Contributions

# Surface intersections using parallel subdivision

Long Chyr Chang,
Wolfgang Bein,
Edward Angel

The University
of New Mexico,
Department of
Computer Science,
Farris Engineering
Center 321, Albuquerque
NM 87131, USA


*Supercomputer* 40, VII-6
*Received June 1990*

**Surface–surface intersection problems play an important role in the support of Boolean set operations in solid modeling systems. Present approaches are sequential and have to make trade-offs between accuracy, robustness and efficiency. In this paper, we will show that improved algorithms can be obtained using parallel methods and vectorization. We give a macro-subdivision algorithm for an MIMD shared memory machine. The overall approach is to successively subdivide the potentially intersecting regions of the surface patches in parallel until the subsurfaces are sufficiently flat. The intersection can then be obtained by direct rectangular or triangular intersections. The intermediate subdivisions are stored using quad trees. Process synchronization is controlled by the use of appropriate data structures. For an MIMD pipelined vector machine such as the Cray-2, the algorithm is adapted to vectorization using a "lookahead" strategy.**

A major problem in extending solid modeling systems to surfaces more complex than quadratic surfaces (the "free-form" surfaces) is the difficulty of the surface–surface intersection (SSI) problem (see Figure 1). In a survey article, Pratt and Geisow [1] point out that a successful SSI algorithm must have three properties: accuracy, robustness and efficiency. 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 CAD applications behavior as close to real-time as possible is desired. In order to make practical use of one of the present sequential algorithms, an implementator must make significant trade-offs between these properties (see [1, 2]). We show that parallel algorithms present an opportunity to surmount this difficulty.

Specifically, we will consider surfaces which are given as parametric images $P$ over a rectangular domain $\Omega$. $P : \Omega \to E^2$ is assumed to be a $C^2$ function. Given the two such surfaces $S_1$ and $S_2$, which are images under $P_1$ and $P_2$ of rectangles in $E^2$ and a tolerance $\epsilon$, the problem is to compute the intersection curves $\gamma_1 \subseteq \Omega_1$ and $\gamma_2 \subseteq \Omega_2$ such that

Supported by Sandia National Laboratories grant, SURP No. 05-9858 task9; Cray time grant from the National Center for Supercomputing Applications.

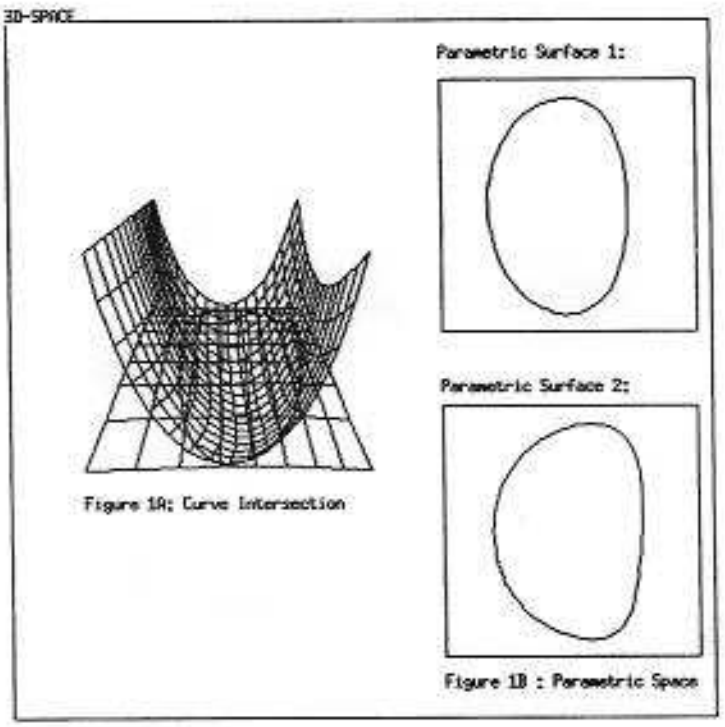


Figure 1. Surface intersection.

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

Filip, Magedson, and Markot [2] subdivide the domains $\Omega_1$, $\Omega_2$ successively into smaller subdomains. The idea is that after a sufficient number of subdivisions, the resulting subsurfaces will be flat within a specified tolerance. The flat subsurfaces can then be intersected as planes. If $\Omega_1$ is divided into $n_1$ pieces and $\Omega_2$ into $n_2$ pieces respectively, then the problem can be solved by considering $n_1 n_2$ subtasks. A brute-force method might proceed by performing all those tasks. Instead, for each subtask, we can compute "bounding boxes". These boxes give maximum ranges for the image values for the two subdomains. For two surfaces, S1 and S2, a conceptual algorithm might proceed as follows

- determine the bounding boxes of S1 and S2;
- if the bounding boxes do not intersect then return;
- determine the "flatness" of S1 and S2;
- if S1 and S2 are both flat then return the intersection curve of the planar approximations;
- else subdivide S1 and S2 into smaller surfaces and recursively apply the algorithm to all the combinations of all sub-surfaces of S1 and sub-surfaces of S2.

In what follows we present a detailed algorithm which has been implemented on a Cray-2. We will show not only the success of the algorithm but also the care which must be taken to implement a sequential algorithm on a parallel architecture.

**A parallel algorithm for SSI**

In this section we will present a parallel algorithm designed for an MIMD shared memory machine. The algorithm consists of two phases. First, we find the intersection edges. This part involves the main work and is called the *Computing Phase*. Second, we connect the edges to obtain consecutive curves. This part is called the *Connecting Phase*.

For the Computing Phase, the method described above has a straightforward parallelization. We subdivide both rectangular domains iteratively into smaller rectangles. If one domain is split into $n_1$ rectangles and the other domain is split into $n_2$ rectangles then the total number of tasks generated is $n_1 n_2$. Each task can be performed by a processor independently at each level of the iteration. For fine-grain parallel architectures, the domains are divided into many very small rectangles. Therefore, the depth of iteration will be low because for each of the rectangles the flatness condition will be achieved after just a few subdivisions. Then all the processors perform a simple intersection over two rectangles (referred to as IRR). Alternatively one can divide each rectangle once more into two triangles, and then perform the intersection of two triangles (ITT), which is easier and more accurate.

For the coarse and middle grain parallel architectures considered here, there are many levels of iteration. Some tasks have an empty intersection. Others either require further subdivision or have achieved the flatness condition and require surface intersection (ITT or IRR). Now processor synchronization becomes a crucial issue. To minimize process synchronization time, we must design the appropriate data structures to incorporate the subdivision method with quad tree data structures. The algorithm is named the Macro-subdivision method.

Specifically the rectangles will be subdivided into 4 subrectangles. This allows us to use two quad trees $T_1$ and $T_2$ to represent the subdivision (see Figure 2). Leaves at the deepest level of the quad tree store the intersection edges. Additionally, for every vertex in the quad tree, we keep a split flag that records whether the corresponding region has been subdivided.

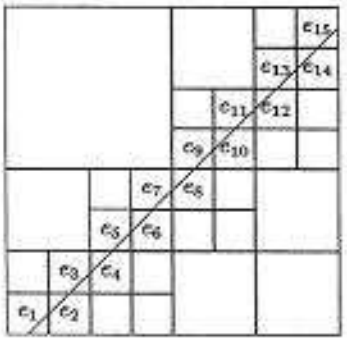


Figure 2A: Curve in parametric space $\Omega_1$ after SSI

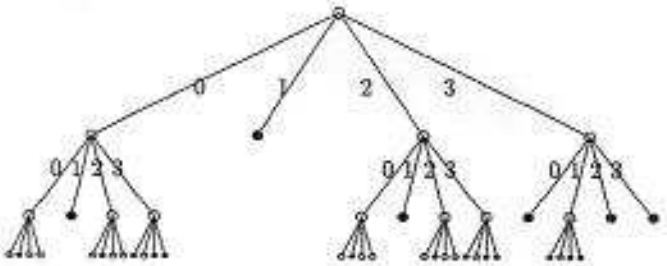


Figure 2B: Corresponding Quadtree Representation

- : denotes a non-vital area
- ○ : denotes a vital area

Figure 2. Curve in parametric space $\Omega_1$ and corresponding quad tree representation.

To synchronize the parallel computation we keep a Global Task Queue (GTQ) and, for each processor, $p = 1, \ldots, P$, a Local Task Queue (LTQ(p)). The root task is inserted into the GTQ. In the general step, a set of tasks that is generated by processor i will be inserted into LTQ(p) unless the GTQ becomes empty, in which case the tasks will be inserted into the GTQ. An idle processor q will fetch a new task from LTQ(q) first, and only use the GTQ if LTQ(q) is empty. In this way, contention for the shared resource, the GTQ, is kept small. A detailed implementation is discussed in [3].

After the computation phase we need to connect the line segments which are solutions of ITTs to form the intersection curve. We employ a variant of curve tracing using the neighbor following method. This method requires the algorithm to traverse up and down the quad tree to find the neighbor of a node. Thus it requires $O(nr)$ computation, where $r$ is the

height of the quad tree and $n$ is the number of the leaves where the curve passes through their correspooding sub-domain. (We 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 Figure 2.)

Instead of going up and down the quad tree to find a neighbor for a node each time, we keep a list of pointers which point to the vital nodes in the quad tree with the same level. At the beginning of the connecting phase this list contains the vital leaves in the quadtree. It then fills a new list, which contains those pointers that are parents of the leaves in the old list. For each parent node, it connects all the curve segments within the area that it corresponds to. Each time it 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. Therefore, connecting the intersection curve can be viewed as a procedure similar to a bottom-up merge sort.

Our solution has the same complexity $O(nr)$ as in the sequential neighbor following approach. In the parallel case, if there are $P$ processors available, then our solution may achieve nearly linear speedup. That is, the complexity is $O(nr/P)$.

**Vectorization**

In addition to parallelism, vectorization of a subdivision job is implemented to take advantage of vector pipeline features. For each surface, a subdivision process needs to compute the parametric function at five grid points in the parametric domain (referred to as grid evaluation) and compute four bounding boxes. We could vectorize them directly by implementing vector functions that compute grids and bounding boxes, but the improvement would not be significant. The reason is that the vector length is much too short to fully take advantage of vector pipeline features. To improve the performance of our Macro-subdivision method here, a strategy named as "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 quad tree it must also pass through at least one node of its children. This means that if a node is subdivided then later some children will have to be treated. The Lookahead subdivision algorithm precomputes grids and bounding boxes in the next subdivision levels 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 next descendent level. Similarly, there are 81 grids and 80 bounding boxes needed to be computed if it looks ahead into the next two descendent levels (see Figure 3).

The lookahead feature increases the computation of a subdivision task only slightly, but there will be significant savings since it 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.

(A): Lookahead tree of node P with level 2


5*5 = 25 grid points
4+16 = 20 bounding boxes

(B): Lookahead tree of node P with level 3


9*9 = 81 grid points
4+16+64 = 80 bounding boxes

Figure 3. Task in lookahead subdivision levels; curve passes at least one node of its children.

**Benchmarks**

We first implemented the Macro-subdivision algorithm (without vectorization) on a Sequent Balance 21000, which is an MIMD 24-processor machine. The benchmarks for these tests are summarized in Figure 4. The results indicate that speedup is especially high for up to 10 processors. These results suggested our approach would be well suited for a 4-processor Cray-2 architecture. Implementations on the Cray-2 were carried out at the National Supercomputing Application Center, University of Illinois at Champaign. To implement of the Macro-subdivision and Lookahead subdivision algorithms as portable as possible, we used the MONMACS package, which is the parallel programming library developed at Argonne [4]. On the Cray-2 we first implemented the pure Macro-subdivision algorithm, and then added the lookahead features for vectorization.

Figure 5 shows the performance comparison of Lookahead subdivision (level 2, 3) and Macro-subdivision with (and without) vectorization method for various tolerance inputs on the Cray-2. The actual test surfaces are shown in Figure 1.

The performance of Lookahead subdivision method is much better than

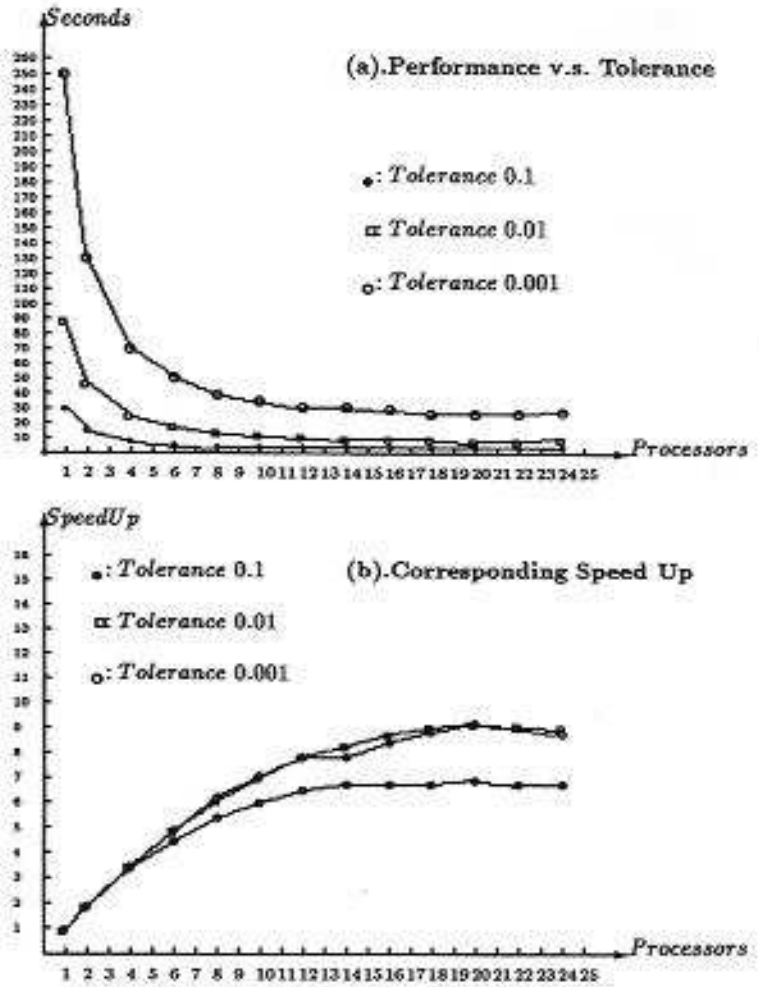


Figure 4. Performance of the Macro-subdivision method on the Sequent Balance 21000.

that of Macro-subdivision method with (and without) vectorization. The speedup of the Lookahead subdivision method can be up to 6 times that of the sequential algorithm on the Cray-2. We conclude that, given the underlying architectures, our implementations of Macro-subdivision and Lookahead subdivision algorithms are close to the expected theoretical bounds.

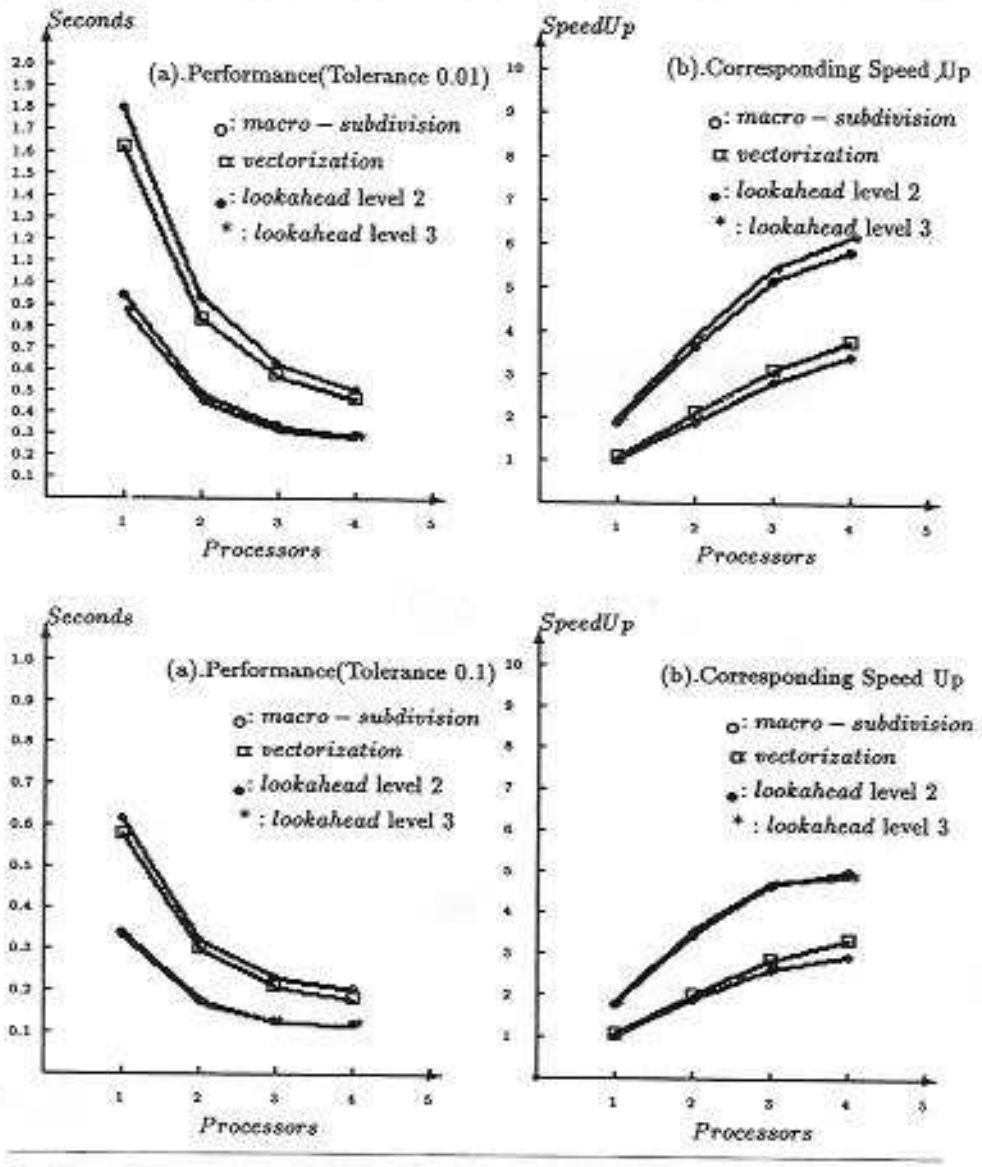


Figure 5. Comparison of Lookahead with Macro-subdivision on the Cray-2.

It is also interesting to note that one can increase the precision tolerance and make up for the loss of speed by using a proportional number of processors. This gives 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 speedup can be obtained, in actual implementations, there is an optimal number of

processors to achieve the most effective use of the available processors. For example from Figure 4, one can notice that the second half of processors does not contribute as effectively as the first half of processors. That is, the second half of processors should be used independently.
Since surface intersections are the "lowest level" computation in supporting Boolean set operation, it is recommended to find out the optimal number of processors so that the rest of processors can be used for independent tasks. This makes multi-level parallelism very useful in a higher level computation (such as the intersection of two objects) since we can divide the processors into several groups of processors and each group of processors works independently on different tasks (e.g., intersection of two surfaces) simultaneously.

**Acknowledgements**
We want to give our sincere thanks to Donald Peterson at Sandia National Laboratories for many useful discussions. We also thank Brian Smith for assisting us in making the necessary contacts at Argonne National Laboratories.

References

1
Pratt, M.J. and A.D. Geisow, *Surface/Surface Intersection Problems*, in: Gregory, J.A. (ed.), The Mathematics of Surfaces, Oxford University Press, 1986.

2
Filip, D., R. Magedson and R. Markot, *Surface Algorithms using Bounds on Derivatives*, Computer Aided Geometric Design 3, 295–311, 1986.

3
Chang, L.C., W.W. Bein and E. Angel, *Parallel Algorithms for Surface Intersection*, Technical Report CS90-7, The University of New Mexico, Department of Computer Science, 1990.

4
Lusk, E. and R. Overbeek, *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, 1987.