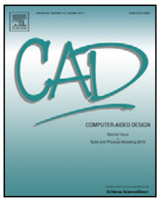
Computer-Aided Design 43 (2011) 1360–1369

Contents lists available at SciVerse ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad



A hybrid parallel solver for systems of multivariate polynomials using CPUs and GPUs

Cheon-Hyeon Park[a], Gershon Elber[b], Ku-Jin Kim[c], Gye-Young Kim[d], Joon-Kyung Seong[d,*]

[a] *Nexon Corporation, Republic of Korea*
[b] *Department of Computer Science,Technion-Israel Institute of Technology, Israel*
[c] *School of Computer Science and Engineering, Kyungpook National University, Republic of Korea*
[d] *School of Computer Science and Engineering, Soongsil University, Republic of Korea*

A R T I C L E   I N F O

Keywords:
Geometric constraint solver
Hybrid algorithm
Non-linear system
Graphics hardware
Bezier subdivision

A B S T R A C T

This paper deals with a problem of finding valid solutions to systems of polynomial constraints. Although there have been several quite successful algorithms based on domain subdivision to resolve this problem, some major issues are still demanding further research. Prime obstacles in developing an efficient subdivision-based polynomial constraint solver are the exhaustive, although hierarchical, search of the zero-set in the parameter domain, which is computationally demanding, and their scalability in terms of the number of variables. In this paper, we present a hybrid parallel algorithm for solving systems of multivariate constraints by exploiting both the CPU and the GPU multicore architectures. We dedicate the CPU for the traversal of the subdivision tree and the GPU for the multivariate polynomial subdivision. By decomposing the constraint solving technique into two different components, hierarchy traversal and polynomial subdivision, each of which is more suitable to CPUs and GPUs, respectively, our solver can fully exploit the availability of hybrid, multicore architectures of CPUs and GPUs. Furthermore, our GPU-based subdivision method takes advantage of the inherent parallelism in the multivariate polynomial subdivision. We demonstrate the efficacy and scalability of the proposed parallel solver through several examples in geometric applications, including Hausdorff distance queries, contact point computations, surface–surface intersections, ray trap constructions, and bisector surface computations. In our experiments, the proposed parallel method achieves up to two orders of magnitude improvement in performance compared to the state-of-the-art subdivision-based CPU solver.

1. Introduction

Finding valid solutions to systems of multivariate polynomials is a core operation in many applications in science and engineering. In particular, many problems in geometric modeling can be reduced to the problem of finding solutions to systems of equality/inequality polynomial constraints. For example, the closest points between two freeform surfaces can be computed by solving a system of four polynomial equations with four unknowns. Similarly, intersecting two freeform surfaces can be represented by a system of three equations with four unknowns. Other applications include bisectors, offsets, shape matching, and morphing, to name a few.

Although there have been several quite successful algorithms for constraint solving, finding solutions to a set of non-linear constraints is a non-trivial task. In recent years, however, non-linear constraint solvers that are based on domain subdivision have been successfully used in solving quite a few geometric problems [1,2]. The subdivision-based solvers find roots of multivariate polynomial constraints by recursively subdividing the parameter domain in a hierarchical manner. Based on the ability to exhaustively examine the specified domain of a constraint for zeros, the algorithm is highly robust and effective in finding all solutions, providing a global answer. On the other hand, the exhaustive search is computationally demanding, which therefore limits possible applications to geometric problems that do not require interactive performance. This issue is critical if the number of variables increases in the polynomial constraints: the size of a tensor product constraint grows exponentially with its number of variables [3]. Efficient search of the parameter domain is important in the design of a highly scalable algorithm for a subdivision-based solver, which therefore extends its availability to real-time applications with many degrees of freedom.

* Correspondence to: Room #328, Information Science Building, Soongsil University, 511 Sangdo-dong, Dongjak-gu, Seoul 156-743, Republic of Korea. Tel.: +82 2 820 0911; fax: +82 2 822 3622.
E-mail addresses: cheonhyeon@gmail.com (C.-H. Park), gershon@cs.technion.ac.il (G. Elber), kujinkim@yahoo.com (K.-J. Kim), gykim11@ssu.ac.kr (G.-Y. Kim), seong@ssu.ac.kr, seong@cs.utah.edu (J.-K. Seong).

0010-4485/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2011.08.030

In order to alleviate the computational complexity of the subdivision-based solvers, there have been several suggestions exploiting either triangular hyperpatches [4] or expression trees [3]. Although the method proposed by [4] could reduce the size of the constraint using barycentric blending functions, it still remains exponential in terms of the number of variables. Recently, [3] has proposed an alternative representation scheme for multivariate constraints, which is based on expression trees. This method successfully shows polynomial growth in the size of the constraint, while it only deals with the zero-dimensional solution space.

In this work, we present a different alternative to further alleviate the computational complexity of subdivision-based solvers by exploiting the hybrid, multicore architectures of both CPUs and GPUs. Our approach is based on a simple observation: adaptive traversal of the hierarchy of the domain subdivision is suitable to the CPU, while the process of multivariate function subdivision fits the GPU. We therefore dedicate the CPU for the traversal of the subdivision tree and the GPU for the polynomial subdivision. Furthermore, our GPU-based subdivision method takes advantage of the inherent parallelism in the multivariate function's subdivision. We demonstrate the efficacy and scalability of the proposed parallel solver through several examples in geometric applications, including Hausdorff distance queries, contact point computations, surface–surface intersections, ray trap constructions, and bisector surface computations. In our experiments, the proposed parallel method achieves up to two orders of magnitude improvement in performance compared to the state-of-the-art subdivision-based CPU solver.

The remaining part of this paper is organized as follows. In Section 2, we review previous work that are mostly related to our approach, and we present an overview of our method in Section 3. In Section 4, we introduce the CPU-based parallel algorithm for the traversal of the subdivision tree, while the GPU-based polynomial subdivision method is described in Section 5. In Section 6, some examples are shown, comparing the proposed parallel solver to the previous CPU-based subdivision solvers. Finally, we conclude in Section 7.

2. Previous work

Since Lane and Riesenfeld [5] suggested a subdivision based solver for a univariate function in the Bernstein–Bézier basis, subdivision methods have been intensively investigated [1,6,7,2,8,9]. Subdivision-based root-finders repeatedly subdivide the input polynomials to isolate and even approximate roots. Given the input polynomials and a domain to seek the roots in, the method recursively subdivide this domain. The subdivision is terminated when it is assured that there is no root in the domain or the domain's size is smaller than a user-provided threshold. Most of subdivision methods are working on equations that are represented by the Bézier or *B*-spline basis functions, and utilize their convex hull and subdivision properties [10]. The convex hull property assures that the curve or multivariate function lies entirely within the convex hull of its control polygon. Based on this property, subdivision methods robustly find all roots without requiring a good initial value on a given interval.

Lane–Riesenfeld [5] couples recursive bisection with properties of the Bernstein form to isolate and eventually approximate roots to a specified precision over an interval. Rockwood [11] presents a variation on the Lane and Riesenfeld method suitable for parallel and vector processing. The estimate becomes a Newton step in the vicinity of the approximate root, exhibiting quadratic convergence for single roots. Schneider [12], also a variation on Lane–Riesenfeld, determines a root when either the recursion depth limit is reached, or when the control polygon crosses the abscissa once and approximates a straight line, or is flat enough.

Nishita et al. [8] introduced a Bézier clipping technique for the efficient computation of roots of univariate Bézier functions. Bartoň and Jüttler [7] have recently presented an algorithm which is able to compute all roots of a given univariate polynomial using a quadratic clipping. In the case of single roots, they have a convergence rate of three, while the convergence rate is still superlinear for double roots.

Sherbrooke and Patrikalakis [9] presented a domain clipping approach to solving a set of multivariate polynomial equations. They reduce the domain via domain clipping to a much smaller subdomain at each subdivision level. Although one can get a much smaller subdomain at each subdivision, the cost of this domain clipping is high, so there is a trade-off between the reduction of domain size and the computation cost. Mourrain and Pavone [2] improved the method proposed by Sherbrooke and Patrikalakis [9] by exploiting a domain reduction strategy, and showed a local quadratic convergence speed for simple roots.

Given a set of multivariate rational equations represented by *B*-spline basis functions, Elber and Kim [1] presented a subdivision-based constraint solving technique. They bisect the domain along every axis. When the problem has many roots, a large number of subdivision steps are required at an early stage of the algorithm. They also eliminates redundant subdivisions at the final stage by guaranteeing that there is at most one root in each subdomain. Domain clipping has no explicit mechanism to guarantee such a condition. The concepts of the normal cone and the surface bounding cone [13,14] are employed for this purpose. They generalize these tools to higher dimensions for the intersection of multivariate implicit hypersurfaces [1,6]. When they have isolated all the different roots using the surface bounding cones, they terminate the subdivision procedure and switch to Newton–Raphson iterations, for faster, quadratic, convergence at each simple root.

Recently, a hybrid method has been presented to resolve continuous collision detection (CCD) for mesh surfaces by exploiting both CPUs and GPUs [15]. In their method, the bounding volume hierarchy (BVH) is traversed in parallel using multicores of the CPU. Lauterbach et al. further refines this method by implementing the traversal of the BVH on GPUs, and thus reduces overheads caused by data transmission between the CPU and the GPU [16]. Although, the work by [16] could reduce additional computation time for transferring data between the two processing units, dynamic generation of child nodes during traversal of the BVH needs to be performed in serial with the task of CCD between bounding volumes. Unlike traversing the BVH for CCD, however, the geometric data (control points) of a multivariate function in child nodes are created dynamically for traversing the subdivision tree, which may result in performance degradation for the multivariate polynomial solver.

3. Overview

The proposed parallel constraint solver adopts the framework of a subdivision-based constraint solving technique to exploit multicore architectures of CPUs and GPUs. In this section, we first describe the framework of a subdivision-based solver, and then present an overview of our parallel solver.

3.1. Subdivision solvers

Given n multivariate constraints $\mathcal{F}_i(u_1, \dots, u_m) = 0, i = 1, \dots, n$, in R^m, where $n \leq m$, we consider all simultaneous solution points, $u = (u_1, \dots, u_m)$, such that $\mathcal{F}_i(u) = 0$, for all $i = 1, \dots, n$. In our problem setting, the polynomial constraints $\mathcal{F}_i$ will be represented as piecewise parametric Bézier basis functions. Then, the parameterization of the multivariate

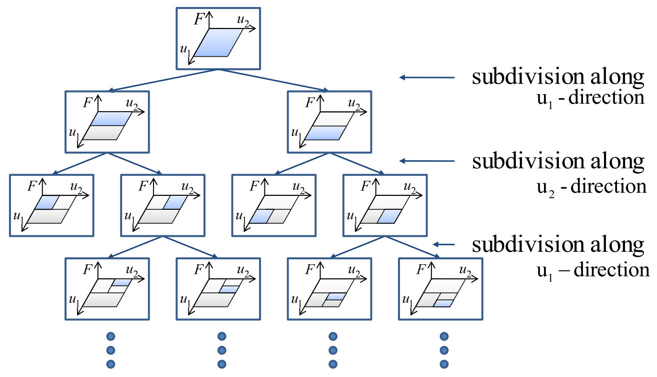


Fig. 1. An illustration of traversing the subdivision tree for bi-variate functions.

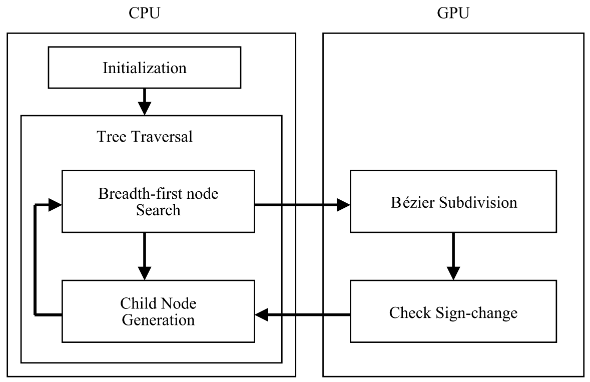


Fig. 2. An overview of our hybrid parallel constraint solver.

functions will implicitly define the parametric domain R^m of the given system. Due to the convex hull property of the Bézier basis functions, the parametric domain contains zeros of $\mathcal{F}_i$ only if the control coefficients of $\mathcal{F}_i$ have different signs over the domain. Based on this property, a subdivision-based solver finds robustly the simultaneous zero-set of $\mathcal{F}_i(u),\ i = 1, \ldots, n$, by recursively subdividing the parameter domain. Fig. 1 shows an example of this recursive subdivision process for functions $\mathcal{F}_i(u_1, u_2)$ with two variables.

Considering this subdivision process as a tree traversal, the root node of the subdivision tree contains the whole parameter domain (see Fig. 1). The solver subdivides the domain of the parent node into two sub-domains along a parameter direction one by one. In Fig. 1, the root node is subdivided along the u_1-direction, of which child nodes contain sub-domains with half sizes along the u_1-direction. Subdividing the domain along one parameter direction involves n function subdivisions for each $\mathcal{F}_i(u),\ i = 1, \ldots, n$: at each level of the subdivision tree, if the control coefficients of $\mathcal{F}_i$ have different signs, for all $i = 1, \ldots, n$, then the solver subdivides every $\mathcal{F}_i$ into two functions $\mathcal{F}_i^{left}$ and $\mathcal{F}_i^{right}$ for testing zeros at the next level. The subdivision process will be terminated if the size of the parameter domain is less than the user-provided tolerance, τ, in every direction $u_i,\ i = 1, \ldots, n$. The result of this subdivision will be a set of discrete cells in R^m, each of which has size of less than τ in every coordinate direction: each such cell may generate one point, the center of each cell, which potentially contributes to the solution space.

3.2. Overview of our method

Being inherited from the subdivision-based constraint solving technique, the basic framework of our constraint solver is the same as a subdivision-based solver. As illustrated in Fig. 2, the proposed parallel algorithm consists of two parts: CPU-based traversal of the subdivision tree and GPU-based multivariate function subdivision.

First, multiple CPU cores are exploited for the traversal of the subdivision tree, which adaptively generate child nodes at every level. In order to design a highly scalable algorithm, we adopt a task management scheme proposed by [15] to our problem setting: we decompose the tree traversal into a set of independent task units, which enable lock-free parallel tree traversal. Unlike the work by [15], our task management scheme is based on a single ring buffer data structure operated with atomic instructions, which greatly reduces the number of memory I/Os.

Second, at each level of the subdivision tree, GPUs perform the actual subdivision of the multivariate functions, in parallel. This algorithm takes advantage of the inherent parallelism in the Bézier basis function's subdivision. After the polynomial subdivision, the control coefficients of each function are checked for sign change, and the results are transferred to the CPU, which will be used for node generation at the next level of the subdivision tree.

4. CPU-based traversal of the subdivision tree

The CPU-based tasks consist of two components (Fig. 2): initialization and traversal of the subdivision tree. The initialization step is done at once, while nodes in the subdivision tree are traversed recursively in the main loop of our solver. For initialization, the solver sets up a system of multivariate functions, given a geometric application of interest: the root node of the subdivision tree contains every function $\mathcal{F}_i(u),\ i = 1, \ldots, n$, defined over the whole parameter domain. Now, starting from the root node, the solver searches solutions of the system by recursively traversing the subdivision tree. In this section, we concentrate our discussions on the tree traversal step.

4.1. Task decomposition

We first define a few terminologies to describe our method. We define a *node* in the subdivision tree so as to contain all the necessary information on the parameter domain for recursive zero-set search, which include range of the current parameter domain and memory addresses for storing control coefficients of every multivariate function. We denote by *task buffer* a one-dimensional ring buffer containing a set of *nodes*. Our method maintains a single *task buffer*, which is shared by multiple CPU threads during the tree traversal.

Our CPU-based algorithm decomposes the tree traversal into two independent task units: CPU–GPU interface and dynamic node generation. Having multiple CPU threads, these task units need to be carefully assigned to the threads in order to avoid performance degradation due to locks among multiple threads. To address this synchronization problem, there have been many efforts to eliminate the use of locks by designing lock-free algorithms based on either atomic swap instructions [17] or a task decomposition method [15]. Our tree traversal method combines the advantages of both approaches: the decomposed tasks are assigned to two different threads, mainThread and workThread, while the nodes in the subdivision tree are manipulated based on atomic instructions through the buffer in a lock-free manner.

The task of the CPU–GPU interface is assigned to the mainThread, while a set of workThreads is used for dynamic node generation. Fig. 3 shows an overall structure of our tree traversal method. At each level of the tree, the mainThread collects a set of *nodes* that are candidates of simultaneous zeros for all functions $\mathcal{F}_i$, and executes GPU kernels to subdivide the multivariate functions by transferring the candidate set of *nodes* to the GPU. At the same time, a set of workThreads generates child *nodes* in parallel. Specifically, given the results of the GPU-based function subdivision, a workThread pops a *node* from the *task buffer*, and determines whether the current *node* can contain zeros for all multivariate functions.

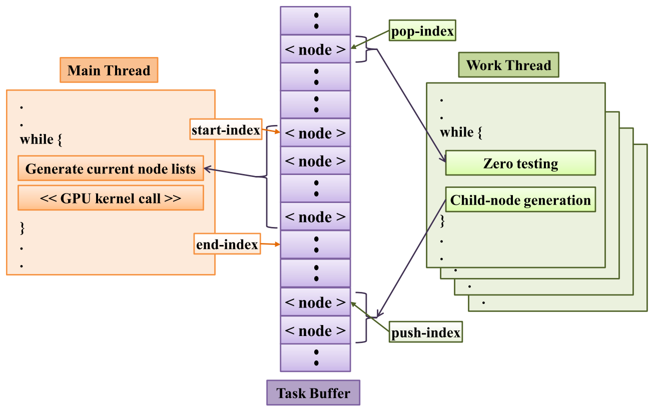


Fig. 3. An overall structure of our CPU-based algorithm for traversing the subdivision tree.

The workThread then generates child *nodes* if the current node potentially has simultaneous zeros, and pushes the child *nodes* into the *task buffer*. Both the mainThread and workThread perform their own tasks in parallel without intervention, in which the detailed parallel algorithm will be explained in the following section.

4.2. Parallel subdivision tree traversal

As illustrated in Fig. 3, at every level of the subdivision tree, multiple threads access a set of *nodes* simultaneously through the *task buffer*. The synchronization of multiple threads may result in latency in the tree traversal. To address this problem, our tree traversal method exploits atomic instructions: the mainThread and the workThread access the *node* data in the *task buffer* independently using four index variables. Specifically, the mainThread reads a set of *nodes* in the active range, (*start-index*, *end-index*), where the variables *start-index* and *end-index* indicate the starting and ending positions of the current active range in the subdivision tree, respectively. The two index variables are manipulated only by using atomic instructions, so that the mainThread always reads unvisited *nodes*. Once the mainThread transfers the extracted set of *nodes* to the GPU kernels, it increases *start-index* atomically by the number of the transferred *nodes*. The mainThread repeats this process iteratively until no *node* remains in the *task buffer*.

Independently of the mainThread, a set of workThreads accesses *node* data based on the other index variables, which are also manipulated using atomic instructions. A workThread tries to read a *node* indexed by *pop-index*. If it succeeds to read a *node*, it performs a simultaneous zero testing for the multivariate functions referenced by the *node* as explained in Section 4.1, and increases the index atomically by one. The workThread further processes the *node*: if the *node* potentially contains simultaneous zeros of the functions, then the workThread pushes the child nodes into the *task buffer* using the variable *push-index*, which will be again increased atomically by the number of child nodes. This process is repeated by a set of workThreads until reaching the leaf nodes of the subdivision tree.

The above tree traversal method is lock-free since multiple threads are synchronized based on the four atomically-manipulated index variables. The proposed method is different from the conventional approaches based on atomic swap instructions (such as [17]): our approach first decomposes the tree traversal into two independent task units, each of which is assigned to different threads. Then, the two independent threads work on the *task buffer* using their own atomic variables. Decomposing the task makes the system more flexible in dealing with the shared *node* data. Also, our tree traversal method is different from the task management scheme proposed by [15]: In their work, [15] could eliminate the use of locks by maintaining several queues attached to each thread, and by handling workloads of the threads using the master thread. Unlike this method, our algorithm maintains a single *task buffer* and both the mainThread and workThread share the *node* data through the buffer. Although sharing a single buffer by multiple threads looks inefficient than maintaining multiple independent task queues, we observed that the former actually reduced lots of overhead caused by the management of the multiple queues. We speculated that this observation would come from the fact that our workThread has relatively small size of workload: a workThread performs child node generation. Thus, instead of maintaining multiple queues, sharing a single task buffer and reducing both the additional data structures and the number of memory I/Os could result in much improvement in performance. In our experiments, we got five times of speedup on average when maintaining a single task buffer with the atomic variables, rather than using multiple queues without atomic instructions.

5. GPU-based multivariate Bézier subdivision

In this section, we present a data parallel algorithm for subdividing multivariate constraints represented as piecewise Bézier basis functions. For the clarity of the explanation, a parallel algorithm for subdividing univariate Bézier curves is first described, which is later extended to multivariate cases. We also present the CUDA implementation details of the proposed parallel subdivision algorithm.

5.1. Parallel Bézier curve subdivision

Consider a univariate (piecewise) homogeneous polynomial function $\mathcal{F}(u)$, where $\mathcal{F}(u)$ is represented as piecewise Bézier univariate scalar curves. A Bézier curve of degree n is given by the Bernstein representation,

$$\mathcal{F}(u) = \sum_{i=0}^{n} P_i \theta_{i,n}(u), \quad \theta_{i,n}(u) = \binom{n}{i} \frac{(b-u)^{n-i}(u-a)^i}{(b-a)^n},$$

where $\theta_{i,n}(u)$ is the i-th Bernstein basis function. Denoted by $\mathcal{F}[P_0, \ldots, P_n; a, b](u)$, the Bézier curve $\mathcal{F}(u)$ is defined with control coefficients $\{P_i\}$, $i = 0, 1, \ldots, n$, on the interval $[a.b]$. Now, a subdivision algorithm [18] splits the curve on $[a, b]$ into two curves of smaller intervals $[a, c]$ and $[c, b]$,

$$\mathcal{F}[P_0, \ldots, P_n; a, b](u) = \begin{cases} \mathcal{F}[P_0^{[0]}, P_1^{[1]}, \ldots, P_n^{[n]}; a, c](u) \\ \mathcal{F}[P_n^{[n]}, P_n^{[n-1]}, \ldots, P_n^{[0]}; c, b](u), \end{cases}$$

where

$$P_i^{[k]} = \begin{cases} \dfrac{(b-c)P_{i-1}^{[k-1]} + (c-a)P_i^{[k-1]}}{(b-a)} & k > 0,\ k \le i \le n, \\ P_i & k = 0. \end{cases}$$

After the application of this subdivision process, the curve is defined as the union of the two separate Bézier curves: instead of $n+1$ control points, there are $2n+2$ control points, $n+1$ for each part.

We now focus our attention on the process of control polygons during the subdivision of Bézier curves. The recursive sequence of control polygons $P_i^{[k]}$ at the k-th level is a convex combination of points on the $(k-1)$-th polygon (Fig. 4). We consider such recursive sequences as a subdivision graph for a point on the curve. All the original polygon points are at the leaves of the graph. The iteration is continued, with one less node at each level until the final node represents the function value. Fig. 4 shows this graph. In Fig. 4, notice that only $P_{j+1}^{[j+1]}$ and $P_{j+2}^{[j+1]}$ depend on

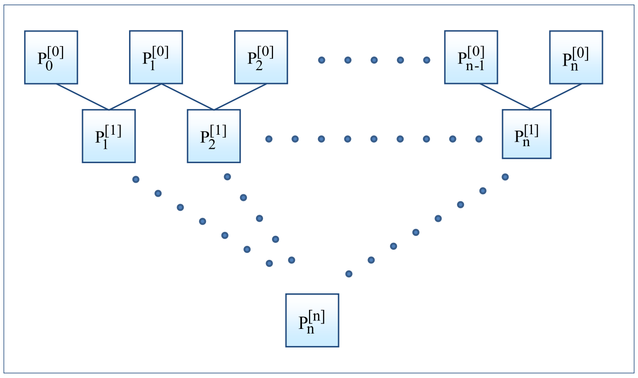


Fig. 4. A subdivision graph for a Bézier curve of degree n.

$P_{j+1}^{[j]}$, $j = 0, 1, \ldots, n-1$. This dependency in the subdivision graph is important in developing a parallel algorithm for subdividing Bézier curves.

Given $n+1$ control points $P_i^{[0]}$, $i = 0, 1, \ldots, n$, this recursive subdivision can be done using a conventional single-core algorithm: *Algorithm* 1 describes the pseudo code of the single-core algorithm. Assuming $n+1$ processors, on the other hand, the same subdivision process can be done in parallel by performing the convex combination of control polygons separately. *Algorithm* 2 describes the parallel version of the subdivision process: the statement *for all i in parallel do s end for* causes all processors to execute the same statement s in synchrony, but the variable i has a different value for each processor, namely, the index of that processor within the array of the processors. Clearly, *Algorithm* 1 has $O(n^2)$ time complexity, while the subdivision can be done in time $O(n)$ when using *Algorithm* 2 under the assumption that $n+1$ processors are available.

Algorithm 1 single-core subdivision

```
1: for $k \in 1, \ldots, n$ do
2:     for $i \in step \cdots n$ do
3:         $P_i^{[k]} = \frac{b-c}{b-a} P_{i-1}^{[k-1]} + \frac{c-a}{b-a} P_i^{[k-1]}$
4:     end for
5: end for
```

Algorithm 2 multi-core subdivision

```
1: for $k \in 1, \ldots, n$ do
2:     for all i in parallel do
3:         $P_i^{[k]} = \frac{b-c}{b-a} P_{i-1}^{[k-1]} + \frac{c-a}{b-a} P_i^{[k-1]}$
4:     end for
5: end for
```

5.2. *Parallel multivariate function subdivision*

In our problem setting of the parallel constraint solving technique, we assume a tensor product representation for multivariate scalar functions. The proposed hybrid framework, however, could be applied to other representation schemes, such as expression trees [3].

Extending the univariate curve subdivision method to subdivision of scalar functions with arbitrary number of variables raises several issues. First, the subdivision graph shown in Fig. 4 need to be constructed according to the subdivision axis. Suppose that a function $\mathcal{F}(u_1, \ldots, u_m)$ with m variables, each of which has degree n_i, $i = 1, 2, \ldots, m$, has a tensor product representation,

$$\mathcal{F}(u_1, \ldots, u_m) = \sum_{i_1=0}^{n_1} \sum_{i_2=0}^{n_2} \cdots \sum_{i_m=0}^{n_m} P_{i_1, i_2, \ldots, i_m} \theta_{i_1, n_1}(u_1) \cdots \theta_{i_m, n_m}(u_m).$$

The function $\mathcal{F}$ has a control mesh $\{P_{i_1, i_2, \ldots, i_m}\}$, $i_j = 1, 2, \ldots, n_j$, $j = 1, \ldots, m$. Now, assume that $\mathcal{F}$ is subdivided along the parameter axis u_j, $1 \le j \le m$. Then, this subdivision process can be considered as consecutive subdivisions of $\Pi_j = (n_1+1)(n_2+1) \cdots (n_{j-1}+1)(n_{j+1}+1) \cdots (n_m+1)$ numbers of isoparametric curves of $\mathcal{F}$. We therefore perform the curve subdivision of degree n_j for Π_j different isoparametric curves to subdivide $\mathcal{F}$ along the u_j-direction.

While this subdivision scheme seems complex at first glance, it is useful from the aspect of designing a data parallel algorithm: the parallelism in this subdivision process comes from simultaneous operations such as convex combinations of two consecutive control coefficients across large sets of data, rather than from multiple threads of control. Our solver takes advantage of this inherent parallelism in the subdivision process by exploiting massive parallel processors of the GPU.

Another issue is how to decide the order of multivariate function subdivision during tree traversal. A naive approach is to subdivide each function in the order of parameters one by one. However, eliminating a node of the subdivision tree at higher levels would provide much improvement in performance by reducing the number of function subdivision. Thus, we could employ the following alternative ways to decide the subdivision order at every level of the subdivision tree:

- we choose the parameter axis that has the maximum range: direction $= \max_j(u_j^{\max} - u_j^{\min})$, where $u_j^{\max}$ and $u_j^{\min}$ are the upper and lower bounds of the current domain along the u_j-direction,
- we choose the parameter axis along which the control coefficients have the maximum difference: direction $= \max_j(P_j^{\max} - P_j^{\min})$, where $P_j^{\max}$ and $P_j^{\min}$ are the maximum and minimum values of the control coefficients along the u_j-direction.

Considering the overhead of determining the subdivision direction, the former is easier to compute since no additional calculations are required for computing the directional min/max of the control coefficients. However, in terms of culling efficiency, the latter gives better performance. In our experiments, we employ the latter scheme for improvements in performance.

We now consider the computational complexity of the proposed parallel algorithm. Assume that k processors are available and those processors are running in synchrony without additional communication costs. Since we perform the univariate curve subdivision for Π_j different isoparametric curves when subdividing each function along the u_j-direction, the time complexity of the single-core algorithm is $O(\Pi_j n_j^2)$. Assuming $n_j = n, \forall j = 1, \ldots, m$, $\Pi_j = n^{m-1}$, which results in the asymptotic complexity of $O(n^{m+1})$. The complexity of a data parallel algorithm can be expressed in terms of two architecture-independent parameters: the *step* complexity and the *work* complexity. Our parallel subdivision algorithm requires n parallel steps, and the total number of operations used by the algorithm is the same as that of the single-core algorithm, namely, $O(n^{m+1})$. Thus, the time complexity of the parallel subdivision algorithm is $O(n^{m+1}/k + n)$. In practical applications, such as all examples shown in Section 6, we have $k \approx n^{m+1}$.

5.3. *CUDA implementation details*

The proposed parallel subdivision method is implemented using an NVIDIA CUDA for GPU programming (http://www.nvidia.com/object/duca_home.html). Our GPU kernel consists of two components: multivariate function subdivision and zero testing. *Algorithm* 3 describes the pseudo code of the GPU kernel.[1]

The GPU kernel is executed by the CPU mainThread at every level of the subdivision tree (Fig. 2). For each such execution, the mainThread transfers a set of *nodes* to the GPU as explained in Section 4.1. The GPU kernel then assigns a single block for each multivariate function $\mathcal{F}_i$ of each *node*: given n multivariate functions $\mathcal{F}_i$, $i = 1, \ldots, n$, n different blocks perform function subdivision in parallel for each *node*.

A set of threads in a block first subdivides the given function $\mathcal{F}_i$ in parallel based on the subdivision algorithm explained in Section 5.2: In *Algorithm* 3, step 2 through step 8 perform the subdivision algorithm. The number of threads in a block is equal to the number of control coefficients in $\mathcal{F}_i$. An index of each isoparametric curve of $\mathcal{F}_i$ is assigned to each thread according to the subdivision axis u_j (step 2). This index for each isoparametric curve depends only on the degrees of $\mathcal{F}_i$, which could be pre-computed in the initialization step. We store the pre-computed index set in the texture memory of the GPU for fast access in the subdivision stage. Through the subdivision stage along the given parameter axis, a block finally generates two sets of control coefficients for $\mathcal{F}_i^{left}$ and $\mathcal{F}_i^{right}$ (steps 7 and 8).

After subdividing $\mathcal{F}_i$ into two functions $\mathcal{F}_i^{left}$ and $\mathcal{F}_i^{right}$, the GPU kernel performs zero testing for each subdivided function: the kernel tests if their control coefficients have different signs (steps 10 through 20). We employ a parallel reduction technique for this purpose, which could be implemented quite efficiently using CUDA. Finally, the kernel sends the results of the zero testing step to the CPU mainThread (step 22). Since we keep all the control coefficient data on the GPU global memory during the tree traversal, the amount of data transmission between the CPU and the GPU is quite small: each block sends two bits to the CPU for both $\mathcal{F}_i^{left}$ and $\mathcal{F}_i^{right}$, where 0 means that there is no solution on the domain and 1 means that it potentially has solutions. The GPU global memory is reused to efficiently store the control coefficient data after traversing down a few levels of the subdivision tree.

6. Experimental results

In this section, we validate the proposed hybrid parallel constraint solver through several experiments: Hausdorff distance queries, contact point computations, surface–surface intersections, ray trap constructions, and bisector surface computations. All experiments are performed on a PC equipped with an Intel i5-750 Quad CPU 2.67 GHz with 4 GB of memory. Our GPU algorithm is implemented on an NVIDIA Geforce GTX580 graphics card, which supports 512 CUDA cores with 16 multi-processors.

For all experiments in this work, we compare the performance of our parallel solver with that of the previous CPU-based constraint solvers: the latest subdivision solver based on a tensor product representation [6] and the one based on expression trees [3]. For a fair comparison, we assume that

- the CPU solvers do not perform the numerical improvement step [1],
- the proposed parallel solver converts the input *B*-spline geometries to piecewise Bezier functions, while preserving their shape,
- the three solvers including the proposed parallel one compute the solutions in double precision.

Also, the same subdivision tolerance was used by the three solvers, and the resulting solution sets are verified to be the same with each other.

[1] The complete source code of our method is available at http://cgna.ssu.ac.kr/papers/gpu_solver/.

Algorithm 3 CUDA subdivision kernel

```
1: // subdivision stage
2: read $P^{[0]}_{threadID}$ from the GPU global memory
3: crvIdx $\Leftarrow$ index of the isoparametric curve;
4: for $k \in 1, \ldots, n$ do
5:     $P^{[k]}_{crvIdx} = \frac{b-c}{b-a} P^{[k-1]}_{crvIdx-1} + \frac{c-a}{b-a} P^{[k-1]}_{crvIdx}$
6: end for
7: $\mathcal{F}_i^{left}[crvIdx] = P_0^{[crvIdx]}$
8: $\mathcal{F}_i^{right}[crvIdx] = P^{[n]}_{crvIdx}$
9: // zero-testing stage
10: for $stride = (n+1) \gg 1$; $stride > 1$; $stride = (stride+1) \gg 1$ do
11:     $secondIdx = threadId + stride$
12:     if $secondIdx < stride$ then
13:         $HasSolution^{left}[threadId] +=$
14:         $(\mathcal{F}_i^{left}[threadId] * \mathcal{F}_i^{left}[secondIdx] < 0)$
15:         $+ HasSolution^{left}[secondIdx]$;
16:         $HasSolution^{right}[threadId] +=$
17:         $(\mathcal{F}_i^{right}[threadId] * \mathcal{F}_i^{right}[secondIdx] < 0)$
18:         $+ HasSolution^{right}[secondIdx]$;
19:     end if
20: end for
21: // write the results
22: send $HasSolution^{left}$ and $HasSolution^{right}$ to the CPU
```

Table 1
Timing data for computing the Hausdorff distance using three different constraint solvers (units: ms).

# of ctl. pts.	5 × 5	10 × 10	20 × 20	30 × 30	40 × 40
Tensor product	2639	2166	3914	9506	21,756
Expression tree	1672	1374	2095	3160	6415
Parallel solver	19	22	26	38	45

6.1. Hausdorff distance between two curves

The first experiment is to compute the Hausdorff distance between two planar curves. The purpose of this experiment is to show scalability of our parallel solver in terms of the input curves' complexity. The Hausdorff distance between two planar curves, $C_1(t)$ and $C_2(r)$, can be computed by solving the following system of three equations with three unknowns [3]:

$$\begin{aligned} &\langle C_1(t) - C_2(r), C_1(t) - C_2(r)\rangle = \langle C_1(t) - C_2(s), C_1(t) - C_2(s)\rangle, \\ &\langle C_1(t) - C_2(r), C_2'(r)\rangle = 0, \\ &\langle C_1(t) - C_2(s), C_2'(s)\rangle = 0. \end{aligned} \quad (1)$$

Note that the same curve C_2 is independently parameterized by two variables r and s. With the same numbers of equations and variables, the common zero-set of the above equations is a set of discrete points corresponding to the Hausdorff distance between $C_1(t)$ and $C_2(r)$. Fig. 5 shows an example of the Hausdorff distance computation.

In order to show the scalability of each solver with the complexity of the input curves, we computed the Hausdorff distance with varying control polygon sizes for the input curves. Table 1 shows the timing data, and their plots are shown in Fig. 6. The top row of Fig. 6 shows the timing data in log scale. Our parallel solver achieved more than two orders of magnitude improvement in performance given the complex input curves. At the bottom row of Fig. 6, an improvement of our solver in performance is shown relative to the other CPU-based solvers: as expected, the improvement grows rapidly with the complexity of the input curves. This result could be explained by the fact that massive parallel processors of the GPUs are suitable to complex geometric problems requiring large amount of geometric data to

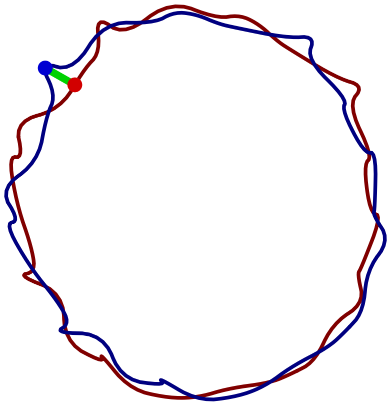


Fig. 5. An example of computing the Hausdorff distance between two planar curves.

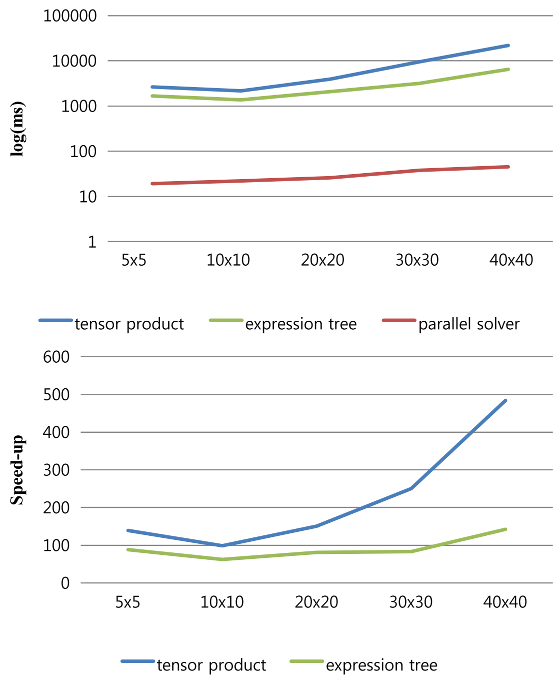


Fig. 6. A comparison of the three solvers in performance: (top row) the computation time in log scale and (bottom row) relative improvements of our parallel solver.

be processed. This trend is more apparent in comparison with the tensor product-based solver.

6.2. Contact points between two moving surfaces

The second experiment deals with the problem of computing contact points between two moving freeform surfaces, which is important in robotics, NC machining, and simulation. In this experiment, we demonstrate the efficiency of our parallel solver in constraints with a large number of variables: given two C^1 freeform parametric surfaces, $S_1(u_1, v_1)$ and $S_2(u_2, v_2)$, and a scale and translation transformation $T(t)$, the contact points between $T(t)[S_1(u_1, v_1)]$ and $S^2(u_2, v_2)$ satisfies the following five equations with five unknowns:

$$\begin{aligned}
&T(t)[x_1(u_1, v_1)] = x_2(u_2, v_2),\\
&T(t)[y_1(u_1, v_1)] = y_2(u_2, v_2),\\
&T(t)[z_1(u_1, v_1)] = z_2(u_2, v_2),\\
&\left\langle \frac{\partial T(t)[S_1(u_1, v_1)]}{\partial u_1}, \frac{\partial S_2(u_2, v_2)}{\partial u_2} \times \frac{\partial S_2(u_2, v_2)}{\partial v_2} \right\rangle = 0,\\
&\left\langle \frac{\partial T(t)[S_1(u_1, v_1)]}{\partial v_1}, \frac{\partial S_2(u_2, v_2)}{\partial u_2} \times \frac{\partial S_2(u_2, v_2)}{\partial v_2} \right\rangle = 0, \qquad (2)
\end{aligned}$$

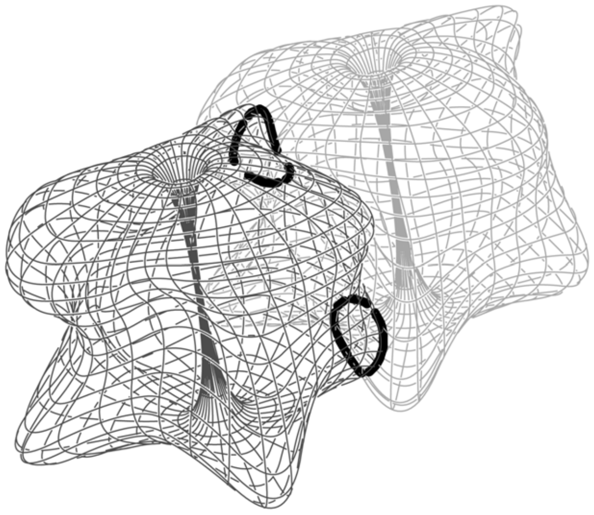


Fig. 7. An example of computing surface–surface intersection.

where $T(t)[\cdot]$ denotes the transformation operator.

In this experiment, the two input surfaces are biquadratic B-spline surfaces with a control mesh size of 14×7. The transformation curve $T(t)$ is a cubic B-spline curve with 29 control points. The timing data for this example are presented in Table 2. The proposed parallel solver computes the contact points in 127 ms, while it took 48,750 ms and 9923 ms for the same example using the two CPU solvers. Our parallel solver gives about two orders of magnitude improvements in performance relative to the better CPU-based solver.

6.3. Surface–surface intersection

The next experiment is one of the most classic geometric problems: surface–surface intersections. Given two freeform parametric surfaces $S_i(u_i, v_i) = (x_i(u_i, v_i), y_i(u_i, v_i), z_i(u_i, v_i)),\ i = 1, 2$, the formulation for this problem is well known, which can be represented by the following three equations in four unknowns:

$$\begin{aligned}
&x_1(u_1, v_1) = x_2(u_2, v_2),\\
&y_1(u_1, v_1) = y_2(u_2, v_2),\\
&z_1(u_1, v_1) = z_2(u_2, v_2). \qquad (3)
\end{aligned}$$

The purpose of this experiment is to show the efficacy of the proposed parallel solver in finding solutions of dimension one. If the dimension of the solution set is greater than zero, then we can fit a multivariate surface (a univariate curve in this specific example) to the set of discrete solution points. This fitting process produces an approximation to the solution set.

Since the current version of the expression tree-based solver does not support a system of constraints with higher-dimensional solution space, we measured the computation times only for the other two solvers. Fig. 7 and Table 2 show an example of computing the intersection curves and timing data, respectively. We performed this experiment with two sets of freeform surfaces: similarly to the results of the first experiment, the example with complex input surface shapes gives more improvements in performance.

6.4. Ray traps

The next example deals with the problem of *ray traps* for planar curves in R^2: given k planar parametric curves, $C_i(t_i),\ i =$

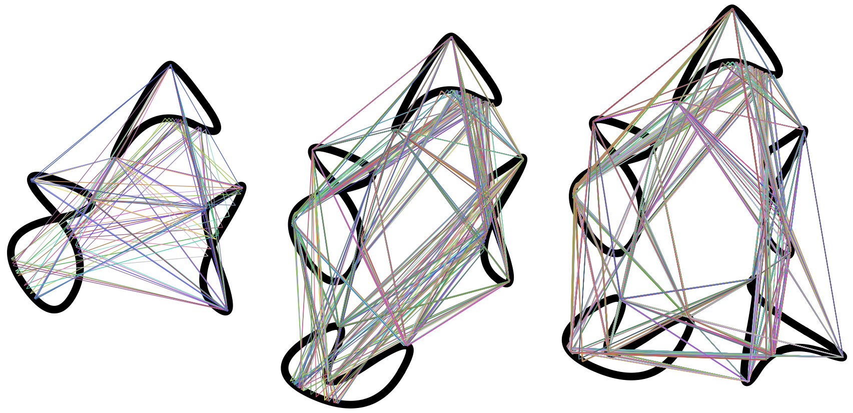


Fig. 8. Examples of all ray traps between k planar quadratic B-spline curves.($k = 3, 4, 5$.)

Table 2
Timing data for geometric examples using three different constraint solvers (units: ms).

Examples	Contact points	SSI		Bisectors
# of ctl. pts.	$14 \times 7 \times 14 \times 7$	$14 \times 7 \times 14 \times 7$	$26 \times 10 \times 26 \times 10$	$3 \times 3 \times 2 \times 2$
Tensor product	48,750	252	843	1170
Expression tree	9,923	NA	NA	NA
Parallel solver	127	46	66	50

Table 3
Timing data for ray traps (units: ms).

# of curves	3 curves	4 curves	5 curves
Tensor product	171	948	14,529
Expression tree	252	2011	23,303
Parallel solver	12	117	1775

$0, \dots, k-1$, in the xy-plane, a ray trap of length k is a set of k points $P_i = C_i(t_i)$ such that a bounding ray off P_i toward $P_{(i+1) \bmod k}$ will be reflected at $P_{(i+1) \bmod k}$ toward $P_{(i+2) \bmod k}$. Let N_i denote a normal field of curve C_i. Then, the ray trap problem could be formulated using the following constraints in terms of the k parametric curves [3]:

$$\frac{\langle C_{(i-1) \bmod k}(t_{(i-1) \bmod k}) - C_i(t_i), N_i\rangle}{\|C_{(i-1) \bmod k}(t_{(i-1) \bmod k}) - C_i(t_i)\|} = \frac{\langle C_{(i+1) \bmod k}(t_{(i+1) \bmod k}) - C_i(t_i), N_i\rangle}{\|C_{(i+1) \bmod k}(t_{(i+1) \bmod k}) - C_i(t_i)\|}. \quad (4)$$

For k curves, one needs to solve k equations of the form shown in Eq. (4) with k unknowns. In order to explicitly show the scalability of our solver, we consider a set of examples with increasing k (the number of curves). Fig. 8 shows examples for the cases $k = 3, 4, 5$. The timing data for these examples are shown in Table 3.

The proposed parallel solver achieved about an order of magnitude improvement in performance compared to the tensor product-based solver. Unlike the other examples in this work, the expression tree-based solver took more time than the one based on a tensor product representation since the constraints require greater depths in expression trees. Also, the relative improvements of our parallel solver to the other two solvers are almost constant in terms of the number of the input curves. This is due to the fact that each constraint of the form in Eq. (4) depends only on the three consecutive input curves even though the example takes more than three input curves, which results in a constraint of constant size with varying numbers of the input curves.

6.5. Surface bisectors

Our last geometric example is to compute the bisectors between two freeform surfaces. Adopting the formulation of this surface–surface bisector problem from [1], we can reduce this problem into that of finding the common zero-set of the following two functions with four unknowns:

$$F_1(u, v, s, t) = \left\langle \delta(u, v, s, t), \frac{\partial S_2(s, t)}{\partial s} \right\rangle = 0,$$

$$F_2(u, v, s, t) = \left\langle \delta(u, v, s, t), \frac{\partial S_2(s, t)}{\partial t} \right\rangle = 0, \quad (5)$$

where

$$\begin{aligned} &\delta(u, v, s, t) \\ &\quad = -2(S_1(u, v) - S_2(s, t))\langle n_1(u, v), S_1(u, v) - S_2(s, t)\rangle \\ &\qquad + n_1(u, v)\langle S_1(u, v) - S_2(s, t), S_1(u, v) - S_2(s, t)\rangle. \end{aligned}$$

Having two equations in four unknowns, the common zero-set of this system of equations produces a set of discrete points on the bisector surfaces. Fig. 9 shows an example of bisector surfaces between a plane and a biquadratic surface. The set of discrete solution points can be considered as sampled points on the bisector surfaces. We therefore fit a bivariate surface to the discrete solution points. Either the parameterization of $S_1(u, v)$ or that of $S_2(s, t)$ can be employed to define the parameterization of the sampled points. The timing data for this experiment is also shown in Table 2: again, the improvements of our solver in performance is substantial.

6.6. Discussions

In this section, we analyze the performance of our parallel solver in terms of both the performance gain due to the hybrid framework and the load balancing between the CPU and the GPU. We then present the limitations of our approach.

Hybrid framework: Our parallel solver exploits the multiple CPU cores as well as the multi-processors of the GPU. This hybrid approach allows both the CPU and the GPU work simultaneously, which results in better performance than exploiting a single-core CPU. Indeed, this performance gain could be achieved due to the proposed task management scheme: for traversal of the subdivision tree, a single mainThread deals with communication between the CPU and the GPU, while a set of workThreads performs dynamic node generation independently. The mainThread and the

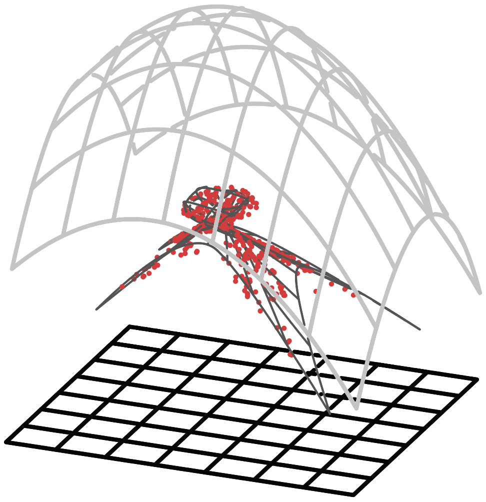


Fig. 9. An example of computing the bisectors between two freeform surfaces.

GPU work in a serial manner in the main loop of our solver, which is, however, independent of the tasks done by the workThreads. Therefore, the parallelism of the proposed hybrid framework comes from the simultaneous execution of both the GPU-based tasks and the tasks of the workThreads.

In order to analyze this performance gain quantitatively, we measured the execution times for each of the three components separately: mainThread (t_{main}), workThread (t_{work}), and GPUs (t_{GPU}). Fig. 10 summarizes timing data for each geometric application shown in Section 6. The results are similar for all examples. We therefore use the example of computing the Hausdorff distance for explanation. The total execution time t_{total} is 47.772 ms, and it is equal to the sum of t_{main} and t_{GPU}, each of which is 18.338 ms and 29.434 ms, respectively. t_{main} was mainly used for data transmission between the GPU and the CPU. We also measured actual working times t^i_{work} for each workThread, which excludes waiting time. In our experiment, we used three workThreads, and the total working time is $\Sigma_i t^i_{\text{work}} = 69.809$ ms.

When using a single-core CPU, the above three tasks should be done in serial, which results in the total execution time for the same Hausdorff distance computation, $t_{\text{total}} = t_{\text{main}} + \Sigma_i t^i_{\text{work}} + t_{\text{GPU}} = 117.581$ ms. Based on the decomposition scheme, however, a set of workThreads can work independently of the other tasks, which provides the total execution time of 47.772 ms. Thus, the performance gain is about 250% in this example.

Our hybrid framework requires data transmission between the CPU and the GPU: for each iteration in the main loop of our algorithm, the mainThread sends a set of *node* indices to the GPU, while the GPU transfers the results of multivariate function subdivision back to the CPU. Since this data transmission is slow relative to local memory access, it is desired to reduce the transmission data size.

For this purpose, our method stores all the geometry information of multivariate functions on the GPU and the mainThread deals with only the parameter domain on each of which function is defined. Furthermore, the GPU transfers a single bit to the CPU for each multivariate function F_i, which determines whether F_i may have solutions. Due to the reduced data size, the data transmission between the CPU and the GPU took relatively small portions of the total execution time: in Fig. 10, 15.4% of the total execution time were spent for the data transmission on average.

Load balancing: In our problem setting, load balancing between the CPU and the GPU is important in achieving high performance. Distribution of the workloads varies with the level of subdivision trees during traversal: at higher levels, the number of function subdivision is quite small relative to that of the parallel processors

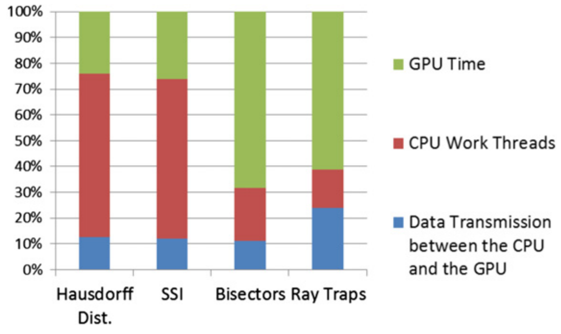


Fig. 10. Split-up of timings.

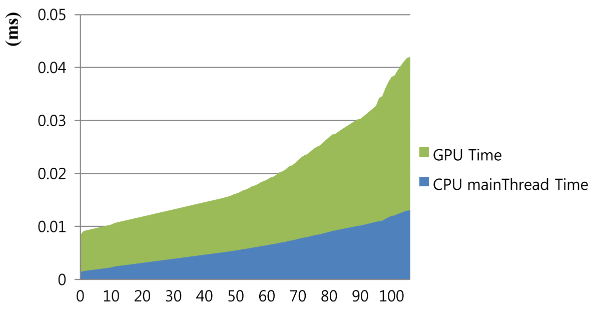


Fig. 11. Time spent on the GPU and the mainThread in the CPU.

of the GPU. While, at some level of the subdivision tree, the GPU multicores are fully exploited, which in turn increases the workloads of the CPU (specifically, the workloads of the workThreads).

We measured execution times spent on the CPU and the GPU separately using the example of computing the Hausdorff distance. Fig. 11 shows the timing data at every iteration in the main loop of our solver. In Fig. 11, one can see that the execution time increases with the iteration number, and also the portion of the GPU increases more rapidly than that of the CPU at lower levels of the subdivision tree (i.e., with the large iteration number). Since the time spent on the workThreads in the CPU are overlapped with the GPU-based function subdivision, we could not exactly measure load balancing between the GPU and the workThreads. However, each workThread spent $69.809/3 = 23.269$ ms on average for this example. Thus, the whole workloads were distributed approximately to the CPU and the GPU with the ratio of 41.607 ms (mainThread+workThread): 29.434 ms (GPUs) = 58.5%: 42.5%, which is closed to the ideal load balancing.

Limitations: Our algorithm has certain limitations. First, due to the limited amount of the shared memory in the GPU, the performance of our solver could be degraded when the size of the control mesh for a constraint exceeds 48 kB: our GPU kernel reads control coefficients of each constraint into the shared memory, and thus if the size of the control mesh exceeds 48 kB, then a special algorithm is required for handling a constraint using multiple kernels. Second, the current graphics card has the maximum number (1024) of threads in a block, which also limits the size of the control mesh in a geometric application. Since the current version of our solver is based on a tensor product representation, this memory issue may become problematic in applications with large number of variables: given n number of variables in a constraint and k number of coefficients in each variable, the size of the tensor product constraint is k^n. Thus, in the current graphics card, k^n should be less than 1024, which was satisfied for all examples of our experiments in this work. This problem could be resolved by using expression trees for representing a constraint [3], which is one possible future work.

7. Conclusions

In this paper, we have presented a hybrid parallel algorithm for solving systems of multivariate polynomial functions by exploiting both the CPU and the GPU multicore architectures. Our approach is based on the decomposition of the subdivision-based constraint solving technique into two independent tasks, traversal of subdivision trees and multivariate function subdivision, each of which is more suitable to the CPU and the GPU, respectively. For CPU-based tree traversal, we proposed a task management scheme using multiple threads, which is lock-free and thus highly scalable in terms of both the number of variables and the complexity of input geometries. For GPU-based polynomial subdivision, our method exploits the inherent parallelism in the polynomial Bezier subdivision. We demonstrated the efficacy and scalability of the proposed parallel solver through several geometric examples, in which our solver achieved up to two orders of magnitude improvement in performance compared to using the latest subdivision-based constraint solvers.

There are several issues for future work. First, we would like to extend our current hybrid parallel solver to support *B*-spline representation for multivariate constraints. We would also like to extend the current solver to find one- and two-manifold solution space, which requires an efficient tracing algorithm in the higher-dimensional parameter space. Implementing a numerical improvement algorithm [1] could also be a future work.

Acknowledgment

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (2011-0002790).

References

[1] Elber G, Kim MS. Geometric constraint solver using multivariate rational spline functions. In: Proc. of international conference on shape modeling and applications. MIT; 2005. p. 216–25.
[2] Mourrain B, Pavone JP. Subdivision methods for solving polynomial equations. Journal of Symbolic Computation 2009;44(3):292–306.
[3] Elber G, Grandine T. An efficient solution to systems of multivariate polynomial using expression trees. IEEE Transactions on Visualization and Computer Graphics 2009;15(4):596–604.
[4] Reuter M, Mikkelsen TS, Sherbrooke EC, Maekawa T, Patrikalakis NM. Solving nonlinear polynomial systems in the barycentric Bernstein basis. The Visual Computer 2008;24(3):187–200.
[5] Lane J, Riesenfeld R. Bounds on a polynomial. BIT 1981;21:112–7.
[6] Hanniel I, Elber G. Subdivision termination criteria in subdivision multivariate solvers using dual hyper-planes representations. Computer-Aided Design 2007;39(5):369–78.
[7] Bartoň M, Jüttler B. Computing roots of polynomials by quadratic clipping. Computer Aided Geometric Design 2007;24(3):125–41.
[8] Nishita T, Sederberg TW, Kakimoto M. Ray tracing trimmed rational surface patches. Computer Graphics 1990;24:337–45.
[9] Sherbrooke EC, Patrikalakis NM. Computation of the solutions of nonlinear polynomial systems. Computer Aided geometric Design 1993;10:379–405.
[10] Piegl L, Tiller W. The NURBS book. Springer; 1997.
[11] Rockwood A. Constant component intersections of parametrically defined curves. Technical report. Silicon Graphics Computer Systems. 1989.
[12] Schneider PJ. A bezier curve-based root-finder. In: Graphics gems. San Diego: Academic Press, Inc.; 1990. p. 408–15.
[13] Sederberg TW, Meyers R. Loop detection in surface patch intersections. Computer Aided Design 1988;5(2):161–71.
[14] Sederberg TW, Zundel AK. Pyramids that bound surface patches. Graphical Models and Image Processing 1996;58(1):75–81.
[15] Kim D, Heo J, Huh J, Kim J, Yoon S. HPCCD: hybrid parallel continuous collision detection. Computer Graphics Forum (Pacific Graphics) 2009;28(7): 1, 8, 9.
[16] Lauterbach Christian, Mo Q, Manocha Dinesh. gProximity: hierarchical GPU-based operations for collision and distance queries. Computer Graphics Forum 2010;29(2):419–28.
[17] Herlihy M. Obstruction-free synchronization: double-ended queues as an example. In: Int. conf. on distributed computing systems. 2003. p. 522–9. 2.
[18] Cohen E, Riesenfeld RF, Elber G. Geometric modeling with splines: an introduction. A.K. Peters, Ltd.; 2001.