ELSEVIER

Computer Aided Geometric Design 11 (1994) 39–69

COMPUTER AIDED GEOMETRIC DESIGN

Surface intersection using parallelism

Long Chyr Chang [a,*,†], Wolfgang W. Bein [a,‡,§,¶], Edward Angel [a,‖]

[a] *Department of Computer Science, The University of New Mexico, Albuquerque, NM 87131, USA*

Received October 1991; revised September 1992

Abstract

The support of Boolean set operations in free-form solid modeling systems requires the repeated intersection of parametric surfaces. Present approaches to this problem are sequential and must make trade-offs between accuracy, robustness and efficiency. In this paper, we investigate a parallel approach to the surface intersection problem that shows, both theoretically and empirically, that with parallelism we can achieve both speed and precision simultaneously. We first develop a theoretical foundation for a subdivision method and derive complexity bounds. We show that the basic algorithm can be improved by parallelism. We then design two tolerance-based parallel subdivision algorithms, a macro-subdivision algorithm designed for MIMD shared memory machines and a lookahead-subdivision algorithm for pipelined MIMD machines. Empirical results on the Sequent Balance 21000, the Alliant FX/8, and the Cray-2 verify that significant speed-up is achievable.

Key words: Surface intersections; Parallel algorithms; Free-form solid modeling systems; Algorithm complexity

1. Introduction

Solid modeling systems provide an environment that allows designers to edit, visualize, and analyze complex objects bounded by sculptured (free-form) surfaces for prototyping and testing on computer screens before they are manufactured. The success of such systems has been recognized as a key

*Present address: Superconducting Super Collider Laboratory, 2550 Beckleymeade Avenue, Dallas, TX 75237, USA.

†Sequent–Alliant time grant Argonne Advanced Computing Research Facility.

‡Present address: American Airlines Decision Technologies, P.O. Box 619616, MD 4462 HDQ, Dallas/Fort Worth Airport, TX 75261-9616, USA.

§Supported by Sandia National Laboratories grant, SURP No. 05-9858 task9.

¶CRAY time grant from the National Center for Supercomputing Applications

‖Corresponding author. Email: angel@cs.unm.edu.

SSDI 0167-8396(93)E0013-4

Table 1
Complexity of surface/surface intersection

Surface pair	Resultant curve
Two planes	Straight line
Plane and quadric	Conic
Two quadrics	Quartic space curve
Plane and rational bicubic	Degree-18 plane algebraic curve (190 terms)
Two rational bicubics	Degree-324 space curve (17 million terms)

element in developing a fully automated computer-aided design/manufacturing (CAD/CAM) system for practical applications (Wolfe, 1987). One of the major research problems in designing improved solid modeling systems is to extend the geometric domain from quadric to higher degree or *free-form* surfaces.

A major computational problem in designing such a free-form system is the difficulty of the surface/surface intersection (SSI) problem. SSI is the fundamental computation for supporting Boolean set operations such as intersection, union, and difference, that must be performed in the editing and modeling of complex objects constructed from simple primitives. Fundamental to the SSI problem are the problems of computation and representation of the intersection curves. Wilson (1988) gives a table (see Table 1) illustrating how the complexity of the SSI problem grows as the degree of the mathematical surfaces increases.

Several approaches to solve the SSI problem have been proposed (Casale, 1985; Crocker, 1987; Farouki, 1987) Regardless of which approach is used, a good algorithm must satisfy three properties: accuracy, robustness, and efficiency (Pratt, 1986; Hoffman, 1989). The computed intersection curve must be within a specified tolerance to the true intersection curve. The algorithm must not be subject to failure and must find all the portions of an intersection curve which may have many disjoint branches. For many applications, algorithms must be efficient enough so that interaction between a CAD user and a CAD system can be achieved. These requirements present a serious dilemma to the algorithm designer. An algorithm that is robust and accurate may be very slow, while an efficient algorithm might fail in some cases. In spite of much effort, no present algorithm satisfies all three of the above properties. To make practical use of one of the present sequential algorithms, implementors must make significant trade-offs between desirable properties of the algorithm.

As commercial parallel computers become cheaper and more available, a natural approach to SSI that takes advantage of parallel architectures appears promising. However, before parallel approaches can be used effectively in practice, the following issues must be addressed:

- How can the various parallel architectures be used to improve the trade-off dilemma for SSI algorithms?
- How much improvement can a parallel algorithm make over its sequential counterpart?
- Is there an optimal number of processors which achieves the best cost-effective performance ratio?

We will show, both theoretically and empirically, that parallelism indeed presents an approach that can surmount the difficulties in achieving speed and precision simultaneously. The paper is organized as follows: Section 2 presents a framework of algorithm complexity based on derivative bounds of surfaces and discusses how parallelism can be used to solve the speed and precision dilemma theoretically. We develop a framework for algorithm complexity based on the curvature information of C^2 surfaces. We show that the complexity of subdivision methods can be expressed in terms of the curvature of the two surfaces and an input tolerance. This complexity analysis indicates precisely how a parallel solution can resolve the trade-off dilemma between speed and precision of a SSI algorithm.

Section 3 discusses the inherent problems involved in parallelizing subdivision and describes two parallel algorithms for surface intersection. We develop both parallel algorithms and appropriate data structures for SSI using subdivision or curve-tracing methods. We will show empirically how much improvement a parallel algorithm can make over its sequential counterpart. We have chosen shared memory MIMD (multiple instructions multiple data streams) machines as the primary hardware platforms for implementing parallel algorithms because of their wide availability and ease of programming. The proposed algorithms have been tailored to three different MIMD and vector pipeline[1] parallel machines (Sequent Balance 21000, Alliant FX/8, and Cray-2). Section 4 illustrates the benchmarks which were obtained from these implementations. Empirical results are shown and analyzed.

2. Complexity analysis

2.1. Background

In what follows, we derive formal properties of the tolerance based SSI algorithms. In particular, we will consider surfaces which are given as parametric images P over a rectangular domain $\Omega \in E^2$. $P : \Omega \to E^3$ is assumed to be a C^2 function. Given two such surfaces S_1 and S_2, and a tolerance ε, the problem is to compute the intersection curves $\gamma_1 \subseteq \Omega_1$ and $\gamma_2 \subseteq \Omega_2$ such that

$$\sup_{(u,v)\in\gamma_1} \inf_{(s,t)\in\gamma_2} | P_1(u,v) - P_2(s,t) | < \varepsilon.$$

Our algorithm complexity analysis is based on the use of derivative bounds for parametric surfaces. Several studies (Filip, 1986; Herzen 189, 1990; Kala, 1989, Lane, 1979) have demonstrated that the derivative bound of a surface can be used in computing piecewise linear approximation within a tolerance,

[1] A vector computer can carry out the same operation simultaneously on a vector of data. For example, a vector machine can often add two vectors or multiply a vector by a constant in one operation. Pipeline machines have two or more successive processors connected sequentially. Multiple operations can be carried out concurrently on a stream of data, similar to the way an assembly line functions.

constructing a hierarchy of bounding volumes for a surface, and computing surface intersection within a specified tolerance. The primary advantage of using a derivative approach is that the algorithms developed for the above applications are very generic since only C^1 or C^2 continuity is required. In addition, a hierarchy of bounding volumes of a surface can be computed very easily. Previous work using bounds on the derivatives of C^1 and C^2 surfaces can be classified as the Lipschitz-condition approach and the curvature approach respectively. Both approaches can be used for the above applications, the primary difference being the size of the bounding volumes they generate. For the detailed comparison of both approaches, the reader is referred to (Chang, 1991).

We will use the following theorem due to (Filip, 1986) as the basis for our work.

Theorem 2.1 (Filip). *Let $T \subset \mathbb{R}^2$ be a right triangle with vertices (A, B, C) of the form $B = A + (l_1, 0)$ and $C = A + (0, l_2)$. Let $f : T \to \mathbb{R}^3$ be any C^2 surface, and $l(u, v)$ be any linearly parameterized triangle with $l(A) = f(A)$, $l(B) = f(B)$ and $l(C) = f(C)$ (see Fig. 1(A), (B)). Then*

$$\sup_{(u,v)\in T} \|f(u,v) - l(u,v)\| \leqslant \tfrac{1}{8}(l_1^2 M_1 + 2l_1 l_2 M_2 + l_2^2 M_3) \tag{1}$$

where

$$M_1 = \sup_{(u,v)\in T} \left\| \frac{\partial^2 f(u,v)}{\partial u^2} \right\|, \qquad M_2 = \sup_{(u,v)\in T} \left\| \frac{\partial^2 f(u,v)}{\partial u \partial v} \right\|,$$

$$M_3 = \sup_{(u,v)\in T} \left\| \frac{\partial^2 f(u,v)}{\partial v^2} \right\|.$$

Piecewise linear surface-approximation

By the above theorem, we can compute a piecewise linear approximation function $l : [0,1] \times [0,1] \to \mathbb{R}^3$, for a given C^2 surface $f : [0,1] \times [0,1] \to \mathbb{R}^3$ and an arbitrary tolerance ε such that $\sup \|f(u,v) - l(u,v)\| \leqslant \varepsilon$.

If we divide the domain $[0,1] \times [0,1]$ uniformly into mn rectangular subdomains, we can form $2mn$ right triangle subdomains (see Fig. 1(C)). The desired approximation can be computed within a given tolerance by evaluating f at an $(m+1) \times (n+1)$ grid of points and forming $2mn$ triangular patches. Letting $l_1 = 1/n$ and $l_2 = 1/m$, we can compute n and m by the equation

$$\frac{1}{8}\left(\frac{1}{n^2} M_1 + \frac{2}{nm} M_2 + \frac{1}{m^2} M_3\right) = \varepsilon. \tag{2}$$

Without loss of generality, we can assume that n and m are equal. This approach will be called the uniform domain decomposition (UDD) approach.

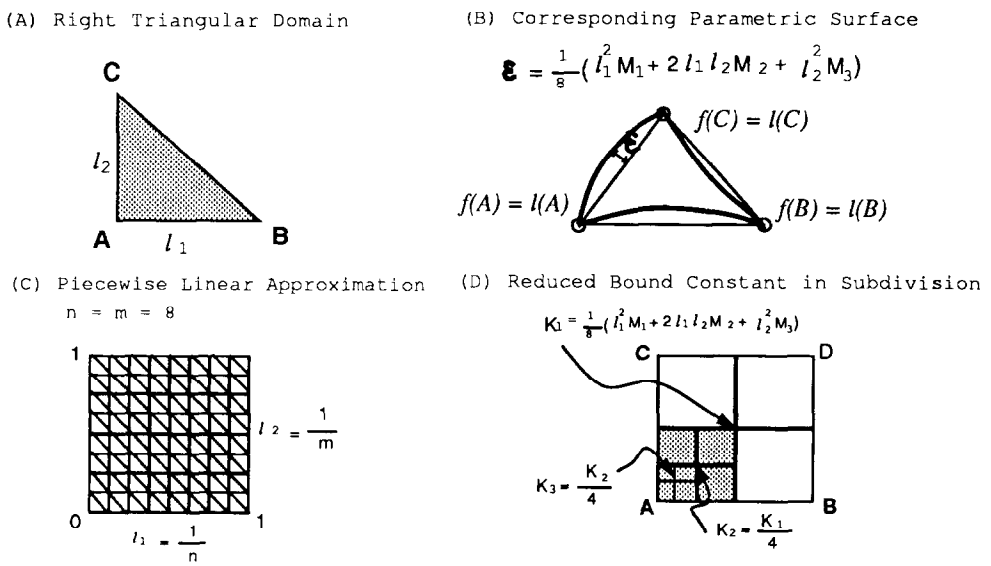


Fig. 1. Derivative bounds and piecewise linear approximation.

Thus, we have

$$2mn = \frac{M_1 + 2M_2 + M_3}{4\varepsilon}. \tag{3}$$

In particular, suppose we let $M = \max\{M_1, M_2, M_3\}$, that is, M is the largest second derivative of the surface mapping function in the u, v, and uv direction (referred to as the curvature bound of surface later). Then

$$2mn \leqslant M/\varepsilon. \tag{4}$$

Bounding box and scale factor

Computing a bounding box for a surface patch using bounds on its derivatives is straight forward. We first compute the Min–Max bounding box $\langle P_{\min}, P_{\max}\rangle$ of four corners of the surface patch. The bounding box of the whole surface patch is then computed as $\langle P_{\min} - (K,K,K), P_{\max} + (K,K,K)\rangle$ where

$$K = \tfrac{1}{8}(l_1^2 M_1 + 2 l_1 l_2 M_2 + l_2^2 M_3).$$

There are two ways to compute a hierarchy of bounding boxes for a surface in the course of a subdivision process. First, one can compute the derivative bounds for each sub-domain in a hierarchy. This will give a tighter bounding box for each sub-surface. For some surfaces, such as Bernstein–Bézier surfaces, this can be done easily, while for most other surfaces, considerable effort is required. Alternatively, one can use the derivative bound for the entire parametric surface domain as a global bound K. For each subpatch split from a surface patch in a uniform subdivision process, its K is just one fourth of its parents since l_1 and l_2 are reduced by half (see the above equation and

Fig. 1(D)). For a surface patch in depth d in a hierarchy of subdivisions, its K is computed as $K/4^d$. The bounding box becomes smaller as the subdivision becomes deeper. This feature makes Filip's subdivision approach attractive. The constant scale factor is called the bounding volume scale factor. In this case, it is $\frac{1}{4}$.

2.2. Complexity of surface intersections

The approximate intersection of two parametric surfaces can be computed directly from their piecewise linear approximations. This approach requires evaluating surface grids for each surface, and then repeatedly computing the intersection of two triangles (ITT).

Proposition 2.2. *Assume two C^2 parametric surfaces $f_1, f_2 : [0,1]\times[0,1] \to \mathbb{R}^3$, are piecewise linearly approximated with n_1 and n_2 triangle surface patches within a tolerance $\varepsilon/2$, respectively. Then, at most $n_1 n_2$ ITT tasks are needed to compute the intersection curve within a tolerance ε.*

Proof. (1) Obviously, the number of ITTs is at most $n_1 n_2$.

(2) Let l_1 and l_2 be piecewise linear approximations of surfaces 1 and 2 respectively. By definition, we have

$$\|l_1(u,v) - f_1(u,v)\| \leqslant \tfrac{1}{2}\varepsilon, \qquad \|l_2(s,t) - f_2(s,t)\| \leqslant \tfrac{1}{2}\varepsilon.$$

Therefore,

$$\begin{aligned}
&\|l_1(u,v) - l_2(s,t) - (f_1(u,v) - f_2(s,t))\| \\
&\quad = \|l_1(u,v) - f_1(u,v) - (l_2(s,t) - f_2(s,t))\| \\
&\quad \leqslant \|l_1(u,v) - f_1(u,v)\| + \|(l_2(s,t) - f_2(s,t))\| \\
&\quad \leqslant \tfrac{1}{2}\varepsilon + \tfrac{1}{2}\varepsilon = \varepsilon.
\end{aligned}$$

If $\|l_1(u,v) - l_2(s,t)\| = 0$, then we have

$$\|f_1(u,v) - f_2(s,t)\| \leqslant \varepsilon. \qquad \square$$

Proposition 2.3. *Given two C^2 parametric surfaces and a tolerance ε, the SSI problem can be approximated by the uniform domain decomposition (UDD) method, and the number of ITTs can be no more than $\lceil 4M_1M_2/\varepsilon^2 \rceil$, where M_1 and M_2 are the curvature bounds of f_1 and f_2, respectively.*

Proof. By Proposition 2.2 and Eq. (4), the number of ITTs is at most

$$\frac{M_1}{\varepsilon/2} \times \frac{M_2}{\varepsilon/2} = \frac{4M_1M_2}{\varepsilon^2}. \qquad \square$$

The above proposition shows the computational complexity is quadratically proportional to the inverse of the given tolerance. Therefore, if we increase the precision of the tolerance by a factor of a hundred, we can expect that the loss

of speed is a factor of ten thousand. Such loss cannot be made up by using coarse grain [2] parallel computers since there are too few processors on a coarse grain architecture to compensate for such a loss.

2.3. The subdivision method

The UDD approximation method can be viewed as an exhaustive approach that demonstrates the worst case complexity of the SSI algorithm. A better solution is to reduce the number of ITT computations. Subdivision is one such method. Subdivision methods use the divide and conquer paradigm to reduce recursively the original problem into a number of subproblems of the same type until each subproblem can be solved directly. The subdivision algorithm for intersecting two parametric surfaces successively subdivides regions of the surfaces that potentially intersect each other until the final regions are so nearly flat that their intersection can be computed by planar approximations. In general, the algorithm requires the following:

- a bounding box test mechanism to check quickly whether or not any pair of surfaces may intersect;
- a subdivision algorithm to subdivide the surfaces into a number of sub-surfaces;
- a criterion for termination;
- a triangle-triangle intersection solver.

Present algorithms using subdivision methods differ in how they handle the bounding box test, subdivision process, termination criterion, and also the type of surfaces they can deal with (Lane, 1991; Houghton, 1985; Filip, 1986; Barnhill, 1987). Some subdivide three-dimensional spatial surfaces as in (Lane, 1991). Others subdivide the two-dimensional parametric domain as in (Filip, 1986). Algorithms can be uniform or non-uniform. For surface modeling, it is more natural to deal with a two-dimensional domain since the intersection curve can be represented as a trimmed-curve in 2D parametric domain and a trimmed region in a parametric domain corresponds to a surface patch after Boolean set operations. We will consider using a uniform domain decomposition method instead of non-uniform adaptive method as a model for complexity analysis of SSI, since a non-uniform version can be viewed as an improved version of a uniform version. The complexity analysis of uniform versions can be used to explain the behavior of non-uniform subdivision version as well. In what follows, we give a general pseudo code of a SSI algorithm using a subdivision method to serve as a basis for the complexity analysis and for parallel implementations. In the code we use a "global queue" to store

[2] The grain size of a parallel machine refers to the complexity of the problems individual processors can handle. Coarse grain machines such as the CRAY-YMP tend to have a few (2–32) sophisticated processors, often with vector capability. Fine grain machines such as the CM-2 can have thousands of very simple processors while medium grain machines such as the MIMD distributed memory machines can have up to a 1024 processors, each as sophisticated as the processor in a scientific workstation.

intersection tasks that have to be processed further, as well as a quadtree structure, to keep the information on subdivisions to this point in the process.

```
Surface_Split(s)
/* pseudo code for a subdivision */
Surface s;
{
       Divide the parametric domain of s into 4 regions.
       For each region, evaluate the parametric mapping of its corners
       and compute the bounding box using curvature information.
       Update quadtree data structure.
}

SSI(S1,S2)
Surface S1,S2;
{
       Determine the bounding boxes of S1 and S2.
       If the bounding boxes do not intersect then return.
       else put pair of surfaces S1, S2 into a global queue.

       while(queue is not empty) {
             Take a pair of sub-surfaces s1, s2 from the global queue.
             Determine the "flatness" of s1 and s2
             If both surfaces s1 and s2 are flat then {
                 Compute planar triangle approximations (t11,t12),
                 (t21,t20) of s1 and s2 respectively;
                 compute the intersection lines of 4 pairs of
                 triangle/triangle intersections.
             }else{
                  if(both surfaces s1 and s2 are not flat ) {
                     Subdivide both surfaces./*call Surface_Split*/
                     Test bounding box intersection for each of 16 pairs
                     of subdivided sub-surfaces from s1,s2.
                     if the bounding boxes intersect then add them into
                     the global queue.
                  } else {
                        subdivide non-flat surface.
                        for each one of 4 sub-surfaces from non-flat
                        surface, test whether their bounding boxes
                        intersect with the flat surface or not;
                        if yes then add them into the global queue.
                }
           }
     }
}
```

2.3.1. *Complexity of the subdivision method*

In the subdivision process, the number of intersections of flat regions, bounding box tests and subdivisions depends on the spatial orientation of the two surfaces, their curvatures and a tolerance. Hence, it is quite natural to express the complexity in terms of curvature information and input tolerance. From different spatial orientations, we can obtain both upper and lower bound complexities for the subdivision algorithm. Fig. 2 illustrates the upper and lower bound situations. In the lower bound situation, the intersection curve passes

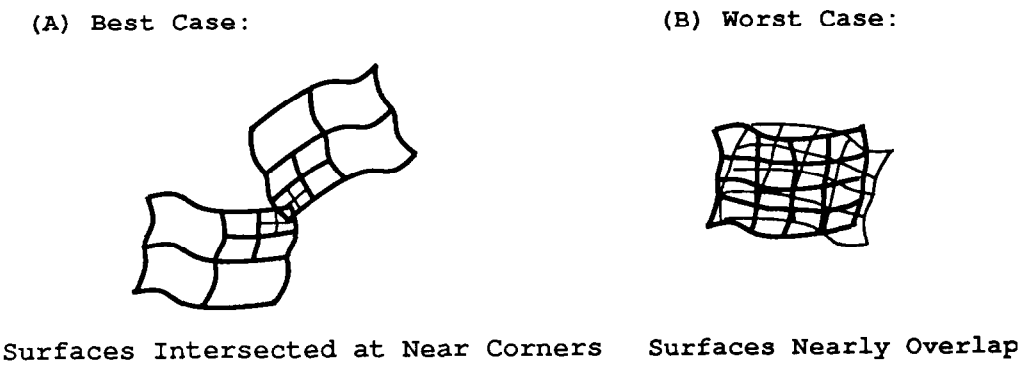


Fig. 2. Complexity relates to surface orientation.

only through one node in the lowest subdivision level of each surface, e.g. two surfaces intersect each other at a corner. In the upper bound situation, two surfaces intersect each other everywhere (e.g. two identical surfaces).

We can now give the complexity of the algorithm and analyze how parallelism can help in solving the trade-off problem between speed and precision.

Proposition 2.4. *Given two C^2 parametric surfaces P_1 and P_2, and a tolerance ε. Assuming the surfaces intersect, the computation time for SSI using the subdivision method and a quadtree data structure has a lower bound $\Omega(\log_4((M_1 + M_2)/\varepsilon))$, and an upper bound $O(((M_1 + M_2)/\varepsilon)^2)$, where M_1, M_2 are the curvature bounds of surface P_1 and P_2, respectively.*

The detailed proof is given in Appendix A.

Observation 1. In the design of a free-form solid, we use Boolean set operations (union, intersection and difference) to design the solid from primitives. In the best case (lower bound) situation, the parallel approach might work worse than a sequential approach. In the worst case (upper bound) situation, a parallel approach can improve the computation speed significantly. However, since the upper bound complexity is quadratically proportional to the inverse of tolerance, even in this case the improvement may not be of much significance as we may still not be able to achieve the desired tolerance in a reasonable amount of time.

Observation 2. We expect the upper and lower bound extremes only in the cases in which the two surfaces either touch minimally or nearly overlap (see Fig. 1). Therefore, we would like to have a bound to characterize the more common situations where there is partial overlap. From a practical point of view a more reasonable assumption is that in each subdivision process, about half of the regions of the surface-patch are eliminated by a bounding box test. In this case, the bound of $O((M_1 + M_2)/\varepsilon)$ is obtained. We regard this bound

as a "practical bound". An exact derivation of this third bound is given in Appendix A.

Observation 3. To support Boolean set operations, it is also necessary to solve the problem of intersecting one parametric surface and an algebraic surface within a tolerance. The intersection curve of the parametric surface $\{(x,y,z) = P(u,v) : (u,v) \in \Omega = [0,1] \times [0,1]\}$ with the algebraic surface $\{f(x,y,z) = 0 : (x,y,z) \in \mathbb{R}^3\}$ has an exact representation of the form $\{C(u,v) = 0 : (u,v) \in \Omega\}$. Therefore, if we can compute the plane curve $C(u,v) = 0$ within a tolerance (for convenience, we refer to such a curve as being defined by $|C(u,v)|^* \leqslant \varepsilon$), then the mapping of the plane curve into the parametric surface represents the computed 3D intersection curves $\{(x,y,z) = P(u,v) : |C(u,v)|^* \leqslant \varepsilon, (u,v) \in \Omega\}$ that fall into the area of the algebraic surfaces in $|f(x,y,z)| \leqslant \varepsilon$. In other words, the problem of tracing the intersection curves of a parametric surface and an algebraic surface within a given tolerance can be solved by computing a plane curve such that $|C(u,v)|^* \leqslant \varepsilon$.

On the other hand, the problem of computing $|C(u,v)|^* \leqslant \varepsilon$ can be viewed as the problem of computing the intersection curves of two surfaces $Z = C(X,Y)$ and $Z = 0$ within a tolerance. Both surfaces can be represented in parametric form. In particular, one surface is a plane. Therefore, the problem actually is a special case of the SSI tolerance problem and can be solved by using the above algorithms.

Proposition 2.5. *The problem of computing $|C(u,v)|^* \leqslant \varepsilon$ can be solved by the subdivision method with a quadtree data structure and has worst case complexity of $O(M/\varepsilon)$, where M is the curvature bound of the surface $Z = C(u,v)$.*

Proof. If subdivision with quadtree data structure is used, then only one surface is subdivided at any time. By the same arguments as in Proposition 2.4, the worst case complexity is $O(M/\varepsilon)$. □

Observation 4. From the above propositions and observations, we expect that in most situations the speed of the subdivision algorithm is proportional to the inverse of the tolerance (precision). Therefore, if we can design a parallel algorithm with linear speed-up, a parallel solution can indeed make up for the loss of speed caused by increasing tolerance linearly, and hence parallelism would be a feasible approach for solving the dilemma of speed versus precision.

3. Parallel subdivision algorithms

We will consider three strategies for parallelizing the underlying subdivision algorithm. The first strategy is to place each task generated by a subdivision process into a task queue. Each task includes any further subdivisions generated by the task, bounding box tests and triangle/triangle intersections. Tasks are

assigned to processors from the queue, a strategy that might appear to divide tasks evenly among all processors and to keep the processors as busy as possible. We call this strategy *macro-subdivision* because each processor gets a task from the queue and works on its task independently. This strategy is easy to implement but has some drawbacks. In practice, the tasks may not be evenly distributed to all the processors, especially when the spatial orientation of two surfaces is near a best case situation. Consider two surfaces that intersect each other at a corner. In this case, only one subdivision for each surface is needed at any level and only a single task is available to the processors at any time. Thus, most processors will sit idle and the parallel approach will be no better than a sequential one.

The second strategy is to reduce the computation time of a subdivision process, bounding box tests, triangle/triangle intersections, and queue operations by having all the processors working together to solve the operations required by a single subdivision. Because all processors are working together on a single task at a time, we call this strategy the *micro-subdivision* approach. This approach might have better performance than the macro-subdivision approach in the best case situation. But it also has some drawbacks. Since each subtask, e.g. an intersection or a bounding box test, has a different grain size, there will not be enough work for all the processors at all times and most available processors may sit idle. In addition, this strategy may have a high degree of synchronization overhead. Unless the overhead of the synchronization primitives in the parallel library can be reduced significantly, the performance of this approach may be worse than the macro-subdivision approach in most practical situations.

A better approach is to combine these two approaches into a *multi-level hybrid parallel* algorithm. This approach combines macro-subdivision and micro-subdivision in a doubly nested fashion. There are two types of multi-level parallel approaches, that depend on the underlying architecture of the parallel machine. For a shared memory MIMD vector pipeline machine the inner-level (micro-subdivision) of parallelism can be implemented by using vector processing techniques and the outer-level can be implemented as a macro-subdivision method. This approach is generally superior to both the micro-subdivision and macro-subdivision approaches although its implementation can be cumbersome. To take full advantage of vector pipeline features, a new technique which precomputes some portions of the subdivision process is developed. We call this approach a *lookahead-subdivision* approach, because it always looks ahead into the next subdivision level.

For middle grain MIMD shared memory machines without vector pipeline features, a multi-level hybrid parallel algorithm is implemented as follows: all the processors available are divided into several groups; each group has one master processor, the others are slave processors. The master processors are dynamically scheduled to get a task from the task queue using the macro-subdivision approach. The slave processors are devoted to those tasks which must be performed by their corresponding master processors (similar to the micro-subdivision approach). These tasks can be statically scheduled at the

beginning. The number of processors in a group depends on the tasks they need to solve. For example, in the subdivision process, the ideal number of a group is 4, or 5 (see pseudo code Surface_Split). This scheme allows a master processor to reduce its time in the subdivision process; thus indirectly reducing the possibility of traffic congestion caused by synchronization overhead.

However, the implementation of this approach is the most difficult of the above approaches, and its performance is also strongly affected by the overhead of synchronization primitives in the parallel library. Our implementation shows that the performance of this approach has no significant improvement as compared with macro-subdivision or micro-subdivision approaches in most practical cases and we will not discuss this approach further. Instead, we will focus attention on the macro-subdivision and lookahead-subdivision approaches.

3.1. Data structures and synchronization issues

For MIMD shared memory machines, there are two primary goals in designing a parallel algorithm. First, all tasks must be partitioned so that each processor has an equal share. Second, all processors must be kept as busy as possible while minimizing the synchronization overhead. To achieve these goals, appropriate data structures must be designed so that the performance of a parallel algorithm can be as close as possible to the theoretical linear speed up.

Theoretically, subdivision, which uses a divide-and-conquer paradigm, has a high degree of parallelism. All subdivided tasks can be put into the global task queue. Each processor takes a task from the global task queue and performs the same subdivision process until there are no tasks remaining and all processors are idle. However, there are some limitations which affect the performance of a parallel subdivision algorithm.

First, subdivisions are processed sequentially until each sub-patch is nearly flat. Since we use a quadtree to represent a subdivided surface, the complexity is dependent on the maximum depth of the quadtrees. Second, the global task queue is implemented as a shared resource. Therefore, care must be taken so that only one processor is allowed to manipulate the global task queue at a time. Third, manipulation of the global task queue must be implemented efficiently so as to minimize the overhead of task contention among the processors.

A parallel subdivision algorithm for SSI is more complex than a general parallel subdivision algorithm. Specifically, it must deal with the following situations:

- If a sub-patch of a surface has been subdivided by a process, then other processes should not also subdivide this surface-patch.
- New tasks are generated by combining any pairs of subdivided surface patches from each surface. Therefore, there is a potential deadlock situation since a process might have to wait for another process to finish its subdivided task.
- Choice of the degree of a tree data structure, corresponding to the degree (number) of sub-patches split from subdivision process, affects the performance of the algorithm.

Solutions for the first two situations relate to algorithm design and data structures, while solutions for the third situation depend on the grain size of the underlying MIMD shared memory machine. For fine grain machines the following should be considered: If surface 1 can be approximated with n_1 rectangular sub-patches and surface 2, n_2 rectangular sub-patches, then the total number of tasks is $n_1 \times n_2$. If this number is less than or equal to the number of processors, then each task can be performed by a processor independently. Therefore, in a fine grain size machine, subdivisions of high degree can be considered so that the number of subdivision levels can be minimized. For coarse grain size machines however, quadtrees turn out to be an appropriate data structure since such computers tends to have small number of processors.

3.2. Macro-subdivision parallel algorithm

In this section, a macro-subdivision parallel algorithm using a quadtree data structure is addressed. It is designed for coarse or middle grain size MIMD shared memory machines, and it is based on the principles discussed above. In addition, we use three data structures to minimize the overhead of synchronization:

1. A global task queue, implemented as a linked list, that holds tasks which are generated dynamically.
2. A lock synchronization primitive for each sub-patch that allows processors to subdivide different sub-patches simultaneously (and if different processors are subdividing the same surface patch then only one processor is allowed to do that task and the others must wait or do other work).
3. A local task queue for each processor to reduce memory contention in the global queue.

The sequential version of the subdivision algorithm using the domain decomposition method is shown in pseudo code SSI(). Specifically, each surface patch is divided into four sub-patches by subdividing its corresponding domain into four regions. Each region corresponds to a node in the quadtree (see Fig. 3). When a region is small enough that its surface patch mapping can be approximated by a planar rectangle formed by four corners of surface patch within a specified tolerance, it is not further subdivided (see Fig. 3). In this situation, it is a leaf (terminal) node. The revised parallel quadtree data structure has a lock flag for each node. It is designed for two purposes: to solve the lock synchronization problem noted above in the subdivision process and to solve similar situations in merging processes (connecting the edges in the leaf nodes of the quadtree data structure).

We can now discuss an outline of the algorithm. At the beginning, all data structures including the global task queue (GTQ) and the local task queue (LTQ(i)) for each processor $i = 1,\ldots,N$ are initialized. A root task, which is the intersection of two surfaces, is put into the GTQ. In a general step, processor i gets a task from LTQ(i) or the GTQ and performs its task independently. A processor i always gets a task from its LTQ(i) first; if it cannot find a task there, then it looks for a task from the GTQ. If it cannot get

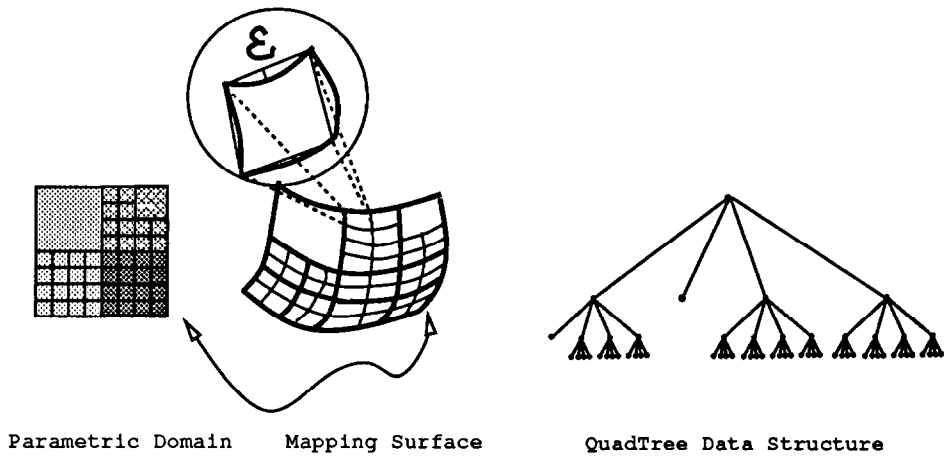


Fig. 3. Parametric surface, subdivision and quadtree.

a task, then it must wait and becomes idle. New tasks generated by processor i are put into LTQ(i) unless it notices that the GTQ is empty and there is a hungry processor waiting there. In this situation, new tasks are put into the GTQ. If a process is in a subdivision step, it must check if its surface patch is being subdivided or not. If not, it can enter and close the door so that no other processors can subdivide it. Otherwise it must wait or do something else. When a processor finishes, it reopens the door and sets a split flag so that other processors can notice it without subdividing again. In this way, the overhead of synchronization and contention of tasks in the GTQ is kept small. Fig. 4 illustrates the data structures used in macro-subdivision method. A detailed implementation is discussed in (Chang, 1990).

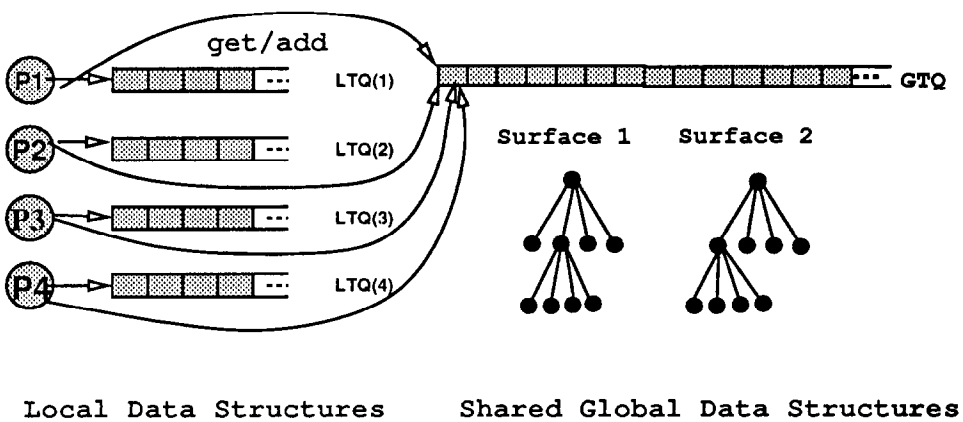
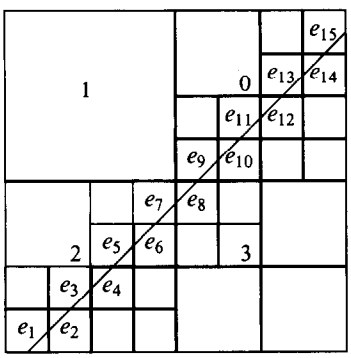
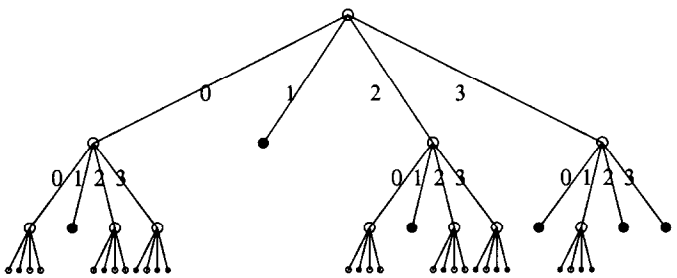


Fig. 4. Data structures of macro-subdivision.



(A): Curve in parametric space Ω_1 after SSI



(B): Corresponding Quadtree Representation

• : denotes a non-vital area

○ : denotes a vital area

$(i)^+, i \in \{0, 1, 2, 3\}$ are vertex names

e_i, $i \in \{0, \ldots, 15\}$ are edge curves in leaf nodes

E(002) = e_{13} : means node (002) contains edge curve e_{13}

Fig. 5. Intersection curve in parametric domain and corresponding quadtree data structure.

3.3. Connecting curve segments

The computation of the subdivision algorithm produces a number of line segments, that are solutions of triangle/triangle intersections. These curve segments, stored in the quadtree, must be connected to form the intersection curve. In this section, we present a parallel solution for curve connecting, that is a variant of curve tracing using the neighbor following method. The neighbor following method requires the algorithm to traverse up and down the quadtree to find the neighbor of a node. Thus it requires a $\Theta(nr)$ computation, where r is the height of the quadtree and n is the number of the leaves where the curve passes through their corresponding sub-domain. We will refer to an area with a curve passing through as a *vital area*; a leaf with a curve passing through is called a *vital leaf*, see Fig. 5.

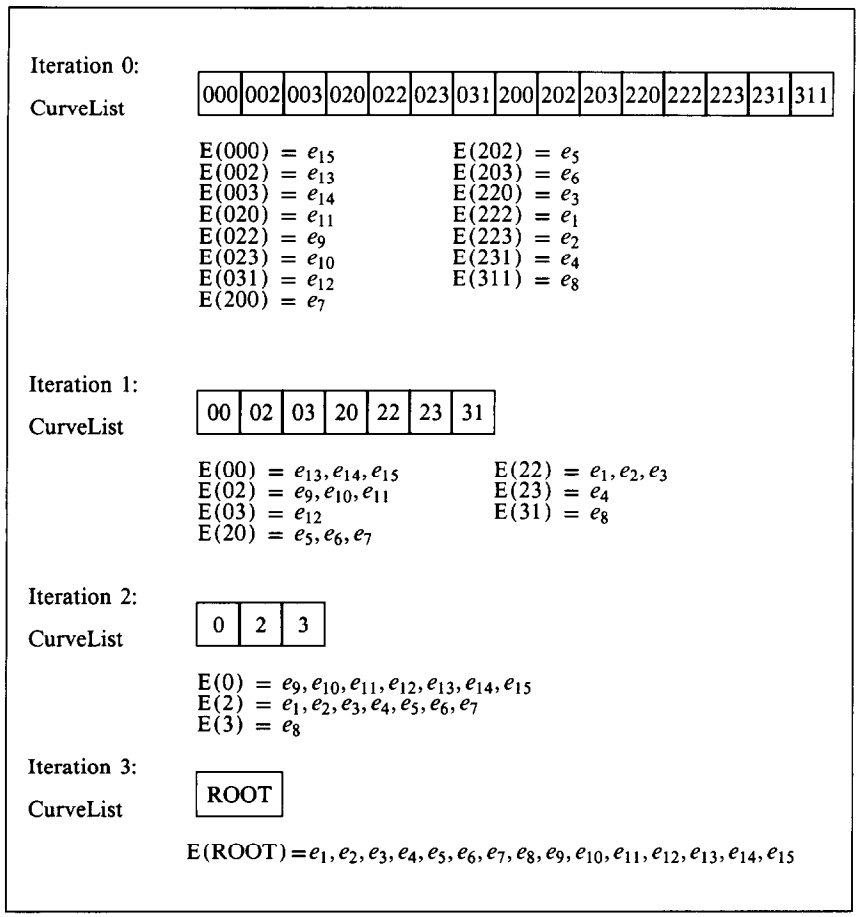


Fig. 6. Illustration of connecting phase.

Rather than going up and down the quadtree each time we need to find a neighbor for a node, instead we keep a list of pointers which point to the vital nodes in the quadtree with the same level. At the beginning of the connecting phase this list contains the vital leaves in the quadtree. It then constructs a new list that contains those pointers which are parents of the leaves in the old list. For each parent node, all the curve segments within the area corresponding to it are connected. Each time the algorithm goes up one level, it merges (connects) curve segments and forms a new list. The process proceeds iteratively until it reaches the root of the tree (see Figs. 5, 6). This procedure can be viewed as similar to a bottom-up merge sort.

Our solution has the same complexity, $\Theta(nr)$, as in the sequential neighbor following approach. In the parallel case, if there are P processors available, then our solution may achieve nearly linear speed up, i.e. complexity of $\Theta(nr/P)$. The details of this algorithm can be found in (Chang, 1990).

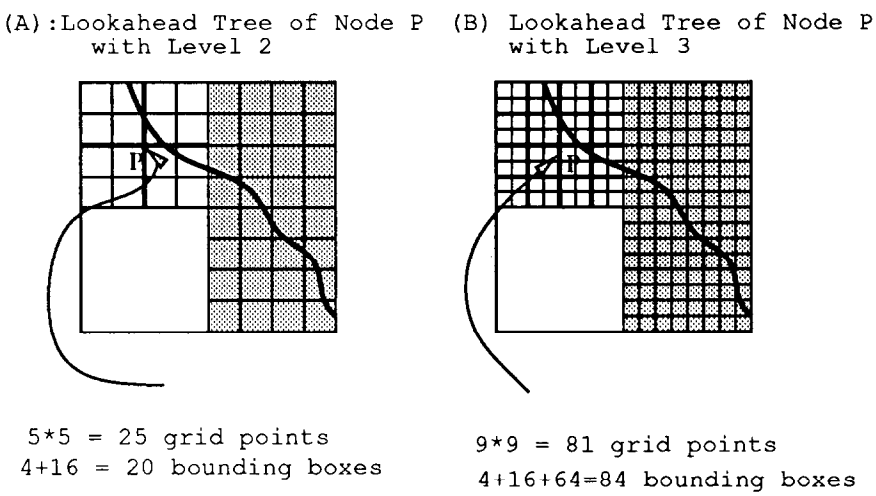


Fig. 7. Tasks in lookahead subdivision levels.

3.4. *Lookahead subdivision algorithm*

The lookahead-subdivision algorithm is a hybrid multi-level parallel algorithm that takes advantage of the vector pipeline features of some parallel machines. One advantage of this algorithm is that the macro-subdivision algorithm can be embedded into the outer level of the lookahead-subdivision algorithm, and the inner level, the subdivision process, is implemented in a lookahead manner to take advantage of vectorization[3].

For each surface, a subdivision process computes the surface function at five grid points in the parametric domain (referred to as grid evaluation) and computes four bounding boxes. Although we could vectorize this calculation by implementing vector functions which compute grids and bounding boxes, the improvement would not be significant since the vector length is much too short to take full advantage of vector pipeline features. Instead, a strategy called "lookahead" is used to increase the vector length of the vector functions.

The strategy is based on the following *inheritance property*: if a curve passes through a node in the quadtree, it must also pass through at least one node of its children. Hence, if a node is subdivided then some children will have to be dealt with later. The lookahead-subdivision algorithm precomputes grids and bounding boxes in the next subdivision level using vector functions. For example, in our algorithm, there are 25 grid points and 20 bounding boxes to be computed if it looks ahead into the next descendent level. Similarly, 81 grids and 84 bounding boxes need to be computed if it looks ahead into the next two descendant levels (see Fig. 7).

The lookahead feature increases the computation of a subdivision task only

[3]Vectorization is a process to restructure a program in a form that the compiler can generate code that takes advantage of vector hardware.

slightly by taking advantage of vectorization, but there will be significant overall savings since the algorithm does not need to compute grid points and bounding boxes in the next lookahead levels. Therefore given the *inheritance property*, the speed can be significantly improved. There are limits in the size of the lookahead tree. First, a large tree might cause a memory overuse problem, if memory is not carefully managed. Second, the performance of a lookahead-subdivision method strongly depends on the fraction of the subdivision code which can be vectorized, the underling system architecture and the lookahead level. If the lookahead level is too high then the performance of the lookahead-subdivision can be worse than its counterpart, the macro-subdivision method. Therefore, the lookahead tree should be kept small. In our implementation, a lookahead level of 2 or 3 was sufficient to improve significantly the performance of the lookahead-subdivision algorithm over the macro-subdivision algorithm. For the detailed complexity analysis of the lookahead subdivision algorithm and how to choose the lookahead level, see Appendix B. A pseudo code of the lookahead surface split algorithm is as follows:

```
Lookahead_Split(s,Qs,d)
/* pseudo code for subdivision algorithm  which precomputes the grids
 and bounding boxes of its lookahead tree with level d */
SurfacePtr s;
QuadtreePtr Qs;
int d;  /* lookahead level */
{
        Initialize and update quadtree data structure of this node(Qs).
        if the corners of this node (sub-surface) have not been computed
        {
              precompute the grids and bounding boxes of its lookahead
              tree with level d  and store this information into node
              Qs in the quadtree data structure (see Fig. 7).
              /* implemented in vectorization code */
        }
        else {
              get its grid and bounding box info from the parent node
              of Qs.  update bounding boxes of child nodes of Qs.
              if the children of Qs are termination nodes then
                 update corners of the child nodes of Qs.
        }
        update split flag of the quadtree;
}
```

4. Empirical results

We first implemented the macro-subdivision algorithm (without vectorization) on a Sequent Balance 21000, which is a MIMD 24-processor machine at the Argonne Advanced Computing Research Facility. Fig. 8 shows the test case, which represents a common case of surface intersections. The benchmarks for these tests are summarized in Fig. 9; The results indicates that the performance of the macro-subdivision algorithm is linearly proportional to the inverse of

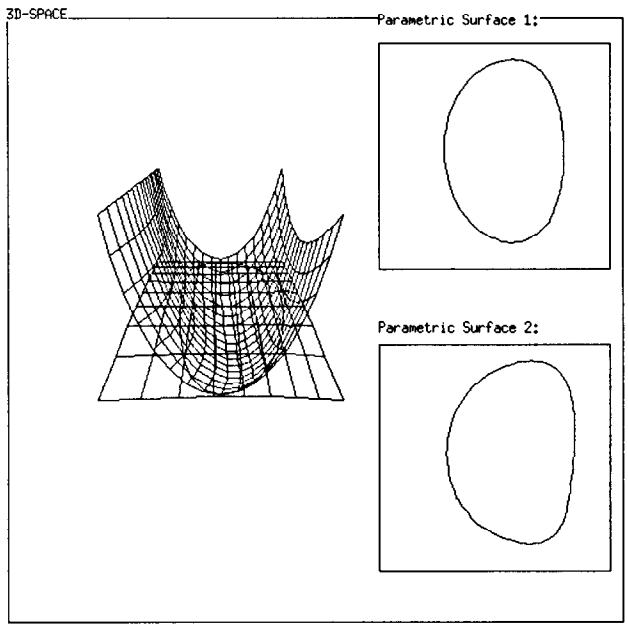


Fig. 8. Surface intersection.

the given tolerance as predicted by Proposition 2.4. The results indicate that speed-up is especially high for up to 16 processors. For 12 processors the speed-up is about 8. For 16 processors the speed-up is about 9. After 16 processors, the performance might be a little slower as more processors are added. These results suggested our approach might be well suited for a 4-processor Cray-2 architecture and 8-processor Alliant FX/8 architecture. Implementations were tested on the Cray-2 at the National Supercomputing Application Center, University of Illinois at Champaign and the Alliant FX-8 at Argonne. We used the MONMACS parallel programming library developed at Argonne (Lusk, 1987) to make implementations of the macro-subdivision and lookahead-subdivision algorithms as portable as possible, On the Cray-2 and the Alliant FX/8, we first implemented the pure Macro-Subdivision algorithm, and then added the lookahead features.

Figs. 10 and 11 show the performance comparison of lookahead-subdivision (level 2, 3) and macro-subdivision with and without vectorization for various tolerance inputs on the Cray-2 and Alliant FX-8, respectively. The performance of the lookahead-subdivision method is much better than the macro-subdivision method with and without vectorization. The speed-up of the lookahead-subdivision method was up to a factor of 6 in Cray-2 and 32 on the Alliant (showing better than linear speed-up). We conclude that, given the underlying architectures, our implementations of macro-subdivision and lookahead-subdivision algorithms are close to the expected theoretical bounds.

It is interesting to note that we can increase the precision tolerance and make up for the loss of speed by using a proportional number of processors. This gives

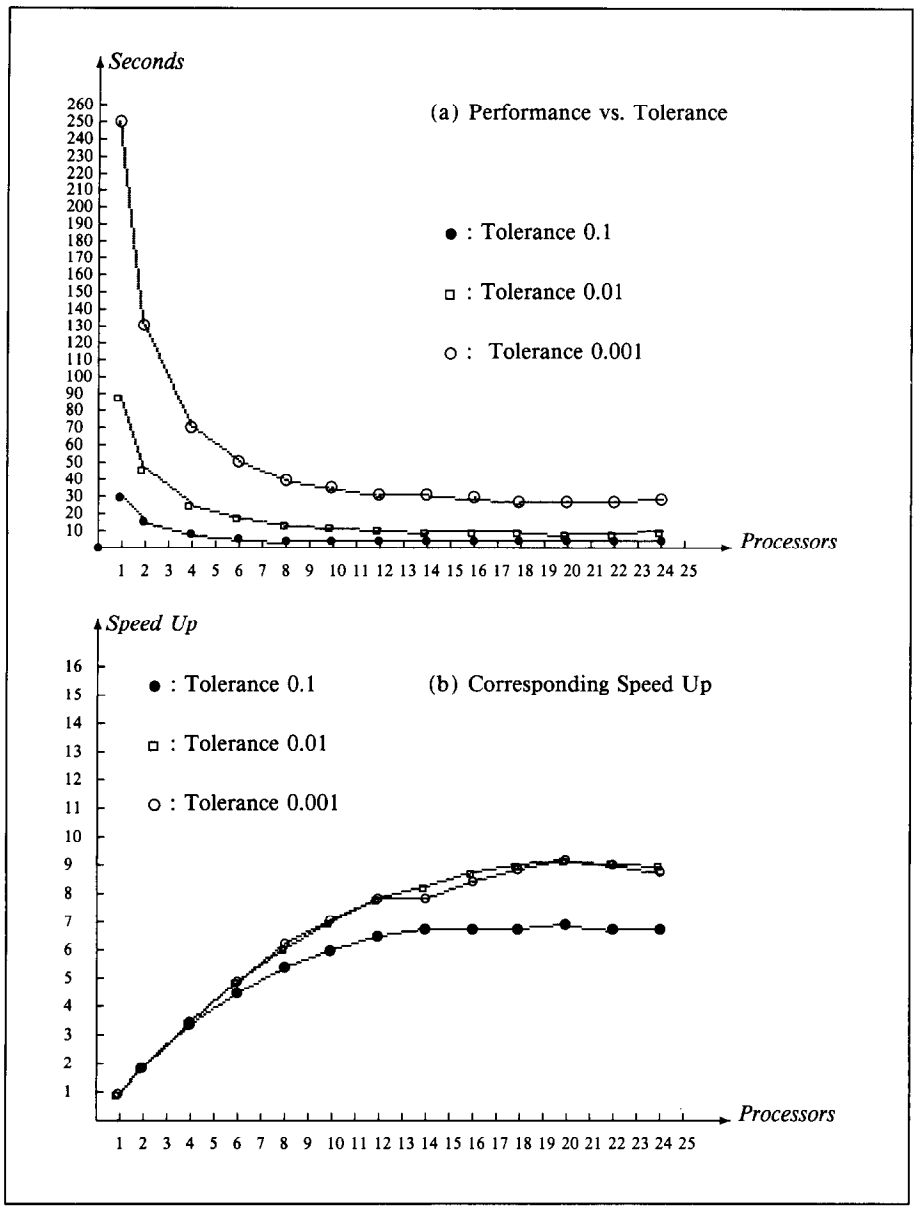


Fig. 9. Performance of macro-subdivision method – Sequent Balance 21000: Argonne National Lab (ACRF).

substance to our claim that the trade-off between speed and precision can be improved by using parallel processing. We also note that for MIMD machines, although an almost linear speed-up can be obtained, in actual implementations there is an optimal number of processors that achieves the most effective use of available processors. For example from Fig. 10, we notice that the second half of processors does not contribute as effectively as the first half

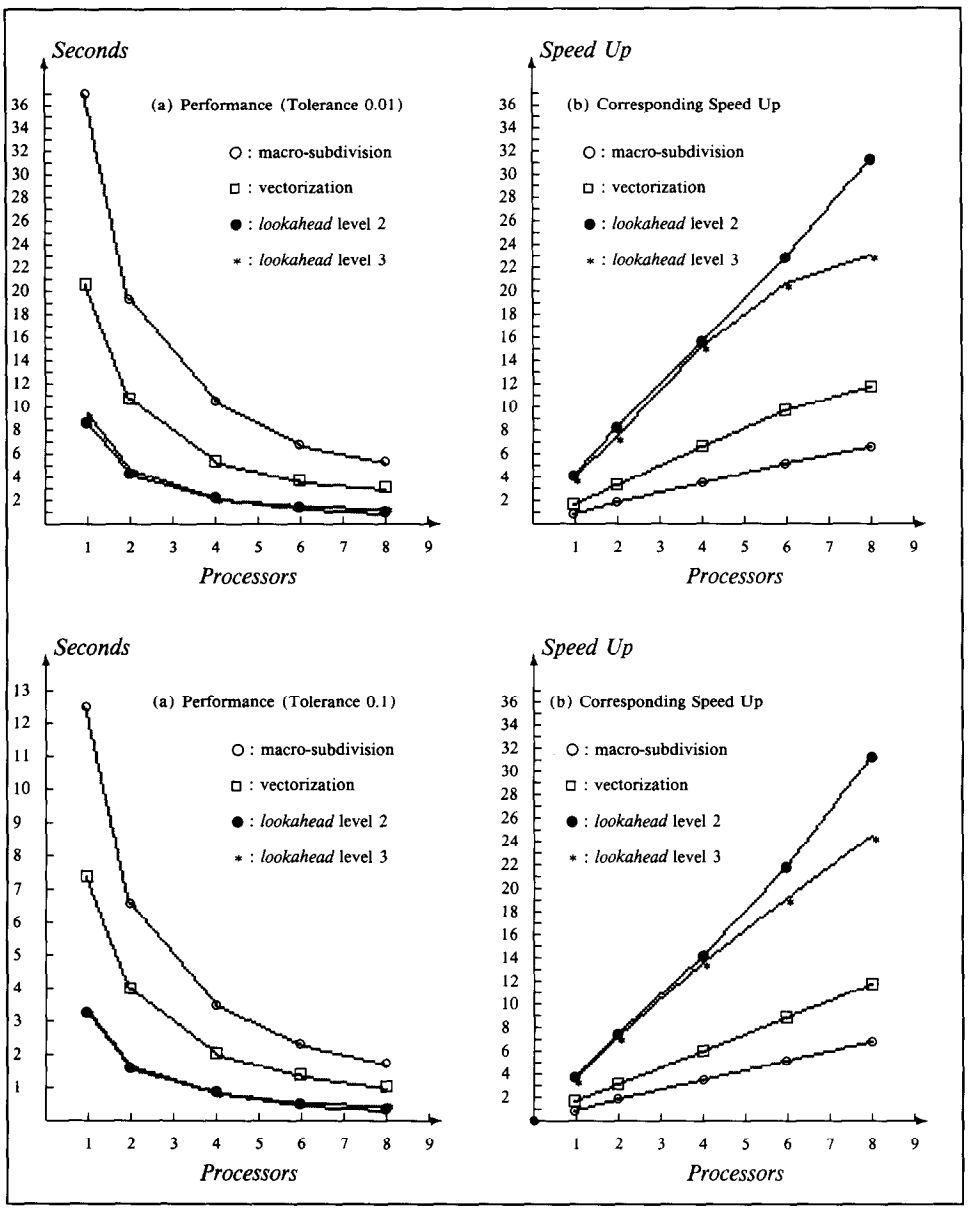


Fig. 10. Comparison of lookahead with macro-subdivision (vectorization) – Alliant FX/8: Argonne National Lab (ACRF).

of processors. Since surface intersections are the "lowest level" computation in support of Boolean set operations, in practice we should find the optimal number of processors, thus freeing any additional processors for independent tasks. This would make multi-level parallelism very useful in a higher level computation (such as the intersection of two objects) since we could divide the processors into several groups of processors, each group working independently

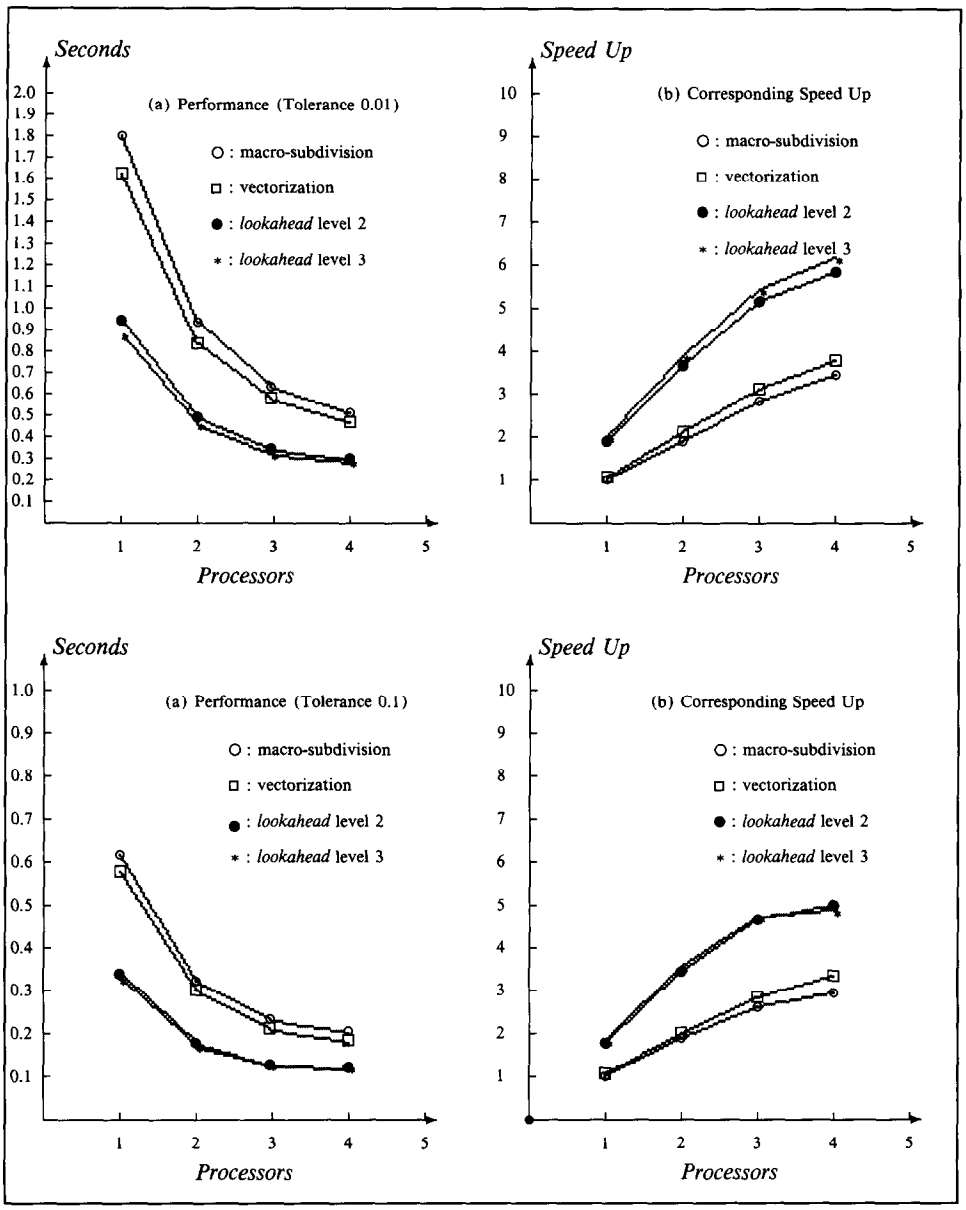


Fig. 11. Comparison of lookahead with macro-subdivision (vectorization) – CRAY-2: National Center for Supercomputer Application (NCSA).

on different tasks simultaneously.

5. Conclusion

We have shown both theoretically and empirically that parallelism is a feasible approach for solving the speed-precision dilemma in surface intersection.

Although our current study of parallelism is primarily on the SSI context, it is also possible to utilize our results on other problems in computer aided geometric design. For example, the curve-tracing technique used in surface intersection can also benefit from parallelism since we can divide the curve into curve branches on different regions using the subdivision method and then trace each curve branch simultaneously, see (Chang, 1991).

As another example, consider the intersection of two solids. For coarse or middle grain parallel computers, we can feed a pair of surface patches into each processor and keep all processors as busy as possible, hopefully achieving a near linear speed-up. Such a strategy is very simple but can be extended as in this paper. For example, in vector pipe-lined parallel architectures, the lookahead-subdivision algorithms can provide better than linear speed up. In a massive parallel computer with thousands of processors multi-level parallelism might be a better strategy. Our algorithms can be used on the fairly simple processors used in massively parallel architectures.

Our exploration of parallelism in the design of a new free-from solid modeling system is still far from complete. Developing parallel algorithms for MIMD distributed memory machines is an important issue. MIMD distributed memory machines are the only type of MIMD machines that are scalable, i.e. the performance of the method is proportional to the number of available processors. In addition, their price is far less expensive than serial supercomputers. Developing a prototype system that can support Boolean set operations (union, difference, and intersection) to construct solids from primitives using multi-level parallel processing techniques is another area for future work.

Appendix A

In the following, we give a lemma and a proposition which state the profile and complexity of subdivision algorithms using a quadtree data structure. We will use the following notation:

Notation.

- ε: input tolerance;
- M_i: the curvature bound of surface i (defined in Proposition 2.3);
- n_i: subdivision level (termination level) of surface i;
- T_{ssi}: time for computing a SSI;
- TSD_i: total computation time of surface i in subdivisions;
- TSD: total subdivision time (i.e., $TSD_1 + TSD_2$);
- T_{e_i}: time for computing a parametric function at a grid point; of surface i;
- T_{box}: time for computing a bounding box of a sub-surface;
- T_{up}: time for updating quadtree data structure for a sub-surface;
- T_{bbt}: time for detecting if a pair of bounding boxes intersects;
- N_{bbx}: number of bounding box tests in SSI subdivision process;
- $T_{\text{sdv}\,i}$: time for subdividing a sub-patch of surface i;
- $N_{\text{sdv}\,i}$: total number of subdivision of surface i in SSI;

- T_{startup}: time for starting up an subdivision algorithm for SSI;
- N_{tasks}: number of tasks (pairs of sub-surfaces) added into queue;
- T_{qop}: time for a queue operation (add/delete);
- T_{tti}: time for computing a triangle/triangle intersection;
- N_{tti}: total number of triangle/triangle intersections in a SSI.

Lemma A.1. *Given two parametric surfaces which are images under P_1 and P_2 of rectangles in E^2, assume both surfaces intersect each other. The computation time for SSI using the subdivision method (Section 2.3) and a quadtree data structure is*

$$T_{\text{ssi}} = \left\{ \sum_{i=1}^{2} N_{\text{sdv}\,i} \cdot [5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}})] \right\} + N_{\text{bbt}} \cdot T_{\text{bbt}} + N_{\text{tti}} \cdot T_{\text{tti}} + 2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}} + T_{\text{start}}.$$

Proof. For the subdivision algorithm, we first compute the start up time of SSI. We need to compute the parametric functions P_1 and P_2 at the four corners of their rectangular domains; a bounding box for each surface; one bounding box intersection test; update quadtree data structure operations for both surfaces and one queue operation. Therefore, the start up time is:

$$T_{\text{start}} = 4 \cdot T_{e_1} + 4 \cdot T_{e_2} + 2 \cdot T_{\text{box}} + 2 \cdot T_{\text{up}} + T_{\text{qop}} + T_{\text{bbt}}. \tag{5}$$

For the Surface_Split (subdivision process) in Section 2, we must compute the following for each subdivision:

1. parametric functions at 5 grid points of a surface;
2. an update to the quadtree data structure for each of four subdivided sub-surfaces;
3. a bounding box for each subdivided sub-surface.

By the above, the time for each subdivision of a sub-surface of surface i is:

$$T_{\text{sdv}\,i} = 5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}}). \tag{6}$$

Total Subdivision Time (*TSD*):

$$N_{\text{sdv}\,1} \cdot T_{\text{sdv}\,1} + N_{\text{sdv}\,2} \cdot T_{\text{sdv}\,2}.$$

Total time for bounding box test is:

$$N_{\text{bbt}} \cdot T_{\text{bbt}}.$$

The total time for queue operations is the time for the number of tasks added into the queue plus the time for the tasks deleted from queue. Both are equivalent. Therefore, the queue operation time is:

$$2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}}.$$

By the above, the computation time of SSI using subdivision and a quadtree data structure can be estimated by

$$
\begin{aligned}
T_{\text{ssi}} &= N_{\text{sdv}\,1} \cdot T_{\text{sdv}\,1} + N_{\text{sdv}\,2} \cdot T_{\text{sdv}\,2} \\
&\quad + N_{\text{bbt}} * T_{\text{bbt}} + N_{\text{tti}} * T_{\text{tti}} + 2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}} + T_{\text{startup}} \\
&= \left\{ \sum_{i=1}^{2} N_{\text{sdv}\,i} \cdot [5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}})] \right\} \\
&\quad + N_{\text{bbt}} \cdot T_{\text{bbt}} + N_{\text{tti}} \cdot T_{\text{tti}} + 2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}} + T_{\text{start}}. \qquad \square
\end{aligned}
$$

Proposition 2.4. *Given two C^2 parametric surfaces P_1 and P_2, and a tolerance ε. Assuming the surfaces intersect, the computation time for SSI using subdivision method and quadtree data structure has lower bound complexity $\Omega(\log_4((M_1 + M_2)/\varepsilon))$ and upper bound complexity $O(((M_1 + M_2)/\varepsilon)^2)$, where M_1, M_2 are the curvature bounds of surfaces P_1 and P_2, respectively.*

Proof. Let n_1 be the subdivision level for surface P_1, and n_2 for surface P_2; and $n_1 \leqslant n_2$. The number of rectangular subdomains at the subdivision level for the two surfaces are 4^{n_1} and 4^{n_2} respectively. Both surfaces are approximated within $\varepsilon/2$. By equation (4), we can easily derive the following equations to compute n_1 and n_2 such that the intersection curve of two surfaces is within the input tolerance ε:

$$n_1 \leqslant \log_4(M_1/\varepsilon), \tag{7}$$

$$n_2 \leqslant \log_4(M_2/\varepsilon). \tag{8}$$

The number of bounding box tests (N_{bbt}), subdivisions ($N_{\text{sdv}\,i}$), triangle/triangle intersections (N_{tti}), and queue operations (N_{tasks}) depends on the spatial orientation , the curvature of two surfaces and the input tolerance ε. By equations (7) and (8), we can compute the lower, practical, and upper bounds for $N_{\text{sdv}\,i}$, N_{bbt}, N_{qop} and N_{tti} in terms of the curvature of two surfaces and input parameter ε, based on the spatial orientation of the two surfaces.

Best case situation (lower bound)

In this case, the intersection curve passes through only one node at the lowest subdivision level of each surface. That is, at each level, only one subdivision occurs for each surface. Thus, if the subdivision level of a surface i is n_i, then $N_{\text{sdv}\,i} = n_i$, which is the lower bound for $N_{\text{sdv}\,i}$; and the lower bound for N_{tti} is 1, since only one computation of a triangle/triangle intersection (TTI) is needed. The number of bounding box tests at the level below or equal to n_1 is 16, otherwise it is 4. Hence by equations (7) and (8)

$$
\begin{aligned}
N_{\text{bbt}} &= 16 \cdot n_1 + 4 \cdot (n_2 - n_1) \\
&= 12 \cdot n_1 + 4 \cdot n_2 \\
&= \Theta\left(\log_4 \frac{M_1 + M_2}{\varepsilon}\right).
\end{aligned}
$$

In each level, only one task is added into the queue, hence

$$N_{\text{tasks}} = n_2 = \Theta\left(\log_4 \frac{M_1 + M_2}{\varepsilon}\right).$$

By Lemma A.1,

$$\begin{aligned} T_{\text{ssi}} &= \left\{ \sum_{i=1}^{2} N_{\text{sdv}\,i} * [5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}})] \right\} \\ &\quad + N_{\text{bbt}} \cdot T_{\text{bbt}} + N_{\text{tti}} \cdot T_{\text{tti}} + 2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}} + T_{\text{start}} \\ &= \left\{ \sum_{i=1}^{2} n_i * [5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}})] \right\} \\ &\quad + (12 \cdot n_1 + 4 \cdot n_2) \cdot T_{\text{bbt}} + 1 \cdot T_{\text{tti}} + 2 \cdot n_2 \cdot T_{\text{qop}} + T_{\text{start}}. \end{aligned}$$

Using the above arguments, it can be easily shown that there exists a C, such that

$$T_{\text{ssi}} \geqslant C \cdot \log_4 \frac{M_1 + M_2}{\varepsilon}.$$

Therefore,

$$T_{\text{ssi}} = \Omega\left(\log_4 \frac{M_1 + M_2}{\varepsilon}\right).$$

Worst case situation (upper bound)

This case assumes that both surfaces nearly intersect each other everywhere (nearly overlap). That is, in each subdivision level, the subdivided surface-patches of both surfaces will intersect each other. Therefore, for a surface i, in level 0, one subdivision is needed; in level 1, 4 subdivisions are needed, and in level 2, 16 subdivisions are needed. By induction, at level n_{i-1}, $4^{n_i - 1}$ subdivisions are needed. Thus,

$$N_{\text{sdv}\,i} = \sum_{i=0}^{n_{i-1}} 4^i = \frac{4^{n_i} - 1}{3} \leqslant \frac{M_i/\varepsilon - 1}{3} = \Theta\left(\frac{M_1 + M_2}{\varepsilon}\right).$$

The total number of pairs of sub-surfaces for triangle/triangle intersection, therefore, is the product of the number of sub-surfaces in level n_i of surface i. That is,

$$\begin{aligned} N_{\text{tti}} &= 4^{n_1} \cdot 4^{n_2} \leqslant \frac{M_1}{\varepsilon} \cdot \frac{M_2}{\varepsilon} = \frac{M_1 M_2}{\varepsilon^2} \leqslant \left(\frac{M_1 + M_2}{\varepsilon}\right)^2 \\ &= \Theta\left(\left(\frac{M_1 + M_2}{\varepsilon}\right)^2\right). \end{aligned}$$

The total number of bounding box tests is:

$$N_{\text{bbt}} = \sum_{i=1}^{n_1} 16^i + 16^{n_1} \cdot \sum_{i=1}^{n_2-n_1} 4^i \leqslant \sum_{i=1}^{n_2} 16^i$$
$$\leqslant \frac{16^{n_2+1}-1}{15} - 1 \leqslant \frac{16 \cdot (M_2/\varepsilon)^2 - 1}{15} - 1$$
$$= \Theta\left(\left(\frac{M_1 + M_2}{\varepsilon}\right)^2\right).$$

The number of tasks added into the queue is the same as the number of bounding box tests in the worst case situation:

$$N_{\text{tasks}} = N_{\text{bbt}} = \Theta\left(\left(\frac{M_1 + M_2}{\varepsilon}\right)^2\right).$$

By Lemma A.1 and above arguments, we have

$$T_{\text{ssi}} = \left\{\sum_{i=1}^{2} N_{\text{sdv}\,i} * [5 \cdot T_{e_i} + 4 \cdot (T_{\text{box}} + T_{\text{up}})]\right\}$$
$$+ N_{\text{bbt}} \cdot T_{\text{bbt}} + N_{\text{tti}} \cdot T_{\text{tti}} + 2 \cdot N_{\text{tasks}} \cdot T_{\text{qop}} + T_{\text{start}}$$
$$= \mathrm{O}\left(\left(\frac{M_1 + M_2}{\varepsilon}\right)^2\right).$$

Observation 2 (*Practical bound*). If we assume that in each subdivision step, half of regions are eliminated by bounding box tests then the complexity is $\mathrm{O}((M_1 + M_2)/\varepsilon))$. This can be shown as follows.

By the same principle as in the worst case,

$$N_{\text{sdv}\,i} = \sum_{i=0}^{n_{i-1}} 2^i = 2^{n_i} - 1 \leqslant \frac{(M_i)^{1/2}}{\varepsilon} - 1 = \Theta\left(\left(\frac{M_1 + M_2}{\varepsilon}\right)^{1/2}\right).$$

The number of bounding box tests at level 1 is 16, at level 2 it is 64 $(16^2 \cdot \frac{1}{4})$, at level 3 it is $16^3 \cdot \frac{1}{4^2}$ and so on. By induction, the total number of bounding box tests from level 1 to n_1 is $\sum_{i=1}^{n_1} 16^i/4^{i-1}$, and the total number of bounding box tests from level $n_1 + 1$ to n_2 is $16 \cdot 4^{n_1-1} \sum_{i=1}^{n_2-n_1} 2^i$. Therefore,

$$N_{\text{bbt}} = \sum_{i=1}^{n_1} 16^i \cdot \frac{1}{4^{i-1}} + 16 \cdot 4^{n_1-1} \sum_{i=1}^{n_2-n_1} 2^i$$
$$\leqslant 16 \sum_{i=1}^{n_2} 4^i$$
$$\leqslant \frac{M_2}{\varepsilon} \cdot 256$$
$$= \Theta\left(\frac{M_1 + M_2}{\varepsilon}\right).$$

The number of tasks added into queue is less than the number of bounding box tests and thus

$$N_{\text{qop}} \leqslant N_{\text{bbt}} = \Theta\left(\frac{M_1 + M_2}{\varepsilon}\right).$$

The number of uneliminated regions in level n_1 of surface 1 are 2^{n_1}. Similarly, the number of uneliminated regions in level n_2 of surface 1 are 2^{n_2}. Therefore,

$$N_{\text{tti}} = 2^{n_1} \cdot 2^{n_2} \leqslant \frac{(M_1 M_2)^{1/2}}{\varepsilon} = \Theta\left(\frac{M_1 + M_2}{\varepsilon}\right).$$

By Lemma A.1 and the above computation,

$$T_{\text{ssi}} = \mathrm{O}\left(\frac{M_1 + M_2}{\varepsilon}\right). \qquad \square$$

Appendix B

In this appendix, we show why the lookahead-subdivision method can outperform the macro-subdivision method with the help of an appropriate lookahead tree and how to choose an appropriate lookahead level. We define three entities:

1. *lookahead*(d) *tree* is a complete quadtree with height d (see Fig. 7);
2. *lookahead*(d) *subdivision time* is the total time for computing all the necessary subdivision tasks for a lookahead(d) tree in the macro-subdivision or lookahead-subdivision algorithm;
3. a *lookahead-split*(d) *job* is a subdivision job with lookahead level d.

According to these definitions, a *lookahead*(1) subdivision process is a process which subdivides a node in the quadtree by using the macro-subdivision or lookahead-subdivision method. Therefore, we can view the macro-subdivision algorithm as a special case of the lookahead-subdivision algorithm with lookahead level 1.

In what follows, we give a derivation of the complexity of a *lookahead*(d) subdivision process which uses the lookahead-subdivision algorithm and the macro-subdivision algorithm with vectorization.

Let

- T_{s} = the time required to perform an operation in scalar mode;
- T_{v} = the time required to perform an operation in vector mode;
- VS = vector speed up (T_s/T_{v}) of the underlying machine;
- $N(d)$ = the total number of operations of a *lookahead-split*(d) task;
- $F_{\text{s}}(d)$ = the fraction of operations of a *lookahead-split*(d) task computed in sequential mode;
- $F_{\text{v}}(d)$ = the fraction of operations of a *lookahead-split*(d) task computed in vector mode;
- $T_{\text{bg}}(d)$ = the time required to perform all bounding box computation and grid evaluations in *lookahead-split*(d);

- $T_{up}(d)$ = the time required to perform quadtree update operations in a *lookahead-split*(d);
- $T_{ms}(d)$ = the time required to perform a *lookahead*(d) subdivision process using macro-subdivision method;
- $T_{msv}(d)$ = the time required to perform a *lookahead*(d) subdivision process using macro-subdivision method with vectorization;
- $T_{la(d)}$ = the time required to perform a *lookahead*(d) subdivision process using lookahead method;
- $SDV(i)$ = total number of subdivision tasks needed to perform in the ith level of a lookahead tree with level d.

By definition, $VS = T_s/T_v$ and $F_s(i) = 1 - F_v(i)$, $i = 1,\ldots,d$.

For the macro-subdivision method, we have

$$T_{ms}(1) = T_{bg}(1) + T_{up}(1) = N(1) \times T_s.$$

For the macro-subdivision method with vectorization

$$\begin{aligned} T_{msv}(1) &= T_{bg}(1) + T_{up}(1) = N(1) \times F_v(1) \times T_v + N(1) \times F_s(1) \times T_s \\ &= N(1) \times F_v(1) \times T_v + N(1) \times (1 - F_v(1)) \times T_s \\ &= N(1) \times F_v(1) \times \frac{T_s}{VS} + N(1) \times T_s \times (1 - F_v(1)) \\ &= N(1) \times T_s \times \left(1 - \frac{VS - 1}{VS} \times F_v(1)\right). \end{aligned}$$

Therefore the speed up for vectorization is

$$Speed\ Up = \frac{T_{ms}(1)}{T_{msv}(1)} = \frac{1}{[1 - (VS - 1)/VS) \times F_v(1)]}. \tag{9}$$

One can view these results as the vectorization version of Amdahl's law (Levesque, 1989). It shows that speed up for vectorization is related to the vector speed-up of the underlying machine and the fraction of vectorizable code.

In the following, we show why the lookahead-subdivision method can peroform better than the macro-subdivision method with vectorization. Both methods utilize vectorization method for computing all bounding boxes and grid evaluations in *lookahead_split*(d) and *lookahead_split*(1), respectively. Conceptually, the lookahead-subdivision method has more data for vectorization than the macro-subdivision method; while the quadtree update time are about the same since they are done in scalar mode.

For the lookahead method, *lookahead*(d) subdivision time is

$$T_{la}(d) = T_{bg}(d) + \sum_{i=1}^{d} SDV(i) \times T_{up}(d).$$

For its corresponding macro-subdivision method with vectorization

$$T_{msv}(d) = \sum_{i=1}^{d} SDV(i) \times T_{msv}(1)$$
$$= \sum_{i=1}^{d} SDV(i) \times T_{bg}(1) + \sum_{i=1}^{d} SDV(i) \times T_{up}(1).$$

We want to choose lookahead level (d) so that the performance of the lookahead version is better than the macro version. That is,

$$T_{la}(d) \leqslant T_{msv}(d).$$

By the above equations, we have

$$T_{bg}(d) \leqslant \sum_{i=1}^{d} SDV(i) \times T_{bg}(1), \tag{10}$$

$$N(d) \times F_v(d) \times \frac{T_s}{VS} \leqslant \sum_{i=1}^{d} SDV(i) \times N(1) \times F_v(1) \times \frac{T_s}{VS}. \tag{11}$$

By definition, we have

$$F_v(d) \geqslant F_v(1), \qquad N(d) \geqslant N(1).$$

Both $N(d)$ and $\sum_{i=1}^{d} SDV(i)$ are exponential functions. However, $N(d)$ grows faster than the sum in most situations when d becomes large. For small d, the above equation is satisfied, since the ratio $N(d)/N(1)$ is much smaller than $\sum_{i=1}^{d} SDV(i)$.

References

Barnhill, R.E., Farin, G., Jordan, M. and Piper, R.B. (1987), Surface/surface intersection, Computer Aided Geometric Design 4, 3–16.

Casale, M.S. and Stanton, E.L. (1985), An overview of analytic solid modeling, IEEE Comput. Graph. Appl. 5, 45–58.

Chang, L.-C., Bein, W.W. and Angel, E. (1990), Parallel algorithms for surface intersection, Tech. Report No. CS 90-7, Department of Computer Science, University of New Mexico.

Chang, L.-C. (1991), Parallel algorithms for surface intersection for free-form solid modeling system, Ph.D. Dissertation, Department of Computer Science, University of New Mexico.

Crocker, G.A. and Reinke, W.F. (1987), Boundary evaluation of non-convex primitives to produce parametric trimmed surfaces, Computer Graphics 21, 129–136.

Farouki, R.T. (1987), Trimmed-surface algorithms for the evaluation and interrogation of solid boundary representations, IBM J. Res. Develop 31, 314–334.

Filip, D., Magedson R. and Markot R. (1986), Surface algorithms using bounds on derivatives, Computer Aided Geometric Design 3, 295–311.

Herzen, B.H. (1989), Applications of surface networks to sampling problems in computer graphics, Ph.D. Dissertation, California Institute of Technology, Computer Science Department, Caltech-CS-TR-88-15.

Herzen, B.V., Barr, A.H. and Zatz, H.R. (1990), Geometric collisions for time-dependent parametric surfaces, Computer Graphics 24, 39–48.

Hoffmann, C.M. (1989), *Geometric and Solid Modeling: An Introduction*, Morgan Kaufmann, Los Altos, CA.

Houghton, W.G. and Emnett, R. (1985), Implementation of a divide-and-conquer method for intersection of parametric surface, Computer Aided Geometric Design 2, 173–183.

Kala, D. and Barr, A.H. (1989), Guaranteed ray intersections with implicit surfaces, Computer Graphics, 23, 297-306.

Lane J.M. and Carpenter L.C. (1979), A generalized scan line algorithm for the computer display of Parametrically defined surfaces, Computer Graphics and Image Processing 11, 290–297.

Lane, J.M. and Riesenfeld, R.F. (1981), A theoretical development for the computer display and generation of piecewise polynomial surfaces, IEEE Trans. Pattern Anal. Mach. Intel. 2, 35–46.

Levesque, J.M. and Williamson, J.W. (1989), *A Guidebook to Fortran on Supercomputers*, Academic Press, New York.

Lusk, E. and Overbeek, R. (1987), *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York.

Pratt, M.J. and Geisow, A.D. (1986), Surface/Surface intersection problems, in: J.A. Gregory, ed., *The Mathematics of Surfaces*, Oxford University Press.

Wilson, P.R. (1988), Solid modeling research and applications in the U.S.A., in: Wozny, M.J., McLaughlin, H.W. and Encarnaçao, J.L., eds., *Geometric Modeling for CAGD Applications*, North-Holland, Amsterdam.

Wolfe, R.N., Wesley, M.A., Kyle, J.C., Gracer, J.F. and Fitzgerald, W.J. (1987), Solid modeling for production design, IBM J. Res. Develop. 31, 277–295.