours. The measured time includes data communication time for sending input data and copying the results to the main memory. Starting from the dotted lines, our method runs in an out-of-core manner.*

an additional GPU, our method achieved 5.4× on average and up to 6.3× performance improvement over the parallel CPU algorithm. This is 46× on average and up to 51× higher performance compared with using a single CPU core. Please note that the performance improvement we report in the thesis is computed by including the data transfer overhead between CPU and GPU.

5.3.2 Benefits of Our Memory Estimation Model

To measure benefits from our estimation model, we have tested our method without using the estimation model. Instead we set a fixed space for recording results of ϵ-NNs; when we have the overflow, we also use the auxiliary space in the video memory and then use space in main memory, as used for our method. Fig. 5.6 shows the average processing time with various fixed space sizes within our method and our method with the estimation model in the benchmarks. Overall, our system equipped with our estimation method achieves much higher performance, 1.4 × to 14.2 × over the tested fixed neighbors.

For a small fixed space (e.g., 16 or 32), we found that some of the identified neighbors have to be recorded in main memory through expensive PCI-Express communication, and thus it can drop down

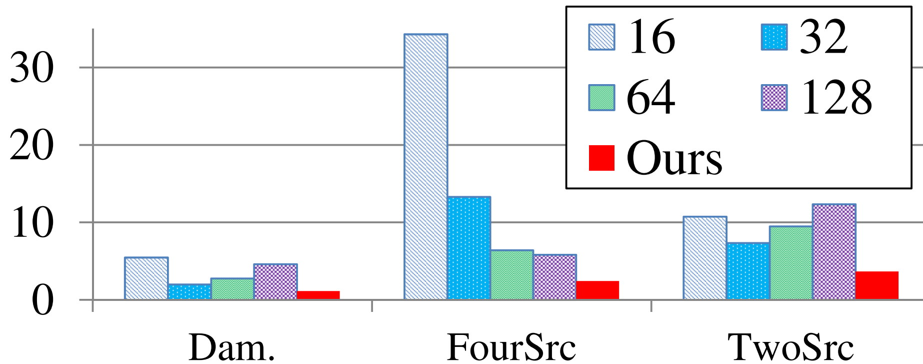


Figure 5.6: *This graph compares the processing time (seconds) for ϵ-NNs with various space sizes and our estimation model for recording results within our method.*

the performance significantly, especially for the four sources benchmark. For a large space (e.g., 64 or 128) the transaction to main memory happens rarely, but we observe low GPU utilizations, especially in the two sources benchmark, since we cannot send many particles to GPU. On the other hand, our memory estimation method results in a high space utilization, i.e., more than 90% of allocated spaces are used, while achieving high GPU utilizations. These results demonstrate the effectiveness and robustness of our approach.

5.3.3 Benefits of Hierarchical Workload Distribution

To measure only the benefit brought by our hierarchical work distribution approach, we have tested the performance of *Fixed-Block(Exp)* that reserves spaces according to our memory estimation method, not to the maximum number of neighbors (e.g., 255). We have tested 16^3, 32^3, and 64^3 as the fixed block size and found that a larger block size shows a better performance, as long as these blocks can fit into the video memory. For example, 32^3 and 64^3 block sizes show 1.5 and 1.8 times higher performance compared with using 16^3 blocks in the four source benchmarks, respectively. When we measure the GPU processing time only, 32^3 and 64^3 block sizes take 22% and 30% less times than using 16^3 blocks on average. This is mainly because a large block can better utilize the massive parallel nature of GPU architecture.

The maximal block size, unfortunately, varies depending on the benchmarks. For achieving the best performance in each benchmark, we have to manually set 32^3 for Dam. and two sources, and 64^3 for two source benchmarks. On the other hand, our hierarchical work distribution with our memory estimation method finds the optimal block size dynamically without those manual tunning. Furthermore, our hierarchical workload distribution method achieved 33% higher performance on average across all the tested benchmarks compared with *Fixed-Block(Exp)*, which reserves memory with our memory estimation method, but uses the manually chosen, best block size for each benchmark.

5.4 Conclusion and Discussion

In this chapter, we have presented an out-of-core technique for ϵ-NN computing used in large-scale particle-based fluid simulation consisting of tens of millions of particles. Our method processes ϵ-NNs based on blocks (i.e., sub-grids) of the uniform grid associated with particles that can fit into the video memory of GPU. Specifically, we have proposed a novel estimation model for the number of neighbors

for particles and used the model for estimating the memory footprint required for processing a block based on the workload tree. We have applied our method to three different large-scale particle-based fluid simulations whose memory requirement is much bigger than the video memory of GPUs. Overall our method has shown higher performances over other out-of-core techniques.

Implicit and incompressible SPHs. In this thesis, we have used an explicit method for the particle-based simulation, i.e., WCSPH [74]. In implicit methods like PCISPH and IISPH [77, 79], the relative computational overhead of the neighbor search step can be reduced, since implicit methods can tolerate larger time steps. Nonetheless, ϵ-NN is still a basic operation even in these implicit methods, and the neighbor search step is also recommended to be used in the convergence iterations for more accurate simulations. As a result, the proposed approach can improve the performance of these implicit methods. We leave to verify this as one of future directions.

Limitations and future work. Our memory estimation method has a high accuracy and did not cause overflows from the auxiliary space in our tested benchmarks. There is, however, no guarantee to prevent such overflows in general. Our workload tree is mainly designed for handling out-of-core particle data, but can be used well even for utilizing multiple GPUs. We have also tested our method by adding one more GPU to our currently tested machine configuration, but have observed about 30% improvement. Along this direction, we would like to extend it further for achieving the optimal performance and higher scalability even for parallelization efficiency based on optimization-based scheduling methods proposed in Chapter 4.

Our current technique adopted a modular approach by decoupling ϵ-NN and simulation parts. As a result, when we have a better module on these two parts, we can achieve higher performance easily, by simply replacing one of existing modules with a better one. However, our current approach assumed that the simulation accesses the simulation grid block by block, which is common practice for accessing them. As a future work, we would like to extend our modular approach to allow random access from the simulation part. Finally, in the same line of the modular approach, we have chosen to record the result, the identified neighbors, of each ϵ-NN query in the video memory and then send to CPU running the simulation. This approach can use a large memory footprint. As a future work, we would like to reduce its memory footprint, while maintaining the benefit of the modular approach.

Chapter 6. Conclusion

In this thesis we have accelerated various proximity computations by efficiently utilizing heterogeneous computing systems. We have proposed proximity computing frameworks and scheduling algorithms while considering the heterogeneity of computing resources in computational and spatial perspectives.

To achieve high utilization efficiency of heterogeneous computing systems, we have proposed the hybrid parallel algorithm that is specialized for a proximity query, i.e. continuous collision detection. Then we have handled various proximity queries in the same way by designing the general proximity computing framework. In addition we maximize the utilization efficiency of heterogeneous computing systems by designing LP-based scheduling algorithm that finds an optimal job distribution in the perspective of processing time. Finally we have extended our attention to out-of-core proximity computations to handle a large-scale or massive-scale data that is larger than a memory space of computing resources. Specially, we have proposed out-of-core proximity computation techniques for particle-based fluid simulations. With the proposed systems and algorithms, we have achieved an order of magnitude performance improvement for various proximity computations by using up to two hexa-core CPUs and four different GPUs compared with the case of using a single CPU core. Also our methods have shown much higher performance over prior scheduling and out-of-core methods. These results demonstrate the generality, robustness, and efficiency of our methods.

Following paragraphs summarize the overview of our approaches and benefits of them.

- **Hybrid Parallel Continuous Collision Detection:** We have presented a novel parallel continuous collision detection algorithm that utilizes both multi-core CPUs and GPUs. Our inter-CD based parallel algorithm removes the synchronization in the main loop of the collision detection process and leads almost linear performance scalability for additional CPU cores. In order to efficiently utilize both multi-core CPUs and GPUs, we allocate suitable jobs to computing resources based on the knowledge of proximity computation and computing resources. In this method, multi-core CPUs process hierarchical jobs that have many conditional branches and requires random memory access since CPUs have well-organized cache hierarchies and branch prediction techniques. We have demonstrated the benefits of proposed methods on three different benchmarks. We have achieved more than an order of magnitude performance improvement by using four CPU cores and two GPUs over using only a single CPU core.

- **General proximity computing framework and optimization-based scheduling:** We generalized various proximity quires as a uniform form, a set of two common jobs including hierarchical traversal and leaf-level computation. Based on the uniform representation of proximity queries, we have designed a general framework that can perform various proximity queries while utilize heterogeneous computing environments. In order to achieve the optimal performance with a given heterogeneous computing resources, we have proposed an optimization-based scheduling algorithm. We have introduced the expected running time model that abstracts a complex performance relationship between jobs and computing sources. Based on the expected running time model, we formulate the scheduling problem as a linear programming (LP) problem that finds the optimal job distribution that minimizes the running time of the heterogeneous system. In addition we have

presented the iterative LP solver to solve the computational overhead of the LP-based scheduling algorithm. We have applied our method to three different types of proximity queries used in various applications including physically-based simulation, global illumination based rendering, and motion planning in robotics. Our method continually improves the performance as we add more computing resources different with prior approaches. As a result, our method have achieved up to an order of magnitude performance improvement by using two hexa-core CPUs and four different GPUs over using only a hexa-core CPU in various applications.

- **Out-of-core proximity computation for particle-based fluid simulations:** We have extended our attention to handling proximity queries for large-scale data that cannot be processed at once in a computing resource due to the memory limit. In particular we have proposed out-of-core ϵ-nearest neighbor (ϵ-NN) search algorithm for large-scale particle-based fluid simulation consisting of tens of millions particles. We have proposed a simple hierarchical work distribution method to efficiently compute a maximal work group (i.e. block). In addition, to efficiently use the limited memory space, we have proposed a novel estimation model for the number of neighbors and used the model for estimating the memory footprint required for processing a block based on the workload tree. We have applied our method to three different large-scale particle-based fluid simulations whose memory requirement is much bigger than the video memory of GPUs. Overall our method handles large-scale scenes robustly and has shown higher performances over other out-of-core techniques.

6.1 Discussions and Future Research Directions

In this section we discuss about limitations of our methods and propose future research directions.

Out-of-core computation for various proximity queries. We have proposed out-of-core computing method only for particle-based fluid simulation in this thesis. However, the out-of-core computing is also important issue in other proximity queries since data size is continue to grow in many applications [63]. Therefore we would like to design a general out-of-core proximity computation for various proximity queries. To do that we need to design a general form of memory footprint estimation model for various proximity queries and we may be able to design it based on the two common job types we have figured out. Also we can employ the compression approaches and adapt that to support various types of computing resources [80].

With the out-of-core methods, we would like to combine the general out-of-core technique with the proposed method in this thesis and present a proximity computing system that support various types of proximity queries while fully exploiting heterogeneous computing systems.

Large-scale heterogeneous computing systems. We have demonstrated the performance with machines consisting of up to six different parallel computing resources and discussed its optimality. However in supercomputing and cloud computing environments, it already has much large number of computing resources and it is evident that future architectures will have more computing resources.

It is one of the most challenging problems to maintain a near-optimal throughput, even though we have more than six computing resources. To address this challenge, it is critical to lower the underutilization of computing resources and is required to design a better communication method among the computing resources and the scheduler in terms of algorithmic and architectural aspects. We have

designed our LP-based iterative scheduler to achieve a high-quality scheduling result with a low computational overhead. Nonetheless, our iterative solver does not guarantee optimality of the solution and can lapse into a local minimum. Also, its overhead can be non-negligible in some cases. A further investigation is required to minimize the overhead and robustly handle local minimum issues.

Another challenging problem is to have a more accurate modeling for the expected running time of jobs. Although our linear formulation matches very well with the observed data, there are many other factors (e.g., geometric configurations) that give useful intuitions for workload prediction. We conjecture that by considering those factors, we can have a better model for expecting the workload of jobs [22].

In addition, we need to study more on hierarchical scheduling and would like to extend it to a multi-resolution scheduling method for large-scale heterogeneous computing systems. In this case, it is very important to group similar, not identical, parallel cores since those systems consist of thousands of computing resources that have different computational capacities.

Future heterogeneous computing architecture. One of the major performance bottlenecks of utilizing heterogeneous computing systems is data transfer overhead among computing resources. To solve the problem, recent heterogeneous computing systems are trying to combine computing resources more tightly and they will shared the physical same memory space in future architectures. Although we can remove the data transfer overhead among computing resources, we still consider the communication cost since each computing core needs its own work space to avoid synchronization overhead. Therefore we can apply our scheduling algorithm by adjusting it to consider the data communication and we believe that our methods still useful for future architectures.

In addition it may become more important to maintain locality in the perspective of each processing core in the future heterogeneous computing architectures since we will have more heterogeneous cores. We would like to design data re-ordering method for heterogeneous computing systems that re-arrange data so that data that process by the same core is nearly placed while avoiding simultaneous data access from a large number of cores. To design efficient data re-ordering algorithm, we may need to couple the scheduling algorithm with data re-ordering algorithms in both ways.

Accurate proximity computation. In this thesis we have focused on improving the performance of proximity computation and have achieved interactive performance for various proximity computations. Another important point of improving proximity computation is accuracy. We believe that our work in this thesis make it possible to employing more accurate proximity computation in applications. For example, the continuous collision detection query is accurate collision detection algorithm, but it is not actively used in interactive application such as modeling and games due to the high computational overhead. In this thesis we have shown that we can achieve real-time performance for the expensive query by efficiently utilizing heterogeneous computing systems and the accurate query can be applied to interactive applications. In other words, we provide more computational budget for proximity computation and it leads future researches that design more accurate proximity algorithm based the high computational capacity.

References

[1] M. Lin and D. Manocha, "Collision and proximity queries," *Handbook of Discrete and Computational Geometry*, 2003.

[2] M. Tang, M. Lee, and Y. J. Kim, "Interactive hausdorff distance computation for general polygonal models," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)*, vol. 28, no. 3, 2009.

[3] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)," in *IEEE Symp. on Interactive Ray Tracing*, 2006, pp. 61–69.

[4] Y.-J. Kim, Y.-T. Oh, S.-H. Yoon, M.-S. Kim, and G. Elber, "Coons BVH for freeform geometric models," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 169:1–169:8, Dec. 2011.

[5] S. Borkar, "Thousand core chips – a technology perspective," *Design Automation Conference*, pp. 746–749, 2007.

[6] NVIDIA, "CUDA programming guide 2.0," 2008.

[7] S. Yeo and H.-H. Lee, "Using mathematical modeling in provisioning a heterogeneous cloud computing environment," *Computer*, vol. 44, pp. 55–62, 2011.

[8] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner, "A parallel SPH implementation on multi-core CPUs," *Computer Graphics Forum*, vol. 30, no. 1, pp. 99–112, 2011.

[9] P. M. Hubbard, "Interactive collision detection," *Proc. of IEEE Symposium on Research Frontiers in Virtual Reality*, pp. 24–31, 1993.

[10] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–13, 1997.

[11] S. Gottschalk, M. Lin, and D. Manocha, "OBB-Tree: A hierarchical structure for rapid interference detection," *Proc. of ACM Siggraph*, pp. 171–180, 1996.

[12] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Trans. on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[13] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Computer Graphics Forum*, vol. 19, no. 1, pp. 61–81, 2005.

[14] T. Larsson and T. Akenine-Möller, "A dynamic bounding volume hierarchy for generalized collision detection," *Computers and Graphics*, vol. 30, no. 3, pp. 451–460, 2006.

[15] D. L. James and D. K. Pai, "BD-Tree: Output-sensitive collision detection for reduced deformable models," in *ACM SIGGRAPH*, 2004, pp. 393–398.

[16] G. Zachmann and R. Weller, "Kinetic bounding volume hierarchies for deforming objects," in *Virtual Reality Continuum and its Applications*, 2006, pp. 189–196.

[17] T. Larsson and T. Akmenine-Moller, "A dynamic bounding volume hierarchy for generalized collision detection," *Computer and Graphics*, vol. 30, no. 3, pp. 450–459, 2006.

[18] M. Otaduy, O. Chassot, D. Steinemann, and M. Gross, "Balanced hierarchies for collision detection between fracturing objects," *IEEE Virtual Reality Conf.*, pp. 83–90, 2007.

[19] S. Yoon, S. Curtis, and D. Manocha, "Ray tracing dynamic scenes using selective restructuring," *Eurographics Symp. on Rendering*, pp. 73–84, 2007.

[20] O. S. Lawlor and V. K. Laxmikant, "A voxel-based parallel collision detection algorithm," *Supercomputing*, pp. 285–293, 2002.

[21] M. Figueiredo and T. Fernando, "An efficient parallel collision detection algorithm for virtual prototype environments," *Parallel and Distributed Systems*, pp. 249–256, 2004.

[22] Y. Lee and Y. J. Kim, "Simple and parallel proximity algorithms for general polygonal models," *Computer Animation and Virtual Worlds*, vol. 21, no. 3-4, pp. 365–374, 2010.

[23] M. Tang, D. Manocha, and R. Tong, "Multi-core collision detection between deformable models," in *SIAM/ACM Joint Conf. on Geometric and Solid & Physical Modeling*, 2009, pp. 355–360.

[24] B. Heidelberger, M. Teschner, and M. Gross, "Detection of collisions and self-collisions using image-space techniques," *Proc. of Winter School of Computer Graphics*, vol. 12, no. 3, pp. 145–152, 2004.

[25] D. Knott and D. K. Pai, "CInDeR: Collision and interference detection in real-time using graphics hardware," *Proc. of Graphics Interface*, pp. 73–80, 2003.

[26] N. Govindaraju, S. Redon, M. Lin, and D. Manocha, "CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware," *EG. Workshop on Graphics Hardware*, pp. 25–32, 2003.

[27] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical gpu-based operations for collision and distance queries," *Computer Graphics Forum*, vol. 29, pp. 419–428, 2010.

[28] A. Kolb, L. Latta, and Rezk-Salama, "Hardware-based simulation and collision detection for large particle system," *EG. Symp. on Graphics hardware*, pp. 123–131, 2004.

[29] T. Vassilev, B. Spanlang, and Y. Chrysanthou, "Fast cloth animation on walking avatars," *Computer Graphics Forum (Eurographics)*, vol. 20, no. 3, pp. 260–267, 2001.

[30] G. Baciu and S. Wong, "Image-based techniques in a hybrid collision detector," *IEEE Trans. on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 254–271, 2002.

[31] N. Govindaraju, D. Knott, N. Jain, I. Kabal, R. Tamstorf, R. Gayle, M. Lin, and D. Manocha, "Collision detection between deformable models using chromatic decomposition," *ACM Trans. on Graphics*, vol. 24, no. 3, pp. 991–999, 2005.

[32] A. Gress, M. Guthe, and R. Klein, "GPU-based collision detection for deformable parameterized surfaces," *Computer Graphics Forum*, vol. 25, no. 3, pp. 497–506, 2006.

[33] A. Sud, N. Govindaraju, R. Gayle, I. Kabul, and D. Manocha, "Fast Proximity Computation among Deformable Models using Discrete Voronoi Diagrams," *ACM SIGGRAPH*, pp. 1144–1153, 2006.

[34] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proc. of the ACM SIGGRAPH/EG conf. on Graphics hardware*, 2003, pp. 41–50.

[35] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *SIGGRAPH Asia*. ACM, 2008, pp. 1–11.

[36] D. Qiu, S. May, and A. Nüchter, "Gpu-accelerated nearest neighbor search for 3d registration," in *Computer Vision Systems*. Springer, 2009, pp. 194–203.

[37] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, "Interactive SPH simulation and rendering on the GPU," in *Proc. of ACM SIGGRAPH/EG Symposium on Computer Animation*, 2010, pp. 55–64.

[38] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on GPUs," *In Proc. of Computer Graphics International*, pp. 63–70, 2007.

[39] J. M. Domínguez, A. J. Crespo, D. Valdez-Balderas, B. Rogers, and M. Gómez-Gesteira, "New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters," *Computer Physics Communications*, vol. 184, no. 8, pp. 1848–1860, 2013.

[40] D. Culler and J. P. Singh, *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999.

[41] M. Herlihy, "Obstruction-free synchronization: Double-ended queues as an example," in *Int. Conf. on Distributed Computing Systems*, 2003, pp. 522–529.

[42] M. L. Pinedo, *Scheduling: Theory, Algorithm, and Systems*. Springer, 2008.

[43] C. N. Potts, "Analysis of a linear programming heuristic for scheduling unrelated parallel machines," *Discrete Applied Mathematics*, vol. 10, no. 2, pp. 155 – 164, 1985.

[44] J. Lenstra, D. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, pp. 259–271, 1990.

[45] E. V. Shchepin and N. Vakhania, "An optimal rounding gives a better approximation for scheduling unrelated machines," *Operations Research Letters*, vol. 33, pp. 127–133, 2005.

[46] A. Nahapetian, S. Ghiasi, and M. Sarrafzadeh, "Scheduling on heterogeneous resources with heterogeneous reconfiguration costs," *5th IASTED Int. Conf. on Parallel and distributed computing and systems*, pp. 916–921, 2003.

[47] I. Al-Azzoni and D. G. Down, "Linear programming-based affinity scheduling of independent tasks on heterogeneous computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 1671–1682, 2008.

[48] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260 –274, mar 2002.

[49] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[50] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Foundations of Computer Science, Annual IEEE Symp. on*, vol. 0, pp. 356–368, 1994.

[51] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Notices.*, vol. 30, pp. 207–216, 1995.

[52] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon, "HPCCD: Hybrid parallel continuous collision detection," *Comput. Graph. Forum (Pacific Graphics)*, vol. 28, no. 7, 2009.

[53] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Euro-Par 2010 parallel processing*, 2010, pp. 235–246.

[54] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, 1996, pp. 318–328.

[55] V. Janjic and K. Hammond, "Granularity-aware work-stealing for computationally-uniform grids," in *Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM International Conference on*, 2010, pp. 123 –134.

[56] S. Hong and H. Kim, "An integrated GPU power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, pp. 280–289, 2010.

[57] Y. Zhang and J. Owens, "A quantitative performance analysis model for gpu architectures," in *High Performance Computer Architecture (HPCA), Symp. on*, 2011, pp. 382 –393.

[58] A. Binotto, C. Pereira, and D. Fellner, "Towards dynamic reconfigurable load-balancing for hybrid desktop platforms," in *IEEE Int. Symp. on Parallel Distributed Processing*, 2010, pp. 1–4.

[59] M. S. Smith, "Performance analysis of hybrid CPU/GPU environments," *Master Thesis, Portland State Univ.*, 2010.

[60] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens, "Out-of-core data management for path tracing on hybrid resources," *Comput. Graph. Forum (EG)*, vol. 28, no. 2, pp. 385–396, 2009.

[61] S.-E. Yoon, E. Gobbetti, D. Kasik, and D. Manocha, *Real-Time Massive Model Rendering.* Morgan & Claypool Publisher, 2008.

[62] NVIDIA, "CUDA programming guide 5.0," 2013.

[63] T.-J. Kim, X. Sun, and S.-E. Yoon, "T-ReX: Interactive global illumination of massive models on heterogeneous computing resources," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 3, pp. 481–494, 2014.

[64] X. Provot, "Collision and self-collision handling in cloth model dedicated to design garment," *Graphics Interface*, pp. 177–189, 1997.

[65] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *IEEE Computational Sci. and Engineering*, vol. 5, pp. 46–55, 1998.

[66] S. Curtis, R. Tamstorf, and D. Manocha, "Fast Collision Detection for Deformable Models using Representative-Triangles," *Symp. on Interactive 3D Graphics*, pp. 61–69, 2008.

[67] NVIDIA, "CUDA programming guide 4.0," 2012.

[68] K. Karmarkar, "A new polynomial-time algorithm for linear programming," *Combinatorica*, vol. 4, no. 4, pp. 373–395, 1984.

[69] C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, 1998.

[70] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge University Press, 2004.

[71] S. M. LaValle, *Planning Algorithms.* Cambridge University Press, 2006.

[72] P. Shirley and R. K. Morley, *Realistic Ray Tracing*, 2nd ed. AK Peters, 2003.

[73] K. Ravichandran, S. Lee, and S. Pande, "Work stealing for multi-core HPC clusters," in *Euro-Par 2011 Parallel Processing*, 2011, pp. 205–217.

[74] M. Becker and M. Teschner, "Weakly compressible SPH for free surface flows," in *Proc. of ACM SIGGRAPH/EG Symp. on Computer Animation*, 2007, pp. 209–217.

[75] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri, "Low depth cache-oblivious algorithms," in *ACM Symp. on Parallelism in Algorithms and Architectures*, 2010, pp. 189–199.

[76] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-time concurrent linked list construction on the gpu," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297–1304, 2010.

[77] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible SPH," *ACM Transactions on Graphics*, vol. 28, no. 3, p. 40, 2009.

[78] R. Hoetzlein, "FLUIDS v.2 - a fast, open source, fluid simulator," http://www.rchoetzlein.com/eng/graphics/fluids.htm, 2009.

[79] M. Ihmsen, J. Cornelis, B. Solenthaler, C. Horvath, and M. Teschner, "Implicit incompressible SPH," *Visualization and Computer Graphics, IEEE Transactions on*, 2013.

[80] T.-J. Kim, B. Moon, D. Kim, and S.-E. Yoon, "RACBVHs: Random-accessible compressed bounding volume hierarchies," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 2, pp. 273–286, 2010.

Summary

Parallel Proximity Computation on Heterogeneous Computing Systems for Graphics Applications

근접질의(proximity query)는 물체들 사이의 거리와 관련된 연산들로서, 컴퓨터 그래픽스, 물리기반 시뮬레이션, 로봇 공학 등 다양한 분야에서 사용되는 기반 연산 중 하나다. 근접질의는 또한 각종 응용 분야에서 가장 많은 연산 시간을 요구하는 연산이기도 하다. 이와 같은 근접질의 연산의 넓은 활용 분야와 그 연산 속도의 중요성에 따라 계층적 공간 분할 자료구조 등과 같은 다양한 가속 기술이 개발되어왔다. 이러한 가속 기술에 힘입어 연산의 속도가 수 십에서 수 백배 향상되었지만, 대용량 데이터에 대해 실시간의 성능을 제공하기에는 여전히 충분한 성능을 보여주지 못하고 있다. 최근 연산장치의 발전 경향을 살펴보면, 단일 연산 코어의 성능을 향상시키는 것 보다 하나의 칩(chip)에 여러 개의 연산 코어를 집적하는 방향으로 발전하고 있다. 이와 더불어 Intel의 Sandy Bridge나 Sony의 Cell 구조 같은 이종 병렬처리(heterogeneous parallel computing) 연산장치도 활발히 개발되고 있다. 하지만 기존 근접질의 가속 기술들은 병렬처리에 대한 고려가 미미한 편이다. 최근 다양한 병렬처리 근접질의 기술들이 개발되고 있지만, 다중 코어 CPU 또는 GPU 등 한 종류의 연산 장치 만 활용한다는 한계점을 가지고 있다. 또한, 여전히 대용량 데이터에 대해 실시간 성능을 보여주지 못하고 있다.

본 논문에서는 이종 병렬처리 연산장치를 활용하여 근접질의 연산을 가속한다. 그리고 효율적인 이종 연산 장치의 활용을 위한 시스템 및 알고리즘들을 제안한다. 본 논문은 우선 근접질의의 하나인 연속 충돌탐지 연산을 다중 코어 CPU와 GPU를 동시에 사용하여 가속하는 병렬처리 알고리즘을 제안한다. 다음으로 알고리즘을 다양한 근접질의로 확장하고 범용적 이종 병렬처리 연산 시스템을 제안한다. 또한, 최적화 기반의 일 분배 알고리즘을 제안하여 연산 시스템의 활용 효율을 극대화 한다. 제안하는 기술들은 다양한 근접질의에 적용되었으며, 두 개의 다중 코어 CPU와 네 개의 서로 다른 GPU를 사용하여 다중 코어 CPU 하나 사용 대비 10배 이상의 성능 향상을 얻을 수 있었다. 이는 본 연구의 범용성과 효용성을 증명하는 결과이다. 또한 본 논문은 연산 장치의 주기억 장치의 용량을 초과하는 대용량 데이터를 다루기 위한 아웃 오브 코어(out-of-core) 근접 질의 기술로 연구의 영역을 확장한다. 본 논문은 대규모 입자기반 유체 시뮬레이션처리를 위한 아웃 오브 코어 근접 질의 기술을 제안한다. 이를 통해 대규모의 입자를 안정적으로 처리 가능하며, 기존 아웃 오브 코어 기술 대비 최대 수십 배의 성능 향상을 얻을 수 있다. 이러한 연구 결과는 본 논문이 제안하는 기술들의 안정성과 효용성을 증명한다.

감 사 의 글

2008년 2월 이불을 둘러매고 카이스트로 들어선 후 어느덧 6년 반이라는 시간이 지나, 박사라는 이름을 가슴에 달고 나가게 되었네요. 지난 6년 반의 대학원 생활동안의 가장 큰 행운은 역시 지도교수님인 윤성의 교수님을 만난 것 인 것 같습니다. 윤성의 교수님의 훌륭한 지도 덕분에 박사 학위라는 긴 여정을 무사히 마칠 수 있었던 것 같습니다. 정말 감사합니다. 바쁘신 와중에도 학위 논문을 지도해주시고 많은 조언을 해주신 김영준, 박진아, 신인식, 허재혁 교수님께도 감사드립니다. 특히 같이 연구를 진행해 좋은 논문을 쓸 수 있게 도와주신 허재혁, 신인식, 김영준 교수님, 그리고 김동준, 홍정모 교수님께 깊은 감사의 마음을 전하고 싶습니다. 하루 중 가장 많은 시간을 부대끼며 서로를 응원하던 우리 SGLab. 동기, 선후배 분들. 좋은 일, 힘든 일, 그리고 각종 고민들까지 누구보다 먼저 들어주고 응원해준 정환이, 네가 있어 수많은 일들에도 큰 방황 없이 헤쳐 나갈 수 있었던 것 같아. 고맙다 친구. 유일한 선배님 김태준 박사, 그리고 우리 동기 보창, 용영, 재필, Pio. 연구실 작은 일 하나하나 같이 체계를 잡아가고, 논문 써보겠다고 같이 고생하던 일들이 이제 추억이 되어가네. 지난 6년간 같이 했는데 이제 자주 못 볼 수 도 있다고 생각하니 너무 아쉽다. 다들 서로 다른 곳에 자리를 잡더라도 꼭 자주자주 보자. 그리고 우리 후배님들! 연구실의 든든한 허리가 되어 주었던 종협, 근호, 종윤, 효섭, 영운, Lin, 그리고 오성형. 다들 졸업 후에 그 빈자리가 얼마나 크게 느껴졌는지 몰라요. 그리고 이제 곧 떠날 명환이 까지, 이제 저희가 어느덧 OB네요. 우리 각자 자신의 자리에서 큰 사람이 되어 SGLab.의 힘을 키워가요! 이제 우리 연구실의 진정한 기둥 동혁! 앞으로 연구실을 잘 부탁한다. 우리 연구실의 꽃들 가연, 수민, 밍양, 그리고 우리 든든한 명배, 창민. 많이 부족한 선배인데 잘 따라주어서 고마웠어. 앞으로 동혁이랑 같이 더욱 멋진 연구실 만들어가길 바래. 은재, 영현 그리고 남선생님, 연구실의 궂은일들을 잘 처리해 주셔서 연구에 집중 할 수 있었던 것 같습니다. 감사합니다.

대전 생활과 대학원 생활의 외로움을 잊게 해준 우리 SwingFever 식구 들. 한 사람 한 사람 다 이름을 적자니 너무 고마운 식구들이 많네요. 피버가 있어 수많은 고민과 스트레스를 땀으로 승화 시킬 수 있었던 것 같아요. 저 자신 보다 더 제 진로를 걱정해주기까지! 정말 감사합니다 :)

마지막으로, 공부 한다고 바쁜 오빠 대신 부모님 잘 챙기고 오빠까지 챙겨준 예쁘고 착한 동생 명선아, 정말 고맙고 사랑하다. 그리고 다 큰 아들이 서른이 넘도록 공부한다고 제대로 모시지도 못하는데도, 항상 믿고 응원해 주신 아버지, 어머니. 아들에 대한 두 분의 믿음과 응원이 없었다면, 지금의 저는 절대 있을 수 없었을 거예요. 이 감사의 마음을 표현하기에는 이 말 밖에는 없는 것 같네요. 아버지, 어머니 사랑합니다.

이 력 서

이　　　름 : 김 덕 수

생 년 월 일 : 1983년 1월 14일

E-mail 주 소 : bluekdct@gmail.com

학　　력

1998. 3. – 2001. 2.　김천고등학교

2001. 3. – 2008. 2.　성균관대학교 정보통신공학부 (B.S.)

2008. 2. – 2010. 2.　한국과학기술원 전산학과 (Integrated (M.S.))

학회활동

1. **Duksu Kim**, Jae-Pil Heo, and Sung-eui Yoon, *PCCD: Parallel Continuous Collision Detection*, ACM symposium on Interactive 3D Grahics and Games poster, Feb., 2009.
2. **Duksu Kim**, Jae-Pil Heo, JaeHyuk Huh, John Kim, and Sung-Eui Yoon, *HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs*,
   - Pacific Graphics, Oct., 2009., received a distinguished paper award.
   - SIGGRAPH poster, Aug., 2009.
   - High Performance Graphics poster, Aug., 2009.
3. **Duksu Kim**, Jinkyu Lee, Junghwan Lee, InSik Shin, John Kim, and Sung-eui Yoon, *Scheduling in Heterogeneous Computing Environments for Proximity Queries*,
   - GPU Technology Conference (GTC), 2013
   - ACM symposium on Interactive 3D Grahics and Games talk, Mar., 2014
4. **Duksu Kim**, Myung-Bae Son, Young J. Kim, Jeong-Mo Hong, and Sung-Eui Yoon, *Out-of-Core Proximity Computation for Particle-based Fluid Simulations*, High Performance Graphics, 2014
5. YongJoon Lee, Sung-eui Yoon, SeungWoo Oh, **Duksu Kim**, and SungHee Choi, *Multi-Resolution Cloth Simulation*, Pacific Graphics, Sept., 2010.
6. Jae-Pil Heo, Joon-Kyung Seong, **Duksu Kim**, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon, *FASTCD: Fracturing-Aware Stable Collision Detection*, ACM SIGGRAPH/Eurographics Symp. on Computer Animation (SCA), 2010.
7. 김경아, **김덕수**, 윤성의, *도시 교통 시뮬레이션*, 한국컴퓨터그래픽스학회 / 컴퓨터그래픽스학회논문지, 17권 4호, pp. 23-31, 2011.

연 구 업 적

1. **Duksu Kim**, Jae-Pil Heo, JaeHyuk Huh, John Kim, and Sung-Eui Yoon, *HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs*, Computer Graphics Forum, vol. 28, no. 7, pp. 1791–1800, Oct., 2009.
2. **Duksu Kim**, Jinkyu Lee, Junghwan Lee, InSik Shin, John Kim, and Sung-eui Yoon, *Scheduling in Heterogeneous Computing Environments for Proximity Queries*, IEEE Transactions on Visualization and Computer Graphics, vol. 19, no. 9, pp. 1513–1525, 2013.
3. **Duksu Kim**, Myung-Bae Son, Young J. Kim, Jeong-Mo Hong, and Sung-Eui Yoon, *Out-of-Core Proximity Computation for Particle-based Fluid Simulations*, Dept. of CS, KAIST, Technical Report CS-TR-2014-385, 2014
4. YongJoon Lee, Sung-eui Yoon, SeungWoo Oh, **Duksu Kim**, and SungHee Choi, *Multi-Resolution Cloth Simulation*, Computer Graphics Forum, vol. 29, no. 7, pp. 2225–2232, Sept., 2010.
5. Tae-Joon Kim, Bochang Moon, **Duksu Kim**, and Sung-Eui Yoon, *RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies*, IEEE Transactions on Visualization and Computer Graphics, vol. 16, no. 2, pp. 273–286, 2010