Hybrid Parallel Computation for Proximity Queries

Duksu Kim, Jinkyu Lee, Junghwan Lee, Insik Shin, John Kim, Sungeui Yoon
Dept. of CS, KAIST, Technical Report CS-TR-2012-368, (Submitted to TVCG)

Abstract—We present a novel, Linear Programming (LP) based scheduling algorithm that exploits heterogeneous multi-core architectures such as CPUs and GPUs to accelerate a wide variety of proximity queries. To represent complicated performance relationships between heterogeneous architectures and different computations of proximity queries, we propose a simple, yet accurate model that measures the expected running time of these computations. Based on this model, we formulate an optimization problem that minimizes the largest time spent on computing resources, and propose a novel, iterative LP-based scheduling algorithm. Since our method is general, we are able to apply our method into various proximity queries used in five different applications that have different characteristics. Our method achieves an order of magnitude performance improvement by using four different GPUs and two hexa-core CPUs over using a hexa-core CPU only. Unlike prior scheduling methods, our method continually improves the performance, as we add more computing resources. Also, our method achieves much higher performance improvement compared with prior methods as heterogeneity of computing resources is increased. Moreover, for one of tested applications, our method achieves even higher performance than a prior parallel method optimized manually for the application. We also show that our method provides results that are close (e.g., 75%) to the performance provided by a conservative upper bound of the ideal throughput. These results demonstrate the efficiency and robustness of our algorithm that have not been achieved by prior methods. In addition, we integrate one of our contributions with a work stealing method. Our version of the work stealing method achieves 18% performance improvement on average over the original work stealing method. This result shows wide applicability of our approach.

Index Terms—Heterogeneous system, proximity query, scheduling, collision detection, ray tracing, motion planning.

1 Introduction

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 [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.

There have been numerous attempts to accelerate the queries. One of the most general approaches is adopting an acceleration hierarchy such as kd-trees [2] or bounding volume hierarchies (BVHs) [1], [3]. 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.

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 [4]. Currently commodity CPUs have up to four or eight cores and GPUs have hundreds of cores [5]. 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 [6]. However, prior acceleration techniques such as using acceleration hierarchies do not consider utilizing such parallel architectures and heterogeneous computing systems. Since we are increasingly seeing more heterogeneous computing systems, it is getting more important to utilize them for various applications, including proximity queries, in an efficient and robust manner.

Main contributions: 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 main common 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

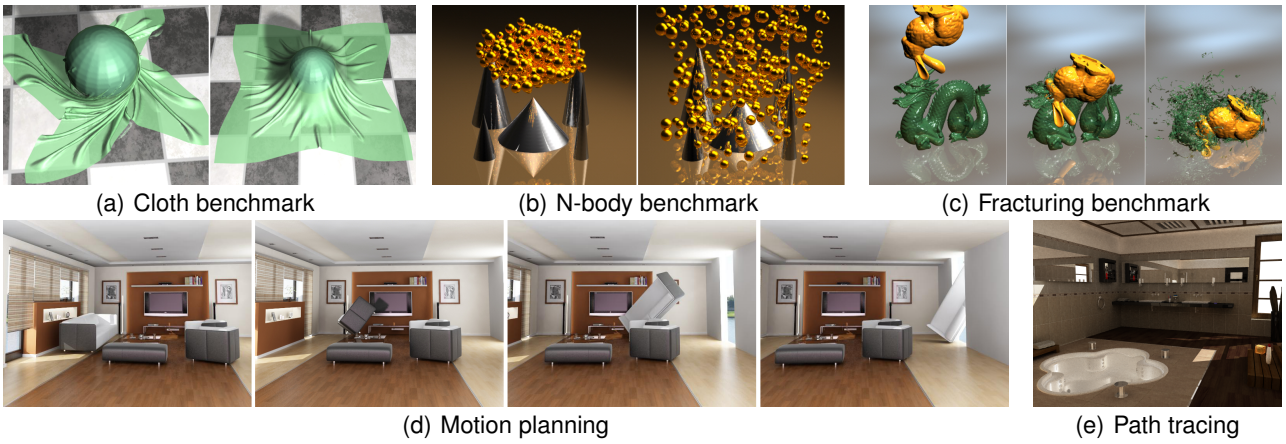

Fig. 1: *This figure shows five different benchmarks, whose proximity queries are parallelized by using CPUs and GPUs within our hybrid parallel framework. Different computations of these queries are automatically distributed by our LP-based scheduler without any parameter tuning. Compared to using a hexa-core CPU with six CPU threads, our method achieves one order of magnitude performance improvement by using two hexa-core CPUs and four different GPUs.*

proximity computations to the heterogeneous computing resources (Sec. 3). 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), 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. 5). In the tested benchmarks (Fig. 1), 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, while eliminating one of the manually tuned parameters that strongly affect its performance (Sec. 5.5).

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.

2 Related Work

Recently, general programming and execution models for heterogeneous architectures have been proposed [7], [8] and these models can be adopted in many different applications. However, designing efficient workload balancing algorithms is still left to users. We briefly discuss prior work on scheduling and parallelization techniques designed for general applications and proximity queries.

2.1 Scheduling for Unrelated Parallel Systems

Scheduling concerns allocating jobs to resources for achieving a particular goal, and has been extensively studied [9]. 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 [10]. These approximate methods achieve polynomial time complexity by using a linear programming (LP), which relaxes the integer constraint on the IP [11], [10], [12]. Lenstra et al. [10] 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. Nahapetian et al. [13] designed a polynomial time, approximate scheduling algorithm. This method clusters job allocations with similar makespan into a representative job allocation given an initial set of tasks. It then refines the clustered result with additional tasks; this work does not formulate the scheduling problem with the LP. Al-Azzoni et al. [14] proposed approximated LP-based scheduling algorithms. Since gathering various information from all the resources incurs a significant overhead, they considered information (e.g., the minimum completion time) from only a subset of resources in their heterogeneous system. 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.

Work stealing is a well-known workload balancing algorithm in parallel computing systems [15], [16]. Kim et al. [17] showed that the work stealing method achieved a near-linear performance improvement up to eight CPU cores for continuous collision detection. Hermann et al. [18] 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 [18], [19]. We compare our scheduling method with a basic work stealing approach (Sec. 5) and improve the efficiency of the working stealing method based on one of our contributions (Sec. 5.5).

2.2 Performance Models

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 [8]. Few works gave attention to modeling overheads such as data communication costs [13]. Performance models for GPUs [20], [21] and heterogeneous framework [22], [23] recently have been proposed. These architectural performance models can be very useful to accurately predict the computational time of jobs in a broad set of GPUs.

In this paper we use a simple performance model of jobs occurred in different proximity queries. Our performance model can be efficiently computed with observed performance data and can be effectively incorporated within our LP-based scheduling method.

2.3 Parallel Proximity Queries and Scheduling

Many scheduling methods have been also proposed for various proximity computations on parallel systems. Tang et al. [24] group jobs into blocks and assign a block to each idle CPU thread in a round robin fashion. Lauterbach et al. [25] check the workload balance among cores on a GPU and perform 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. [26] 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. Kim et al. [17] 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 scheduling methods, our method 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. 5.

3 Overview

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

3.1 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 paper; kd-trees or other acceleration hierarchies can be used with our method as well.

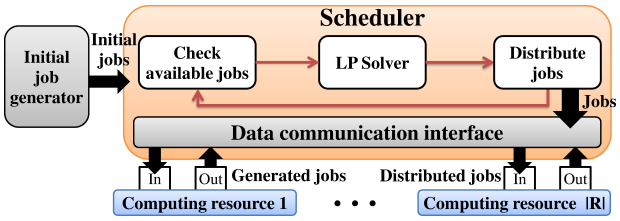

Fig. 2: *Overview of our hybrid parallel framework*

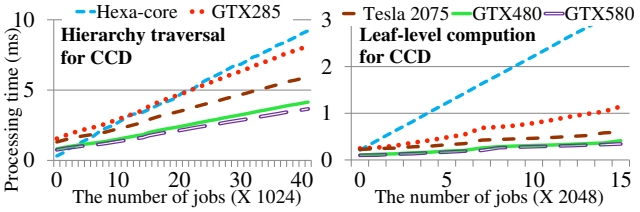

Fig. 3: *This figure shows observed processing time of two different job types on five different computing resources as a function of the number of jobs.*

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).

3.2 Our Hybrid Parallel Framework

Fig. 2 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. More low-level implementation details about our hybrid parallel framework are available in Appendix A as a supplementary material.

4 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.

4.1 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.

unit	Hexa-core CPU		GTX285		Tesla2075		GTX480		GTX580		T_{trans}	
(nanoseconds)	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	11.068	0.219	260.521	0.190	242.177	0.137	135.622	0.096	124.978	0.085	0.003	0.006
Leaf-level	2.766	0.098	81.855	0.030	136.26	0.0127	43.04	0.010	42.017	0.009	0.002	0.003

TABLE 1: *This table shows constants of our linear formulation computed for continuous collision detection.*

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. 3, 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 hierarchical traversal 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 leaf-level computation. 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:

$$T(k \rightarrow i, j, n_{ij}) = \begin{cases} 0, & \text{if } n_{ij} \text{ is } 0 \\ T_{setup}(i,j) + T_{proc}(i,j) \times n_{ij} & \\ \quad + T_{trans}(k \rightarrow i, j) \times n_{ij}, & \text{otherwise.} \end{cases} \quad (1)$$

$T_{setup}(i,j)$ represents the minimum time, i.e. *setup cost*, required to launch a module that processes 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 paper, 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. (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 1.

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 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. (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:

$$Minimize\ L,$$

$$subject\ to\ T_{rest}(i) + \Sigma_{j=1}^{|J|} T(i,j,n_{ij}) \leq L, \forall i \in R \quad (2)$$

$$\Sigma_{i=1}^{|R|} n_{ij} = n_j, \forall j \in J \quad (3)$$

$$n_{ij} \in \mathbb{Z}^+ (zero\ or\ positive\ integers). \quad (4)$$

This optimization formulation leads to find values of n_{ij} that minimize L under the three constraints, from Eq. (2) to Eq. (4). The first constraint (Eq. (2)) defines L as the makespan. The second constraint of Eq. (3) makes sure that there is neither missing nor redundant jobs. Finally, the third constraint (Eq. (4)) ensures that the result values of n_{ij} are restricted to zero or positive integer numbers.

4.3 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)) 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) [27]. 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 [28].

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. (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:

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

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 the overhead is relatively decreased as the number of jobs are increased.

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. A detailed work-flow of our iterative LP solver on an example of scheduling problem is available in Appendix B as a supplementary material.

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.

4.4 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 2 shows the benefit of our iterative solver. As we will see later our method achieves higher improvement

	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 2: *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 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.*

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**	**N-body**	**Fract.**	**Path tracing**	**Motion**
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 3: *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 (Fig. 1).*

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 [29]. However, we found that it increases almost linearly as a function of the number of resources in practice. 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 3) 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 3).

5 Results and Discussions

We have implemented our hybrid parallel framework and scheduling algorithm (*Ours(Exp.+LP)*) in three different machine configurations (Table 3). As we discussed in the hierarchical scheduling method (Sec. 4.4), we treat the identical computing units in a device as a single computing resource. We then use simple work stealing method [17] within a multi-core CPU, and even distribution method [25] 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. 4.1). Therefore, $|J|$ becomes 12.

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 [32]. 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 [25], [17]. We use the OpenMP library [33] and CUDA [5] 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. [25]. 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. [24]. Also, we have implemented a work stealing (*WS*) algorithm [17] 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. Also, we guide each resource to steal preferred jobs first (e.g., hierarchy traversal for multi-core CPUs, leaf-level computation for GPUs). 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 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(a)), N-body simulation (146 K triangles, Fig. 1(b)), and fracturing simulation (252 K triangles, Fig. 1(c)). 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(d)) [34], 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(e)) is ray traced with incoherent rays generated by a Monte Carlo path tracer [35]. 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 start with performing a bounding volume (BV) overlap test between them 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 [17], [25]. This step takes less than 1 ms in the tested benchmarks.

5.1 Results and Analysis

Fig. 4 and Fig. 5 show the performance of various proximity queries that are parallelized with different scheduling methods in two machine configurations (Machine 1 and 2 in Table 3). We use the line plot instead of bar graph in order to highlight the performance trend as we add more computing 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 3), 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

1. LINDO systems (http://www.lindo.com)
2. UNC dynamic benchmarks (http://gamma.cs.unc.edu/DYNAMICB)

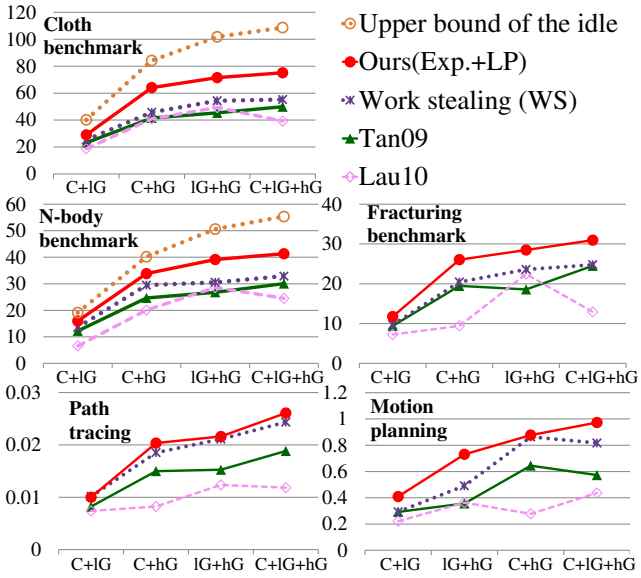

Fig. 4: *This figure shows the throughput, frames per second, of our hybrid parallel framework, as we add a CPU and two GPUs, in the tested benchmarks. (**C** = a quad-core CPU, **lG** = GTX285 (i.e. low-performance GPU), **hG** = GTX480 (i.e. high-performance GPU))*

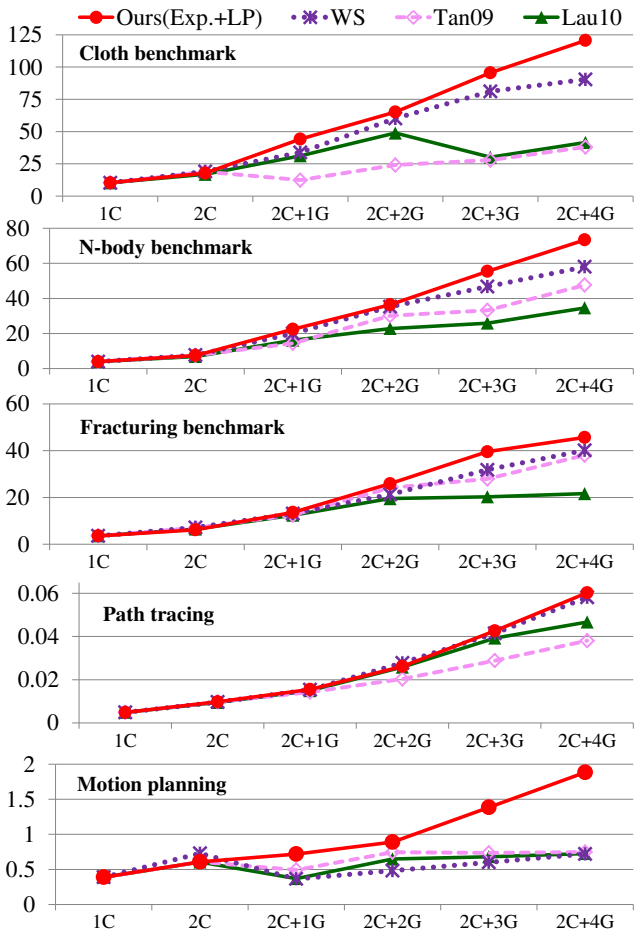

Fig. 5: *This figure shows the throughput, frames per second, of ours and prior scheduling algorithms, as we add more computing resources. (1C = a hexa-core CPU, 1G = GTX285, 2G = 1G+Tesla2075, 3G = 2G+GTX480, 4G = 3G+GTX580)*

to 19 times improvement by using two hexa-core CPU and four different GPUs (Fig. 5).

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). In this case, **C** has relatively lower capacity than other GPUs. However, *Lau10* does not consider the relative capacity difference and assigns 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). 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). On average, our LP-based algorithm shows 22% and 36% higher performance than *WS* in Fig. 4 and Fig. 5 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. 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. 5, 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

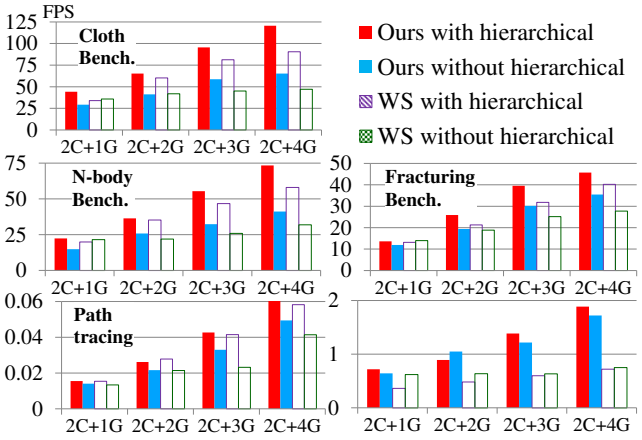

Fig. 6: *This figure shows frames per second of our LP-based scheduling and work stealing method with/without hierarchical scheduling on Machine 2.*

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.

Benefits of hierarchical scheduling: Fig. 6 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). 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.

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

5.2 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 our paper 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 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. 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*,

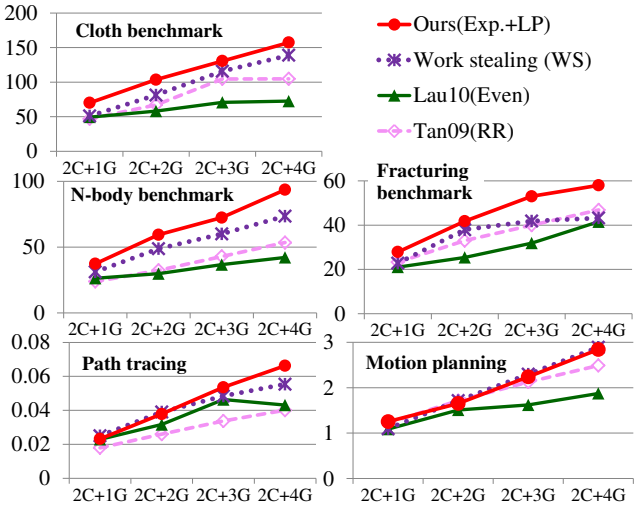

Fig. 7: *This figure shows the performance of tested scheduling approaches on a near-homogeneous computing system consisting of two hexa-core CPUs and four identical GPUs (Machine 3 in Table 3).*

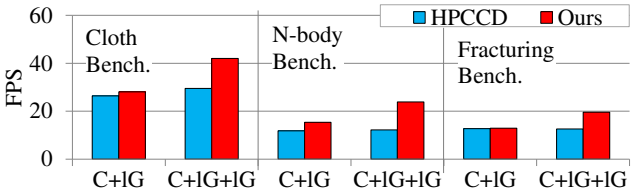

Fig. 8: *This figure compares the performance of our method with HPCCD, which is optimized specifically for the tested application, continuous collision detection. The throughput, frames per second, includes hierarchy update time.*

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 where our method can be improved further, we have studied the under-utilization ratios of each computing resource with our LP-based algorithm. Specifically, we measure how much each computing resource stays in the idle status; a computing resource can be idle when waiting for jobs from the scheduler. We found that the idle time takes a small portion (13% on average) when we use our LP-based algorithm with hierarchical scheduling (Table 4). We also measure how much time our scheduler running on the CPU takes, compared to other working threads running on the same CPU. It takes about 7% of total running time of those working threads. This also indicates that our scheduling method has low computational overhead.

5.3 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 systems, we compare ours and prior approaches in a near-homogeneous system consisting of two hexa-core CPUs and four identical GPUs. Fig. 7 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.

5.4 Comparison to a Manually Optimized Method

Only a few works [26], [17] 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 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 paper, 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) [17]. 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. 8). 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.

5.5 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. 5.1. 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 3) 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.

6 Conclusion

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.

6.1 Limitations and Future Work

It is evident that future architectures will have more computing resources. We have demonstrated the performance with machines consisting of up to six different computing resources and discussed its optimality with up to three different 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 under-utilization 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 depending on chosen benchmarks and machine configurations. A further investigation is required to minimize the overhead and robustly handle local minimum issues.

In addition, we plan to study more on hierarchical scheduling and would like to extend it to a multi-resolution scheduling method for large-scale heterogeneous computing systems like cloud computing. 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. 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 [36]. Also, 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. Finally, we would like to extend our method to other general applications that have more variety of jobs.

We believe that maximally utilizing heterogeneous multi-core architectures is one of the most challenging problems.

We wish that our work makes a step towards it in the context of proximity computation.

REFERENCES

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

[2] 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.

[3] 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.

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

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

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

[7] G. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Symp. on High performance distributed computing*, 2008, pp. 197–200.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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," *Computer Graphics Forum (EG)*, vol. 28, no. 2, pp. 385–396, 2009.

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

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

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

[30] 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.

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

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

[33] 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.

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

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

[36] 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.

Duksu Kim is currently a Ph.D student at KAIST (Korea Advanced Institute of Science and Technology). He received his B.S. degree in information & communication engineering from Sung Kyun Kwan University in 2008. His research interests include collision detection, motion planning, and parallel computing. He received the distinguished paper award at the 17th Pacific Conference on Computer Graphics and Applications (Pacific Graphics) in 2009.

Jinkyu Lee received B.S., M.S. and Ph.D. degrees in Computer Science in 2004, 2006 and 2011, respectively, from KAIST (Korea Advanced Institute of Science and Technology), South Korea. Now, he is a visiting scholar at University of Michigan, Ann Arbor, MI, USA. His research interests include reliability, power management and timing guarantees in real-time embedded systems. He won the best student paper award from the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2011.

Junghwan Lee is currently a Ph.D student at the Dept. of Computer Science at KAIST (Korea Advanced Institute of Science and Technology), South Korea. He received his B.S. degree in computer science from Yonsei University in 2006. His research interests include motion planning and collision detection.

Insik Shin is currently an assistant professor in Dept. of Computer Science at KAIST, South Korea, since 2008. He received a B.S. from Korea University, an M.S. from Stanford University, and a Ph.D. from University of Pennsylvania all in Computer Science in 1994, 1998, and 2006, respectively. He has been a post-doctoral research fellow at Malardalen University, Sweden, and a visting scholar at University of Illinois, Urbana-Champaign until 2008. His research interests lie in cyber-physical systems and real-time embedded systems. He is currently a member of Editorial Boards of Journal of Computing Science and Engineering. He has been co-chairs of various workshops including satellite workshops of RTSS, RTAS, and RTCSA and has served various program committees in real-time embedded systems, including RTSS, RTAS, ECRTS, and EMSOFT He received best paper awards, including the Best Paper award from RTSS in 2003 and the Best Student Paper Award from RTAS in 2011, and Best Paper runner-ups at ECRTS (IEEE Euromicro Conference on Real-Time Systems) and RTSS in 2008.

John Kim is currently an assistant professor in the Department of Computer Science at KAIST with joint appointment in the Web Science and Technology Division at KAIST. He received his B.S. and M.Eng. from Cornell University in the Department of Electrical Engineering in 1997 and 1998. He spent several years working on the design of different microprocessors at Motorola and Intel. He then received his Ph.D. from Stanford University in 2008 from the Department of Electrical Engineering. His research interests includes multicore architecture, interconnection networks, and datacenter architecture. He is a member of IEEE and ACM.

Sung-Eui Yoon is currently an IWON associate professor at KAIST. He received the B.S. and M.S. degrees in computer science from Seoul National University in 1999 and 2001 respectively. He received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 2005. He was a postdoctoral scholar at Lawrence Livermore National Laboratory. His research interests include proximity queries, interactive rendering, geometric problems, and cache-coherent algorithms. He is particularly interested in designing scalable algorithms that can handle massive models in commodity hardware. He wrote a monograph on real-time massive model rendering with other three co-authors. He also gave numerous tutorials on proximity queries and large-scale rendering at various conferences including ACM SIGGRAPH and Eurographics. Some of his work received a distinguished paper award at Pacific Graphics, invitations to IEEE TVCG, an ACM student research competition award, and other domestic research-related awards. He is a member of IEEE, ACM, and Eurographics.