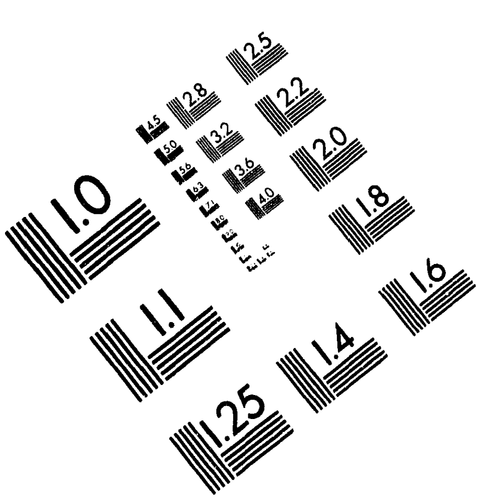
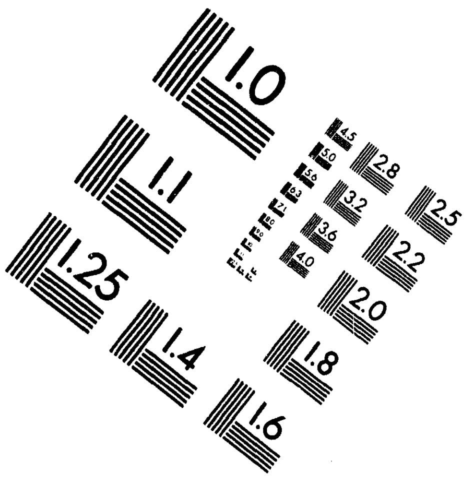
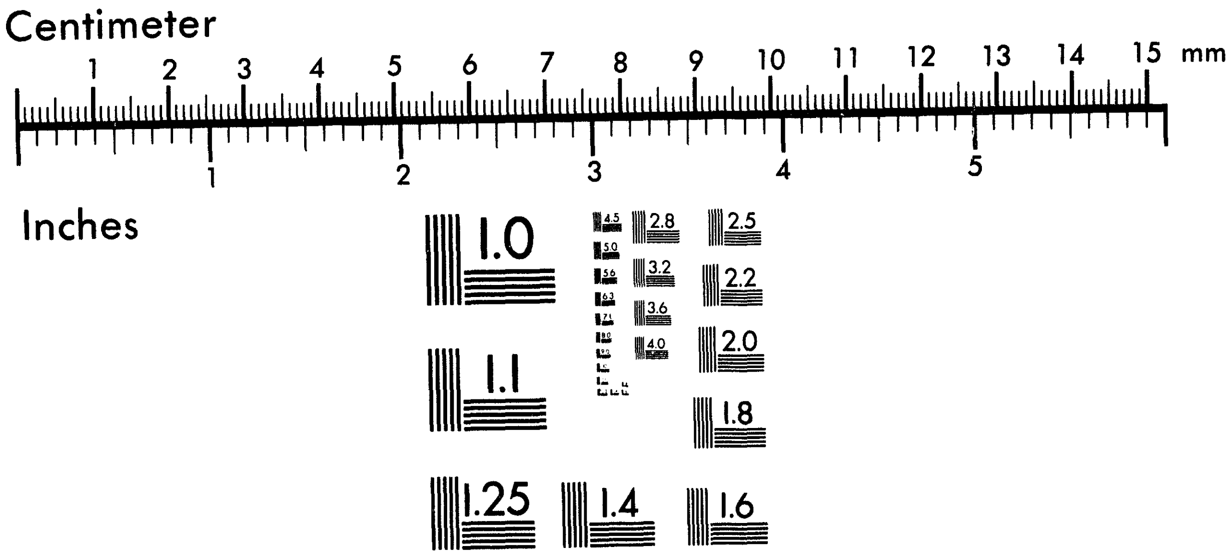
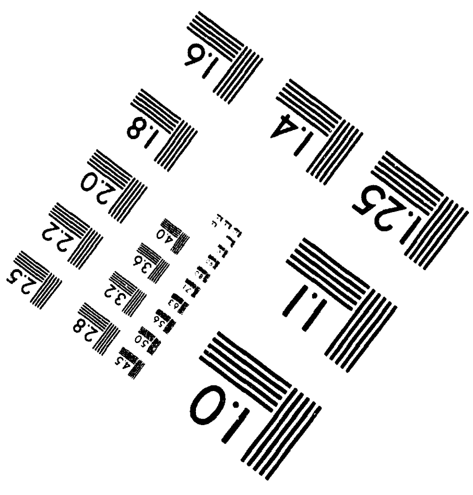
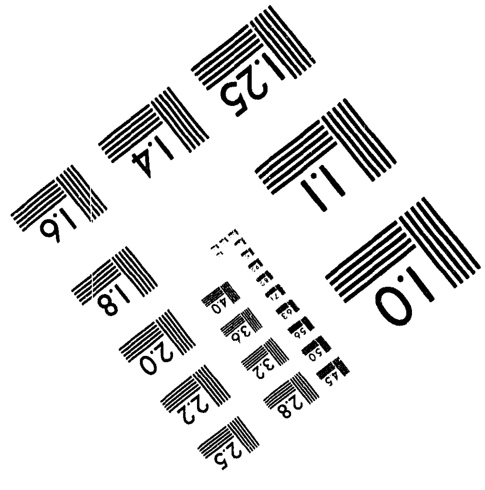




Centimeter
mm
Inches

SAND94-1299
Unlimited Release
Printed June 1994

Distribution
Category UC-705

LDRD Final Report: Fixture and Layout Planning for Reconfigurable Workcells

David Strip and Cynthia Phillips
Sandia National Laboratories
Albuquerque, NM 87185-0951

Abstract

This report summarizes the work performed under the LDRD "Fixture and Workcell Layout for Reconfigurable Workcells." This project spanned the period FY 91–93. The work in this program covered four primary areas: solid modeling, path planning, modular fixturing, and stability analysis. This report contains highlights of results from the program, references to published reports, and, in an appendix, a currently unpublished report which has been accepted for journal publication, but has yet to appear.

1 Program Impacts

Although this program has just completed, and there is follow on research required to bring many of the results to maturity, early results have had impact on programs in the Intelligent Systems and Robotics Center. Several projects within the DP and ER&WM programs are planning to incorporate some of our results, particularly those that relate to planning paths and workholding. These projects are funded to over $5M. As an outgrowth of this and related activities, we were successful in competing for a TTI funded CRADA (with GM) which will extend some of the results reported here. In addition, we have received some follow-on funding from the DP Exploratory Systems Programs office. Our motion planning work, developed in part under this program has a patent filing, and has been licensed to commercial firms for use in their products.

The details of most of the program have been reported elsewhere. In the balance of this report we provide general context and reference to these materials. In one area there are a number of issues that have not been previously reported, and these are contained in their entirety. In addition, one of our major papers has not yet appeared in print, although it has been accepted for publication. This paper is included as an appendix.

2 Solid Modeling

Geometric modeling underlies many of the challenging problems in our approach to fixture and layout planning for reconfigurable workcells. Because we are interested in using CAD models as our input, we are constrained to working with the raw geometry of the parts and workcell. Existing techniques for solid modeling are very time consuming, ruling out many approaches to the analysis phase of the problem. We have attacked this problem by developing an entirely new approach to solid modeling using a massively parallel processor (the Connection Machine). Although the algorithms we developed are rooted in more traditional approaches, they are, in fact, entirely new algorithms rather than ports of existing codes. We have coded and tested all the fundamental algorithms required for solid modeling. While we have yet to optimize the performance of this code, we have achieved speedups of 40 times or more the speed of far more costly supercomputers. Advances in the Connection Machine system software have made some fairly simple improvements now feasible. These improvements are expected to give speed improvements of 2-10, making the overall performance of the code more than two orders of magnitude faster than supercomputer implementations.

The results of this work have appeared in The International Journal of Supercomputing [9] and have been accepted for publication in the ACM Transactions on Graphics [6]. In addition, a more informal SIAM paper [10] has also been accepted. Because [6] represents a major exposition on this work, and it has yet to be scheduled for print, we have included it as an appendix to this report.

3 Motion Planning

In the earlier portion of this program we devoted a portion of our resources to addressing the issue of motion planning. In order to achieve flexibility in robotic workcells, it is necessary to automate a large fraction of the task teaching that is involved in developing new applications. Currently when a workcell layout is reconfigured, the robot has to be manually taught new "points" in order to be able to carry out its task. Our goal has been to automate this process. The key problem lies in the difficulty of finding collision-free paths among the parts, fixture, and assemblies in the environment. Our investigations in this area have lead to the development of the SANDROS path planner [3], a multi-resolution, hierarchical planner which is capable of solving motion planning problems of realistic complexity in a matter of minutes, versus hours for previous methods. A patent application is pending on this development. The code has been licensed to Deneb Robotics, which has incorporated it in the robot off-line programming system, and to Silma, which is evaluating it for the same purpose.

4 Modular Fixturing

In the latter part of the program we shifted our focus from the workcell layout issues to fixturing and related topics. In fixturing, we concentrated on using modular fixturing to hold workpieces. The advantages of modular fixturing are well established: easy re-use of components, rapid assembly of fixturing configurations, and life-cycle economics. The difficulty in using modular fixturing systems has been in designing the fixtures themselves. With the much more limited scope of fixture elements, and a more constrained mounting system, fixture design can often be a time-consuming, trial-and-error process. We have developed a technique for designing optimal fixtures from a geometric description of the part to be held. Although the algorithm is currently limited to planar components, extension to the full 3D components is planned. The benefit of the algorithm is that it allows arbirtary design metrics to be easily optimized over the set of possible designs. Unlike manual methods, in which the search is ad hoc, and fixture quality is unknown, our methods allow the fixturing kits to be used to their greatest benefit. This work is reported in [2].

5 Stability Testing

In this final section we discuss research conducted on the stability testing problem for objects in a gravitational field. This work was still in a preliminary stage at the conclusion of the project. It appears to be a promising avenue for future study, and we hope to find support for its continuation. The balance of this section is a detailed report of our work to date, as this material is not ready for general publication, and hence has not been recorded elsewhere. We begin by defining the problem and the goals of the study. We then discuss the state of the art, indicate errors in the seminal work in the field, give corrections where possible, and discuss promising directions for future work.

5.1 Problem Definition and Goals

Consider an industrial setting where a single robot arm is constructing an object. The robot places each piece in turn. It is critical that each intermediate step of the construction be stable under the force of gravity or the subassemblies will fall down and the task can never be completed. The goal of this research was to learn the state of the art in stability testing for sets of 2-dimensional polygons and 3-dimensional polytopes in a gravitational field and to contribute to the state of the art in efficiency and/or capabilities.

5.2 Previous Work

In this section we summarize the work of Palmer and Mattikalli et. al. This is not a complete survey, but indicates some of the most recent advances. Other extremely recent results are not included in this report at the authors' request.

The seminal work in the area of stability testing is Palmer's PhD thesis [8]. He considers the computational complexity of motion detection and stability testing of 2-dimensional polygons in the plane both with and without friction. More specifically, he considers the following problems and obtained the following results:

- Infinitesimal mobility: is there a set of instantaneous velocities for a configuration of polygons which corresponds to a feasible motion? A feasible motion is one where no polygon intersects another. In the infinitesimal context, this means that in places where two polygons meet, no point in the contact region is moving toward the interior of either polygon. Palmer shows this problem is NP-complete.

- Translational mobility: Is there a real feasible translational motion for a given configuration of polygons? A real motion has at least one pair of polygons changing relative positions and has at least one polygon moving a finite (not infinitesimally small) distance . This problem is shown to be NP-complete.
- General mobility: Is there a real feasible motion for a configuration of polygons? The NP-hardness result for translational motion applies since translation is a subclass of possible motions. There is no upper bound given.
- Infinitesimal instability: Is there an infinitesimal legal motion? This corresponds to potential instability. This problem is NP-complete both with and without friction. There appear to be errors in the proof of the upper bound for the case with friction (see below).
- Stability: is the configuration guaranteed to be stable? This problem is CO-NP-hard both with and without friction.
- Potential stability: Is it possible that the configuration is stable? Does there exist a set of forces that satisfy the conditions for stability? This problem can be formulated as a linear program and therefore is in P both with and without friction. This means that the complement of the problem, guaranteed instability, is also in P.

The NP-hardness results for infinitesimal mobility, translational mobility, and potential instability are clever and correct and provide insight into the computational complexity of the problems. The hardness results hold even for convex polygons. They should not discourage the practitioner, however. The result depends upon the existance of lots of polygons that are fixed in space and disjoint from the boundary. This would imply an assembly where there are a significant number of grippers (eg. half of all polygons in grippers). The hardness result also depends upon point contacts and ill-defined normals, as pointed out by Palmer and others [7]. These contacts can be eliminated in a natural way by assuming there is a tiny, but finite, edge of contact (for the 2D case, finite area for 3D). Palmer states that by disallowing such contact points, the infinitesimal stability problem for frictionless configurations can be solved with a linear program. Each such contact point gives rise to a constraint of the form $c_i \vee c_j$, where c_i and c_j are linear inequalities. If there are no **OR** connectives, the constraints form a regular LP, and if there are only a few such points (say k), one can enumerate over the 2^k possible linear programs formed by choosing either constraint c_i or constraint c_j to be true for each such contact point.

Mattikalli et. al. [7] consider the problem of stability testing for 3-dimensional frictionless bodies. They assume a finite area of contact to avoid the ill-defined normal problem so a linear program solves the infinitesimal stability problem (if an assembly is infinitesimally stable then it is stable). For an assembly to be stable, all feasible motion directions (those that do not result in interpenetration) must result in an increased gravitational potential energy (otherwise the system will lose gravitational potential energy resulting in kinetic energy, ie. gravity will cause motion).

Mattikalli et. al. go further than simple stability testing, however. They find a stable orientation of the assembly, if one exists. If none exists, they find the "most stable" orientation, which corresponds to the minimum decrease in gravitational potential (for fixed-norm velocities). It corresponds to a minimum motion. The algorithm in [7] searches for the most stable orientation using linear programming in the inner loop. The authors later discovered how to find the most stable orientation with a single linear program [1].

5.3 A Technical Comment on the work of Mattikalli et. al.

This section derives an equation used by Mattikalli et. al. to model non-interpenetration constraints. These equations are approximations that are true only infinitesimally.

Mattikalli et. al. define a virtual displacement of the ith body as $\delta p_i = (\delta r_i, \delta\theta_i)$, where $\delta r_i, \delta\theta_i \in \mathcal{R}^3$. The vector δr_i is the translational motion of the body and vector $\delta\theta_i$ represents a rotation of magnitude $|\delta\theta_i|$ about the center of mass. The direction of $\delta\theta_i$ indicates the axis of rotation. The motion $\delta p_m = (\delta r_m, \delta\theta_m)$ for body m produces a motion δa_m for a point a on m. If c_m denotes the position of the center of mass of body m in a global reference frame and ρ_m is the displacement from c_m to point a, then they claim that the displacement δa_m is given by the following equation: $\delta a_m = \delta r_m + (\delta\theta_m \times \rho_m)$. To see where this comes from, consider the motion illustrated in Figure 1. The translational portion δr_m adds on directly. We now consider a small pure rotation about the center of mass within the plane of the page (so $\delta\theta_m$ is directed out of the page). We rotate through an angle of size $|\delta\theta_m|$. For $|\delta\theta_m|$ sufficiently small, we have

$$\begin{aligned} \delta a_m &= |\rho_m| \tan |\delta\theta_m| \\ &\approx |\rho_m|\,|\delta\theta|, \end{aligned}$$

since for $\theta \rightarrow 0$ we have $\tan\theta \approx \theta$. The cross product $\delta\theta_m \times \rho_m$ has the correct direction. The two vectors are at right angles and the right hand rule gives a vector that is tangent to the circular motion (the δa_m direction shown in Figure 1). The magnitude is given by

$$\begin{aligned} |\delta\theta_m \times \rho_m| &= |\delta\theta_m|\,|\rho_m| \sin(\pi/2) \\ &= |\delta\theta_m|\,|\rho_m| \\ &\approx |\delta a_m| \end{aligned}$$

5.4 Errors in Palmer's thesis with corrections

This section serves as a guide for others who want to study the computational complexity of polygon motion and stability using Palmer's thesis. Although errors/corrections will stand alone wherever possible, the reader will sometimes need a copy of the thesis. Errors are addressed in the order of their appearance in the thesis. There are many typographical and labeling errors, some of which change the meaning of sentences. A reader who is on guard will notice most of these. Such errors are addressed here only if they took a substantial effort to correct.

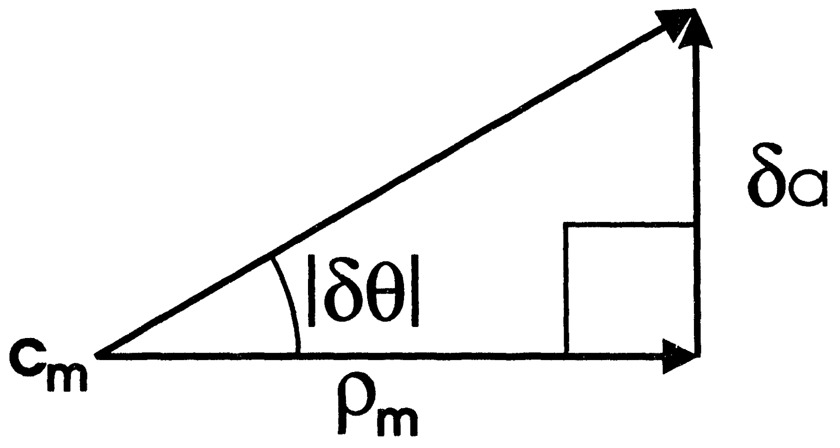


Figure 1: The motion of a point on a body can be derived from the motion of the total body. Here we assume the motion is a pure rotation in the plane of the page.

- The proof of lemma 3.1 (page 15) seems incorrect. The lemma states that any velocity with both translational and rotational elements can be converted to an equivalent motion that is purely rotational about an appropriate point in the plane. Velocities are represented as (x, y, ω) where x is translational motion in the x direction, y is translational motion in the y direction, and ω is a rotation. This motion is for a polygon's frame of reference (translation of the center of mass and rotation about the center of mass).

  Palmer first computes a point with zero velocity. Equation 2.11 gives the following value for the velocity of a point with displacement (x_p, y_p) within a polygon's reference frame:

  $$v_p = (x'(t_0) - y_p\theta'(t_0), y'(t_0) + x_p\theta'(t_0)),$$

  where v_p is the velocity of the point, $x'(t_0)$ is the x translational velocity of the polygon at time t_0, $y'(t_0)$ is the y translational velocity of the polygon at time t_0, and $\theta'(t_0)$ is the rotational velocity. Equation 2.11 also has subscripts to indicate a polygon P_i. The equation assumes $\theta(t_0) = 0$ where $\theta(t_0)$ is the angle of the reference frame with respect to the global reference frame at the initial time t_0. This assumption is not explicitly stated in the lemma or its proof, but it should pose no problems.

  Representing velocity as (x, y, ω) gives $x'(t_0) = x$, $y'(t_0) = y$, and $\theta'(t_0) = \omega$. Substituting into equation 2.11 gives $v_p = (x - \omega y_p, y + \omega x_p)$. Setting each component of v_p equal to zero (to find a point of zero velocity) then yields $x_p = -y/\omega$ and $y_p = x/\omega$. These are the negations of the values given in the proof of the lemma.

Later in the proof, there are more algebraic errors. On page 16, after substituting $t = 0$ into equation

$$\hat{M}'(t) = (x_\nu(\sin(\omega t) - y_\nu \cos(\omega t))\omega, -y_\nu(\sin(\omega t) - x_\nu \cos(\omega t))\omega, \omega),$$

the result should be $\hat{M}'(0) = (-x_\nu y_\nu \omega, x_\nu y_\nu \omega, \omega)$, not $(-y_\nu \omega, -x_\nu \omega, \omega)$ that is claimed. Of course, substituting in the correct values of x_ν and y_ν (the values of x_p and y_p computed above) clearly does not give back (x, y, ω) as required. These algebraic errors may be correctable.

- Theorem 3.3 uses Lemma 3.1 to prove that checking the endpoints of edges of contact suffices to verify the whole edge. The proof is not convincing because too many pieces are explained informally (eg. there is no discussion of what the pure rotation means for the edge in question. Perhaps more illustrations would help), and calculations are passed off as "elementary calculus" or "straightforward". It is plausible that a more convincing and simpler proof could be built from the fact that the position of each point on the edge is a convex combination of the positions of the endpoints at all times.

- On page 22, the reference to equation A.11 should read 2.11. The variable x_i means the x velocity of polygon P_i, w_j is the rotational velocity of polygon P_j and so on. There's a blatant typographical error near the bottom of the page, where $(n - n_2)\cdot \leq< 0$ should read $(n - n_2) \cdot \epsilon \geq 0$.

- On page 23, there is the following claim: "Since any real feasible motion has an infinitesimally feasible velocity, this [the result that infinitesimal mobility is in NP] shows that the complement of the mobility problem is in NP, and thus that the mobility problem is in CO-NP". There are two problems with this statement. First, Palmer distinguishes between infinitesimal mobility and general mobility where there is a real feasible motion. Therefore, there must be cases where there exists an infinitesimal motion, but no real motion (all examples we've seen, including the one on page 63, are somewhat artificial and depend upon point contacts). Thus the "no" cases for infinitesimal mobility do not cover all the "no" cases for general mobility, and complexity results for the former problem cannot be directly applied to the latter. The second problem is that infinitesimal mobility is shown to be in NP, but that does not imply that the complement is.

- Given that a thesis on computational complexity will be read by computer scientists as well as physicists, a more intuitive description of a conservative field (p. 40) may be more appropriate. For example, an equally valid definition is that a force is conservative if the work done in moving a particle between two points depends only upon the two points and not upon the path taken.

- Part of the proof of lemma 4.1 is incorrect. Lemma 4.1 states that reactive forces can be considered to act only at points of node contact. The proof replaces a force f_p acting in the middle of an edge by a pair of forces (one at each endpoint) that have the same effect. More precisely, $f_1 = [l_2/(l_1 + l_2)]f_p$ and $f_2 = [l_1/(l_1 + l_2)]f_p$, where force f_1 is applied at endpoint

n_1 a distance l_1 away from the original point of application (similarly for f_2). The proof of no introduced torque is incorrect. The following proof shows that the torque about point p in Figure 4.1 is zero (as it was originally). The torque from force f_1 has magnitude $l_1 \, |f_1| \sin\theta$ where l_1 is the distance from point p to point n_1 where the force is applied, and θ is the angle each of the parallel forces makes with edge e. The magnitude of the torque from force f_2 is $l_2 \, |f_2| \sin(180 - \theta)$ and the direction is exactly opposite the direction of the torque from force f_1. Therefore the resultant torque has magnitude

$$\begin{aligned} l_1 \, |f_1| \sin\theta - l_2 \, |f_2| \sin\theta &= \frac{l_1 l_2 \, |f_p|}{l_1 + l_2} \sin\theta - \frac{l_1 l_2 \, |f_p|}{l_1 + l_2} \sin\theta \\ &= 0 \end{aligned}$$

- Theorem 4.2 (Virtual work) is stated without proof because "any introductory text in physics or mechanical engineering contains a development of these methods". Unfortunately, it can require substantial effort to locate a good discussion of virtual work if one does not have the specific reference cited in the theorem. An alternative text is Fanger [4]. To provide a little intuition, a negative instantaneous work (power) within a gravitational field corresponds to the case where the velocities are going uphill or the force of friction resisting motion is more than enough to counteract gravity (and is counted as full strength in the calculations). Fanger does not provide a proof of the theorem.

- In the equations on page 45 (and later in chapter 4), it appears that MIcm(i) is used to denote the moment of inertia about the center of mass for polygon i, though in chapter 2 when notation was defined, that quantity was to be represented as Icm(i). The reader could then mistakenly believe that the mass itself appears in the equations, which would make no sense. The equations for rotational work are not directly derived in Haliday and Resnick[5], the primary physics reference. In that text angular work is derived as an integral over θ, not time t.

- The fourth type of constraint in the LP formulation of frictionless potential stability appears wrong. These constraints imply that if vector a is the sum of two vectors $a = b + c$, then $-a = d + e$ where vector d is perpendicular to vector b and vector e is perpendicular to vector c. If this is true it needs a proof. The constraint can be formed correctly, however, by using the original formulation $f_{n_1} = c_1 N(e_i) + c_2 N(e_{i+1})$. The normals are known and the c_j are unknown. This has the disadvantage that there are now variables that do not correspond to a force or velocity.

- On page 56, the formulation of stability with friction as an instance of the constrained max scalar product problem appears incorrect. The length-$2m$ vectors V_1 and V_2 are defined such that components $1, \ldots, m$ are zero in vector V_2 and components $m + 1, \ldots, 2m$ are zero in vector V_1. Thus, no matter how values are assigned to the forces and velocities, the scalar product of V_1 and V_2 will be zero.

- The proof of lemma A.5 is incorrect. This lemma states that the constrained maximum scalar product problem is in NP. The lemma is true, but requires a different proof. The constrained maximum scalar product problem is defined as follows (p. 73-74): Given (V_1, V_2, F), where

  1. V_1 and V_2 are equal sized vectors of variables,
  2. F is a formula consisting of the connectives **AND** and **OR** and base terms c_i where each c_i is a linear inequality in the variables of $V = V_1 \cup V_2$ with rational coefficients,

  the solution is a pair of vectors $X_1 = (x_{11}, \ldots, x_{1k})$ and $X_2 = (x_{21}, \ldots, x_{2k})$ where $X_1 X_2 = (x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2k})$ is an assignment of rational numbers to the variables of V that satisfies F and maximizes $X_1 \cdot X_2$. It's a laborous way of saying assign values to two vectors so as to maximize their scalar product subject to contraints on the components of the vectors that are linear inequalities with and's and or's.

  In the proof given in the thesis, first the constraints to be satisfied are guessed. It is claimed that this yields a $2k$-dimensional convex space $S = S_1 \cup S_2$ where S_1 has the last k components all zero and S_2 has the first k components all zero. Space S is not convex since any nontrivial convex combination $(\alpha s_1 + ((1-\alpha)s_2)$ for $0 < \alpha < 1$, $s_1 \in S_1$ and $s_2 \in S_2)$ has nonzero terms in all components. The convexity is not critical, so this is not the main problem with the proof. The proof then claims that any optimal solution will be at extreme points of both the S_1 space and the S_2 space. Both the form of the space S and the claim that optima are simultaneously extreme points of S_1 and S_2 require that no constraint involves variables from V_1 and V_2 simultaneously. Otherwise, fixing the variables in vector V_1 gives a new polytope that may have different extreme points for V_2 than the original (for example if both vectors are length 1, the original polytope defined by choosing inequalities to be satisfied gives a convex 2D polygon. Fixing one variable corresponds to picking a line. If the line does not pass through a vertex of the polygon, then the restricted extreme point for the second variable is not the same as the original. It is possible that redefining the problem to disallow such constraints may allow this proof to go through.

  Here is a proof of the fully general problem as defined (allowing interacting constraints). First we guess the constraints to be satisfied and verify that this assignment of true/false to the clauses yields a satisfying assignment (*i.e.*, that the and/or structure of constraints F is satisfied). We then need to solve the problem of maximizing $X_1 \cdot X_2$ subject to a set of linear constraints. This is an instance of quadratic programming that Vavasis[11] proved is in NP.

5.5 Directions for Future Research

There are several potential directions for future research. One possibility is to improve the efficiency of what is currently done. The stability test must be solved at each step of the assembly. The linear program for each successive step will be similar to the preceeding one. Furthermore, we know that each preceeding step was stable. This structure could provide insight that will speed up the processing at each stage.

Another, more useful, direction to pursue is to add friction to the current models. This is more difficult, but real systems have friction. It has been claimed that an assembly that relies upon friction for stability has a fragile form of stability, but with appropriate tolerances built in, this would provide many more potential assembly steps (including some that may be much easier to program a robot for).

A third direction of research is sensitivity analysis. If an assembly is stable, where is it's weakest point? What is the smallest force needed to topple the assembly and where should it be applied? This only makes sense in the case with friction. This analysis can help plan approaches for the robot hand (to avoid danger spots) or help to plan fixturing.

References

[1] David Baraff, Raju Mattikalli, Bruno Repetto, and Pradeep Khosla, "Finding stable orientations of assemblies with linear programming", Technical Report CMU-RI-93-13, Carnegie Mellon University, June 1993.

[2] Randy C. Brost and Kenneth Y. Goldberg, "A Complete Algorithm for Synthesizing Modular Fixtures for Polygonal Parts", *Proceedings of the IEEE International Conference on Robotics and Automation, 1994*, San Diego, CA, May, 1994.

[3] Pang C. Chen and Yong K. Hwang, "SANDROS: A Motion Planner with Performance Proportional to Task Difficulty," *Proceedings of the IEEE International Conference on Robotics and Automation, 1992,* Nice, France, 1992.

[4] Carleton G. Fanger, *Engineering Mechanics: Statics*, Charles E. Merrill Publishing Co., Columbus, OH, 1970.

[5] David Halliday and Robert Resnick, *Fundamentals of Physics*, Second Edition, John Wiley & Sons, New York, 1981.

[6] Michael S. Karasick and David R.Strip, "A Massively Parallel Algorithm for Intersecting Solids," ACM Transactions on Graphics, to appear.

[7] Raju Mattikalli, Pradeep Khosla, Bruno Repetto, and David Baraff, "Stability of assemblies", *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, July 26-30, 1993.

[8] Richard S. Palmer, "Computational complexity of motion and stability of polygons", PhD thesis, Technical Report TR 89-1004, Cornell University, May 1989.

[9] David R. Strip and Michael S. Karasick, "Solid Modeling on a Massively Parallel Processor," International Journal of Supercomputer Applications, V. 6, No. 2, (Summer, 1992), pp. 175–192. (Previously available as IBM Research Report RC 16568, Jan, 1991.)

[10] David R. Strip and Michael S. Karasick, "A SIMD Algorithm for Intersecting Three-dimensional Polyhedra," SIAM News, to appear.

[11] Stephen Vavasis, "Quadratic programming is in NP", Information Processing Letters, 36 (1990), 73-77.

Appendix A: Intersecting Solids on a Massively Parallel Processor

Michael Karasick
Manufacturing Research Department
IBM Research Division
Yorktown Heights, N. Y.

David Strip *
Sandia National Laboratories
Albuquerque, N. M.

June 21, 1993

Abstract

Solid modelling underlies many technologies that are key to modern manufacturing. These range from CAD systems to robot simulators, from finite element analysis to integrated circuit process modelling. The accuracy, and hence the utility, of these models is often constrained by the amount of computer time required to perform the desired operations. In this paper we present, in detail, an efficient algorithm for parallel intersection of solids using the Connection Machine, a massively parallel SIMD processor. We describe the data-structure for representing solid models and detail the intersection algorithm, giving special attention to implementation issues. We provide performance results, comparing the parallel algorithm to a serial intersection algorithm.

1 Introduction

Solid modelling is the representation of and reasoning about physical objects. It underlies many aspects of computer aided design and analysis, physical process simulation, robotics, computer vision, and other tasks. Computational experiments using analogues of real processes applied to computer representations of physical objects reduce the need for costly and time-consuming experimentation and prototyping of preliminary designs. These computational experiments often require substantial amounts of computer time. Reducing this time allows more design alternatives to be examined in greater detail. This results in better products faster. This paper describes techniques that enable the use of massively parallel computers to provide faster solid modelling procedures.

A great deal of theoretical and practical work has been done in solid modelling. The Production Automation Project at the University of Rochester produced much of the foundation of contemporary solid modelling [12, 14]. Solids are most often represented using three fundamentally different schemes:

Constructive Solid Geometry (CSG) describes a solid as an algebraic combination of primitive solids using set operations;

Volumetric decomposition describes a solid as a disjoint union of simple solids.

*This work was supported by the U. S. Department of Energy under Contract DE-AC04-76DP00789.

Boundary representation describes a solid by enumerating the zero-, one-, and two-dimensional boundary sets.

Early, but still comprehensive, surveys of solid representations and solid modelling systems that use them are found in [13, 15].

Set operations are of fundamental importance for all three kinds of representation because it is often convenient to create the representation of complex solids using set operations on simpler solids. Furthermore, many queries of solids can be easily phrased using set operations. (For example, interference detection is formulated as an intersection problem.) For these reasons, algorithms for set operations and their implementations have received a great deal of attention [9, 16, 5, 18, 17]. Independent of the basic representation, all known set-operation algorithms require boundary representations (or implicit computation of the boundary). These set-operation algorithms require large amounts of computation time for two reasons. The algorithms are inherently complex and need to deal with many special cases; this alone leads to long computation times. In addition, the computation time is necessarily a function of the size of the result, and interesting models tend to be large.

Parallel computation is often seen as a panacea for time-consuming computations since it offers a way to increase the amount of computation power available as problem sizes increase. Special-purpose parallel computers have been built for modelling-related applications in computer graphics (*e.g.*, [3]), but have not been shown to be useful for solid modelling applications. These computers have been designed primarily to assist in the rendering of computer graphics images and not for the manipulation of the underlying representation of the geometric objects. A specialized parallel architecture for manipulating CSG representations has been built in prototype versions [7]. This approach shows promise for certain modelling applications but not enough experience is available yet to form a judgement. Recently a parallel architecture and related algorithms for solid modelling have been proposed [10]. The technique described uses load-balancing to distribute the work over a small number of very powerful processors and so does not appear to be scalable with problem size. Another algorithm for polyhedral intersection using load balancing is presented in [11]. This algorithm is also restricted to polyhedra with manifold boundaries.

Theoretical results in computational geometry have attempted to address some of the simpler geometric problems in solid modelling [2, 1, 4]. Singular configurations are frequently ignored in these treatments, yet must be addressed in practical applications. (Edge-edge and edge-vertex intersections are examples of singular configurations. In general, singular configurations are those cases in which two solids intersect in such a way that small perturbations in location changes the topology of the boundary of the intersection solid.) More limiting are the models of parallel computation used to derive these results — they rarely correspond to physically realizable architectures.

In an earlier paper [19] we presented an overview of the techniques we developed for parallel solid modelling, including very efficient algorithms for generating primitives, and for rigid transformations. By utilizing DeMorgan's Law, we need only implement two of the four set operations. We chose intersection and complementation. Our parallel representation allows very efficient and simple implementation of complementation, which was also covered in [19].

In this paper we present the data-structure for representing solids and discuss in detail the intersection algorithm. We explicitly address the issues of algorithmic complexity, singular configurations, and realistic machine models. We have implemented this algorithm on Thinking Machines Incorporated's Connection Machine-2 in *lisp (pronounced "star-lisp") and discuss implementation issues as well as the general design of the parallel algorithm. We also present results comparing performance with serial algorithms.

In the next section we define what we mean by solids, how solids are represented on computers by their boundaries, and how we store the representation on the Connection Machine. We follow with a section describing the Connection Machine architecture and certain details regarding our programming of the machine. The remaining sections describe the intersection algorithm, with each section corresponding to a major section of the code. We begin with the mainline routine, which gives an overview of the algorithm, then proceed through the major phases of constructing the intersection solid. After the algorithm is presented we provide performance comparisons between our parallel code and a serial algorithm on a supercomputer.

We have adopted a number of stylistic conventions to (we hope) simplify the reading of this paper. Rather than dwell on identifying which processor carries out an operation, we will refer to a piece of data performing the operation. For example, we might describe an operation in which an edge calculates its tangent. This of course really means that the processor containing the data for the edge performs the computation. Unless it is ambiguous as to which processor is meant, we will use this smoother style of description. Because this paper is intended to give an in-depth description of a fairly complex algorithm, we have adopted a technique used by Knuth [8] to clearly identify sections that might well be skipped on a first reading. Paragraphs marked by the dangerous bend sign

contain material that describes some subtle aspect of the algorithm which, while important to implementation, breaks up the flow in understanding the overall algorithm. These sections may be safely skipped on a first (or even second) reading. Paragraphs marked by the double dangerous bend

are somewhat more obscure than those marked with a single symbol. What was said for the dangerous bend paragraphs goes double here. Finally there are paragraphs marked by a little Connection Machine symbol

which contain material that is specific to implementation on the Connection Machine, or a very similar architecture. These may be skipped if you are interested only in the more general problem

2 Defining Solids

A physical object is represented by a "solid," an object whose boundary consists of planar pieces. (Physical objects whose boundaries have curved surfaces can be approximated using planar facets.) A solid can be described by the planar pieces, called *faces*, the line segments where faces meet, called *edges*, and the points where edges meet, called *vertices*. The descriptions of complex solids are often formed by combining simpler solids using operations such as union, intersection, and difference.

A boundary representation (B-rep) is an organized enumeration of the vertices, edges, and faces of a solid. Practical computations on solids operate on the b-reps of the solids. A *face* f of solid S is a two-dimensional set on the surface of S. Disconnected faces are allowed as shown in Figure 1. The interior of the solid, $Interior(S)$, is below f. More precisely, if f is contained in plane $P(x, y, z) = ax + by + cz + d = 0$, then $Below$ is the half-space defined by $\{(x, y, z) : P(x, y, z) \leq 0\}$, and for every neighborhood $Nbhd$ of every point of f, $Nbhd \cap Below \cap Interior(S)$ is non-empty. Plane P is called the oriented plane of f, and the vector $N = (a, b, c)$ is called the (outward) normal to f. Note that if Q is the plane coincident with, but oriented opposite to P then the face contained in Q is distinct from f. The *edges* of a solid are the line

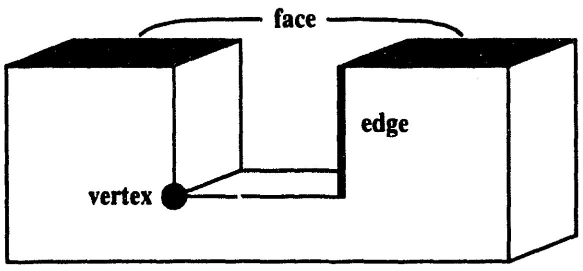


Figure 1: Vertex, edge, and face of a solid.

segments defined by the intersection of two or more faces, and the *vertices* of a solid are the points defined by the intersection of three or more faces. (See Figure 1.)

The relationships among the vertices, edges and faces are called adjacencies. Given a face f, $BoundaryOf(f)$ is a set of edges and vertices; given an edge e, $Bounding(e)$ is a set of faces. Solid modelling operations require efficient access to all the boundary elements adjacent to a given boundary element. For this reason, serial implementations of boundary representations generally use (complicated) multi-linked data structures that provide access while insuring consistency and reducing memory usage.

The Star-Edge data structure [6] is typical of such serial boundary representations. An edge e with initial vertex u and terminal vertex v can bound many faces. For each adjacent face f, it is necessary to describe whether the interior of f is to the right or left of e. This is implemented using *directed edges* e_f^+ and e_f^-. A directed edge, or *d-edge* (rhymes with hedge) is the fundamental entity of the Star-Edge representation. D-edge e_f^+ exists if the interior of f is to the right of e, e_f^- exists if to the left. (We say that d-edge e_f^+ is *positively oriented* and e_f^- is *negatively oriented.*) More precisely, if N is the outward normal of f, then e_f^+ exists if $(v - u) \times N$ points from the interior of e into the interior of f. Similarly, e_f^- exists if $(u - v) \times N$ points from the interior of e into the interior of f. Directed edge e_f^+ has *InitialVertex* u and *TerminalVertex* v. Directed edge e_f^- has InitialVertex v and TerminalVertex u. The *Tangent* vector of a d-edge is $(TerminalVertex - InitialVertex)$ The *FaceDirection* vector of a d-edge is $Tangent \times N$. We denote a d-edge with unspecified orientation as $e_f^\bullet$; the subscript is omitted when the face is unspecified. Edge e is referred to as the *underlying edge* of all the $e_f^\bullet$.

The d-edges of an edge e are ordered by radially ranking their face direction vectors on a plane perpendicular to e. (See Figure 2a.) Likewise, the d-edges incident to a vertex v and contained in a face f are ordered by radially ranking their tangents in the plane of f. (See Figure 2b.) If the interior of the solid lies between two adjacent faces, we refer to the two faces as a *volume-enclosing* pair. (If there are only two faces in the entire radial ordering, we need to specify which face is "first" and the direction of the ordering to determine whether a pair encloses volume.) We also refer to two adjacent d-edges of a face as volume enclosing if the interior of the face lies between the two d-edges. (The added complication of these orderings can be omitted if the domain of the representation is restricted to manifold solids. This restriction is common in modellers, but severely constrains the use of the modeller, particularly in the intermediate steps that are encountered when building a complex solid through set operations on more basic solids.)

The key adjacency relationships of the Star-Edge representation are shown in Figure 3. The Star-Edge representation is conceptually straightforward, yet its realization as a data structure is complicated and difficult to implement and maintain efficiently.

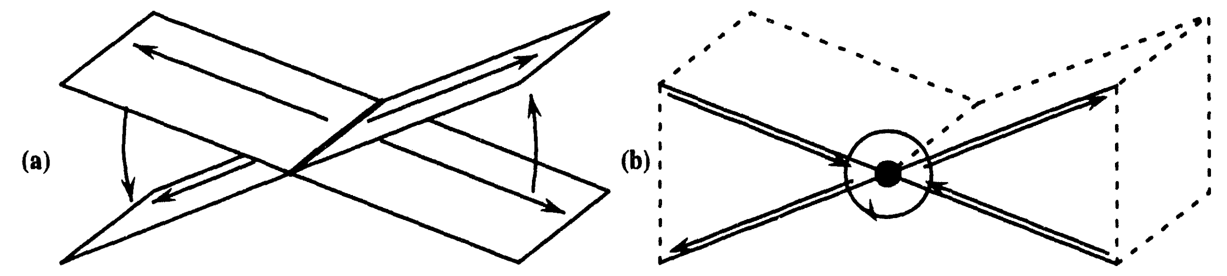


Figure 2: (a) Radial ordering of faces around edges; (b) Radial ordering of edges around vertex.

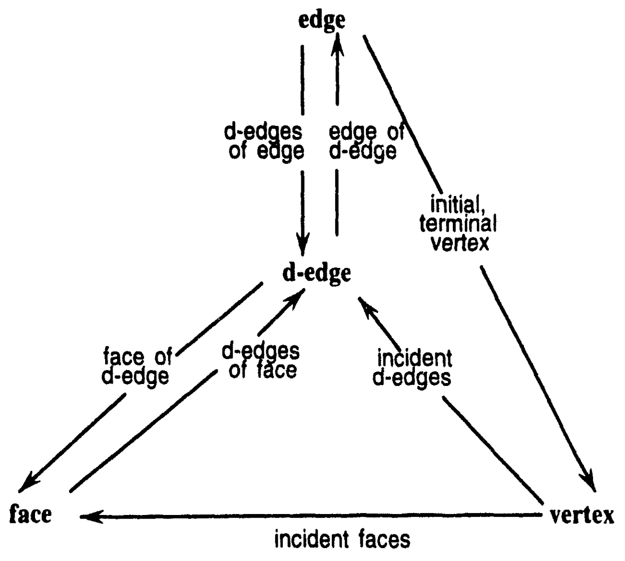


Figure 3: Adjacency relationships in the Star-Edge boundary representation.

3 The Connection Machine

The data-structure and algorithms we describe in this paper were designed for, and implemented on, the Connection Machine (commonly referred to as the CM), a massively parallel processor designed and built by Thinking Machines, Incorporated. The CM consists of up to 65,536 bit-serial processors interconnected with a hypercube communication network. Each processor has its own memory. The CM is a single instruction stream, multiple data stream (SIMD) architecture; a front-end processor (in our case a Symbolics Lisp Machine or Sun 3 or 4) executes a program and broadcasts instructions that the CM processors execute simultaneously. Conditional execution is achieved by maintaining a context flag in each CM processor. Depending on the state of that flag, the individual processor either carries out the broadcast instruction, or does nothing. Processors whose context flag enables execution are referred to as the *active set*. The CM contains a powerful communication facility allowing communication in either regular patterns (along the arcs of the hypercube) or general communication (arbitrary association of processors).

For problems for which 65K processors are not enough, the Connection Machine provides a *virtual processor* facility. The memory of a physical processor is divided into two or more segments, one segment for each virtual processor. The system software then automatically executes each instruction multiple times, once for each virtual processor of the physical processor. Since the code on the front-end processor is executed only once, regardless of the number of virtual processors per physical processor (known as the *virtual-processor (VP) ratio*), and for some technical reasons relating to the internal pipelining of instructions, the time to execute grows sublinearly in the VP-ratio. Asymptotically, the growth becomes linear.

In our analyses of complexity, results will be given in terms of times for machines with an infinite number of physical processors. In practice this means that the execution time of an actual problem on a machine with a bounded number of processors grows as a step function corresponding to the multiplier for the next vp-ratio, which is (currently) constrained to be a power of two. For our purposes we will greatly simplify the treatment of communications time as a measure of complexity. The Connection Machine communications can be fit into a fairly simple hierarchy. The most efficient communications are those along the "natural" edges of the embedding hypercube. We call these "nearest-neighbor" communications. Next in complexity are general communications of a 1:1 nature (a "collision-free" communication in Connection Machine parlance). Finally we have many-to-1 and 1-to-many communications. We have taken great pains to organize data in the machine to facilitate use **of the method** possible for the required operation. The first-class of communications is essentially constant time, regardless of machine size. Because of the manner in which the virtual processor system is constructed, time growth is substantially sub-linear in vp-ratio. General permutation operations require $log_d n$ time, where d is the dimension of the hypercube. The most general communications have a more complex time behavior, but typically behave in the same manner as the permutations, just with a larger constant. Hence, for the purposes of algorithmic complexity, we treat communications as being constant time operations, but respect the differences in placement in the hierarchy.

Because the Connection Machine has a SIMD architecture, programming is facilitated through a data model in which each processor has its own copy of a variable, each copy of which may have a different value. These are called parallel variables, or *pvars*. A pvar may be of any data type, such as float or integer, or may be a data structure. *lisp programming convention is to suffix a variable name with !! to denote a pvar. *lisp functions that return pvar values are also suffixed with !!, while those that return scalar values, or no values, are prefixed with a *.

Communication on the Connection Machine is handled using the ***pset**, **pref!!**, **news!!**, and ***news** operators. ***pset** and **pref!!** allow communications between arbitrary nodes in the network, while the **news!!** and ***news** are for regular communications along the edges of the network hypercube. **pref!!** and **news!!** read data from another processor, while ***news** and ***pset** place data in another processor. Because the ***pset** and **pref!!** allow arbitrary communications patterns, there is the possibility of "collisions" in reading or writing the data. The router software handles these collisions automatically. However, if a given communication can be guaranteed to be collision-free, the router can operate more efficiently. Hence there is substantial value to arranging "collision-free" communications. The efficiency of the ***pset** and **pref!!** operators is not the same; ***pset** is much more efficient that **pref!!**. ('tis better to give than receive.) Complicating matters, the reverse is true for the structured nearest neighbor operators **news!!** and ***news**. The upshot is that it pays to structure the data in a manner that insures the communications can be carried out efficiently. Hence, there are times when it may appear more natural to read from another processor, but through a little effort and planning a write can be used instead.

Pseudo-code descriptions of *lisp procedures given in the remainder of this paper will simplify the somewhat painful *lisp syntax. Rather than write

```
(*pset :no-collisions source!! destination!! address!!),
```

which means "send the value of the pvar called *source*!! to the pvar called *destination*!! at processor numbered by *address*!!, with no collisions expected", we will write

***pset** *source*!! **to** *destination*!! **at** *address*!! {**noCollisions**} .

Instead of writing

```
(*set destination!! (pref!! source!! address!!
        :collision-mode :collisions-allowed)),
```

which means "read pvar *source*!! from processor numbered by *address*!!, with collisions expected, and store the result into pvar *destination*!!", we will write

destination!! = **pref!!** *source*!! **at** *address*!! {**collisionsAllowed**} .

*lisp contains a **scan!!** which is analogous to the reduce operator in Common-Lisp or APL. Instead of operating on a vector or sequence, though, **scan!!** operates on the instances of a pvar distributed across the machine. For example, if *a*!! = 0, 1, 2, ... on processors 0, 1, 2, ... respectively, then **scan!!** applied to *a*!! with operator +!! results in processor *i* containing the sum of the integers from zero to *i*. *lisp also supports a *segmented-scan* capability. The scan can be broken into multiple segments, with each segment operating independently of the others. That is, data does not flow into or out of a segment. Thus in the **scan!!** +!! example, if we were to put a scan-segment boundary at 5, the value of the **scan!!** on processor 4 would be 1+2+3+4=10, while it would be 5 on 5, 5+6=11 on 6, and so on.

We often wish to rank many independent clusters of data. We would like to do this simultaneously for reasons of efficiency. We achieve this goal by prefixing the rank key with a cluster identifier.

Several parts of the algorithm deal with radial orderings, *e.g.*, the d-edges of a face *f* incident to a vertex *v* are radially sorted around *v* on the plane of *f*. We will often want to scan information around the ordering. We do this by using a scan to copy the processor number of the lowest ranked object to the highest ranked one. Then we use a collision-free ***pset** to send the data of interest from the highest ranked object to the lowest one, completing the circularity. Finally, another scan propogates the data values.

Just as we segment scans, we can segment circular scans to do many independent, circular scans simultaneously.

Many steps in our algorithm rely on ranking the data according to some pvar. The *lisp **rank!!** operator provides direct support for ranking a pvar across all active processors. The **rank!!** operator returns a value from 0 to (number of active processors) based upon a key of arbitrary, but specified length. We extended the language to include multiple key ranks by the simple expedient of concatenating the bit fields together. We also created extensions that return tie values based on an ϵ criterion for floating point keys. These extensions are implemented via macros, not through actual extensions in the underlying CM assembly language. Once ranked, we frequently reassign data to different processors to allow nearest neighbor (news) communication among consecutively ranked items, or to permit **scan!!** operations that occur in the rank order.

Although we describe our algorithm in terms of Connection Machine idioms, any massively parallel SIMD machine with a virtual processor facility, nearest neighbor communication, and independent processor context-selection flags can be programmed to perform this algorithm.

4 A Parallel Data Structure for Representing and Manipulating Solids

Efficient parallelization of algorithms depends on providing, at as low a cost as possible, each processor with the data needed to carry out the computations assigned to it. Unlike serial machines on which all data in "memory" has the same access cost, parallel machines can incur a large penalty for accessing data in another processor's memory. Therefore, data structures that are efficient for algorithms on serial machines may turn out to be very inefficient for parallel ones. The traditional linked data structures that represent solids on serial machines are a case in point.

Massively parallel architectures like the Connection Machine achieve maximum performance on large sets of independent, identical computations. This uniformity can be attained by homogeneous data representation. Rather than describing vertices, edges, and faces with linked data structures, we use a single distributed data structure, called a *parallel d-edge.*

The fields of a parallel d-edge are:

- InitialVertex
  - Label
  - Coordinates
  - Successor
- TerminalVertex
  - Label
  - Coordinates
  - Successor
- Edge
  - Label
  - Successor
- Face
  - Label
  - Normal
  - Distance
- SolidLabel

For example, if the faces of the solid are labeled $0,\ldots,F$, then FaceLabel is simply the number assigned to the face of this d-edge. In the previous section we showed that d-edges are radially ordered around vertices and edges. Let the d-edges of a solid be labeled $0,\ldots,D$ and let $e^{\bullet}$ be a d-edge. Then EdgeSuccessor is the label of the radially counterclockwise d-edge around its edge e, InitialVertexSuccessor is the label of the d-edge of f that is radially counterclockwise around the initial vertex, and TerminalVertexSuccessor is the label of the d-edge of f that is radially counterclockwise around the terminal vertex. The equation of the plane of f is represented by FaceNormal, the unit outward normal, and FaceDistance, the signed distance of the plane from the origin. Each d-edge of a solid is represented as a parallel d-edge, with one parallel d-edge per processor. A d-edge's label corresponds to the name of the processor containing the d-edge. For example, a cube is defined by 12 edges and 24 d-edges, and so 24 parallel d-edges (hence 24 processors) are used to describe the cube.

This data structure allows us to easily associate a processor with each entity of the solid, providing a natural assignment of work among processors for certain tasks. For example, solid modelling computations often require other information about d-edges, such as d-edge tangents, face-direction vectors, and so on. Parallel d-edges allow such quantities to be computed without interprocessor communication. The tangent is computed from a parallel d-edge as

$$Tangent!! \equiv e^{\bullet}.TerminalVertexCoordinates!! - e^{\bullet}.InitialVertexCoordinates!!$$

Contrast this with the calculation for the serial representation which requires accessing a total of four instances of three different data structures:

```
$e = EdgeOf(e^{\bullet})$
**if** $e^{\bullet}$ is oriented with $e$ **then**
  $TerminalVertex = e.TerminalVertex$
  $InitialVertex = e.InitialVertex$
**else**
  $TerminalVertex = e.InitialVertex$
  $InitialVertex = e.TerminalVertex$
**end if**
$Tangent \equiv Coordinates(TerminalVertex) - Coordinates(InitialVertex)$
```

Solid modelling algorithms also require the selection of boundary elements that satisfy certain requirements. When calculating the intersection of two solids, for example, we need to select the d-edges of a given face f. In the serial algorithm this requires iterating through several data structures. In contrast, computation on a massively parallel processor is done by selecting a set of processors for a calculation and then performing the calculation on that set of *active* processors. For the example of the d-edges of a face, selection is accomplished by activating those processors whose FaceLabel matches the given face. Associative selection is an important property of parallel d-edges.

5 Canonically Labelling Boundary Elements for Efficient Computation

The particular assignment of parallel d-edges to processors impacts the efficiency of algorithms that use the representation. There are several different schemes for assignment. For example, the processors can be viewed as a spatial mesh — each processor represents a volume — and each vertex is assigned to the processor for the containing volume. This assignment only works well if a solid's vertices are uniformly spatially distributed. Alternatively, because each boundary element is labelled by an integer, we can simply assign the vertex, edge, and face labelled i to processor i. This assignment works well if processor i does not need to operate on more than one of its boundary elements simultaneously. For example, if we wish to enumerate the boundary elements containing point p then this assignment works well if at most one of the boundary elements labelled i contains p.

Each parallel d-edge contains the coordinates of two vertices, the line segment of an edge, and the plane of a face. Unless the edge labelled i is described by the parallel d-edge assigned to processor i, another data structure on processor i is required. (Similarly the vertex labelled j and the face labelled k should be described by the parallel d-edges assigned to processors j and k respectively.) Given an assignment of d-edges to processors, we need to label vertices, edges, and faces so that additional data structures are not needed. Furthermore, we would like to compute this boundary element labelling efficiently.

The intersection algorithm that motivates the parallel d-edge data structure requires that a boundary element labelling have the following properties:

Uniqueness Property No two distinct boundary elements of the same type have the same label. (This is the minimal requirement for a labelling).

Incidence Property Each boundary element is labelled by an incident d-edge.

Intersection Property Given A, a canonically labelled solid, and Ω, a point, line, or plane, no two distinct boundary elements of A with the same label have point intersections with Ω. (For example, a plane cannot transversely intersect an edge and a vertex with the same label.)

The importance of the uniqueness property is self-evident.

The incidence property insures that the data describing a boundary element labelled i is present in the parallel d-edge contained in processor i. This guarantees that no additional data structures are needed.

The intersection property allows a significant reduction in complexity and communications costs of parallel solid modelling procedures. Consider the problem of labelling the vertices of a solid C formed by intersecting solid A with solid B. Some of the vertices of C are vertices of A or B. The others arise from boundary intersections. Suppose that we need to label a vertex that is the point intersection of an edge of A with an edge of B. If the edge of A has m adjacent faces and the edge of B has n of them, then at least $n + m$ processors contain paralle' d-edges incident to the point. All of these processors can label their copy of the point identically using the label (α, β), where α is the label of the edge of A and β is the label of the edge of B. Similarly, any point intersection of a boundary element of A with a boundary element of B can be labelled analogously using the labels of the intersecting boundary elements. This computation can be done efficiently, with no interprocessor communication. The incidence property of the labelling guarantees that any other object of A labelled α is incident to edge α. A case analysis using the intersection property shows that only one point common to both solids is labelled (α, β).

There is a simple algorithm for computing the canonical labelling from an assignment of parallel d-edges to processors. First label each parallel d-edge by the processor to which it is assigned. Then

1. label face f with the largest label of a d-edge of f;

2. label edge e with the largest label of a d-edge of e; and

3. label vertex v with the largest label of a d-edge with TerminalVertex v. (One could also choose InitialVertex.)

The uniqueness and incidence properties of the canonical labelling follow directly. The intersection property holds because:

1. a plane, line, or point transversely intersects an edge iff it does not contain either endpoint;

2. a line or point transversely intersects a face iff it does not intersect an edge or vertex of the face; and

3. the boundary elements labelled i are incident to one another.

An example of the canonical labelling is shown in Figure 4.

The Connection Machine router provides a simple and efficient way to compute the canonical labelling. One of the collision modes provided by the *lisp ***pset** command is {**max**} – processor i receives the maximum of the data values sent to it. Using this mechanism, it is very easy to canonically label a solid. Given a solid with faces labelled $1 \ldots F$, edges $1 \ldots E$, and vertices labelled $1..V$, the following code computes the canonical face labelling:

InitialVertex.Label	15
InitialVertex.Coordinates	(1, 0, 0)
InitialVertex.Successor	12
TerminalVertex.Label	9
TerminalVertex.Successor	8
TerminalVertex.Coordinates	(0, 0, 0)
Edge.Label	15
SuccessorAroundEdge	15
Face.Label	12
Face.Normal	(0, 0, 1)
Face.Distance	0
SolidLabel	6

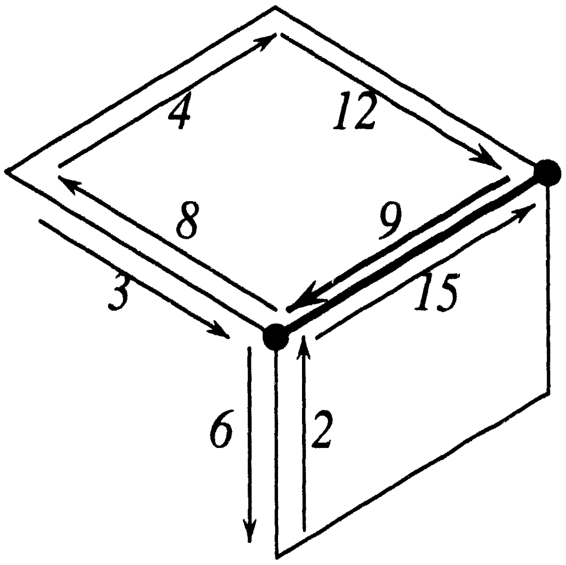


Figure 4: Canonical labels for boundary elements referenced in d-edge 9.

***pset** *ProcessorNumber*!! **to** *temp*!! **at** $e^{\bullet}$.*Face.Label*!! {**max**}
$e^{\bullet}$.*Face.Label*!! = **pref!!** *temp*!! **at** $e^{\bullet}$.*Face.Label*!! {**manyCollisions**}

The canonical edge and vertex labels can be similarly computed.

6 Overview of the Intersection Algorithm

Calculating the intersection of two solids represented by their boundaries is a complex task, requiring many special case tests to handle the various configurations that can arise. Thus any algorithm is, of necessity, complex. In the remainder of this paper we sketch our parallel intersection algorithm. We begin with an overall view of the major phases of the algorithm and then we provide a more detailed description. We hope that this approach facilitates both a general understanding of the overall algorithm and a feel for the details. Where appropriate, we compare or contrast our parallel approach to the more common serial techniques. We finish with some performance comparisons between our parallel algorithm and a serial algorithm on a supercomputer.

Figure 5 shows the intersection of two solids and the result of the intersection. The edges of the new solid can be divided into three groups; part (a) of the figure shows the edges that lie on the boundary of both input solids, (*i.e.*, they lie on the "intersection curve" of the two input solids); part (b) shows the edges that meet the intersection curve at one or two points; and part (c) shows the edges that lie on the boundary of one of the input solids and are completely contained in the interior of the other. The three major phases of the algorithm correspond to these three classes of edges. For each major phase there are numerous subcases, depending on how the edges and faces of the input solids meet. The challenge is to find a way to arrange the data and computations efficiently on a SIMD architecture.

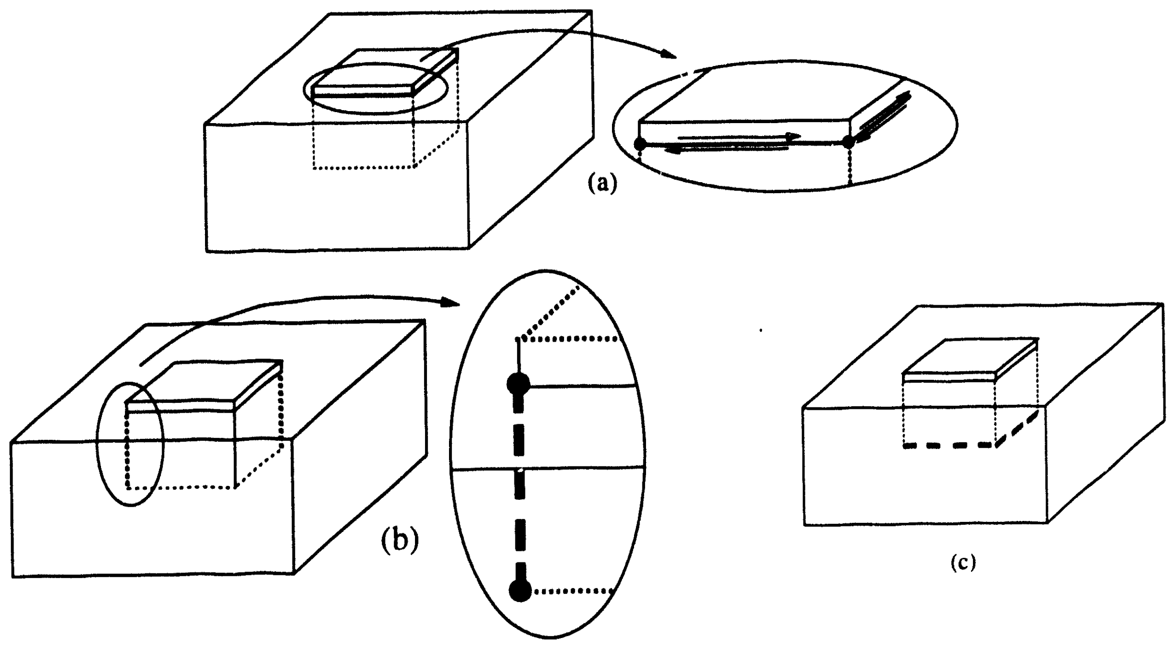


Figure 5:

(a) Phase I: Construct d-edges of the intersection curve.

(b) Phase II: Construct d-edges incident to the intersection curve. Edges that are incident to the intersection curve are constructed by testing to see if either segment of the d-edge incident to the intersection point lies in the interior of the other solid.

(c) Phase III: D-edges of one solid that are contained in the interior of the other are identified by ray-casting and added to the intersection-solid.

6.1 Preprocessing

Before we start identifying the d-edges of the output solid, we carry out a preprocessing step that calculates how the d-edges of each solid intersect the planes of the faces of the other. This is carried out in our algorithm by assigning each d-edge/face pair to a (virtual[1]) processor, and then having that processor calculate the intersection of the d-edge with the face. This step can be viewed as creating two matrices of processors. Each matrix is indexed by d-edges of one solid ("rows") and faces of the other solid ("columns"). A d-edge can intersect the plane of the face, it can be parallel, or a projection (ray) of the d-edge can intersect the plane of the face. Eventually we will be interested in whether those that intersect the plane actually intersect the interior of the face, the boundary of the face, or not at all.

We construct axis-aligned bounding boxes for each solid and use these boxes to eliminate intersections that are outside the bounding boxes.

[1]Throughout the algorithm we assume that the machine has enough processors. This often requires very many processors, *i.e.*, millions. We assume that the SIMD machine provides a means of simulating more processors than are physically available. In the remainder of the paper we drop the modifier "virtual."

We classify the intersections of the d-edges with the face planes and then create a sparse-matrix representation for the d-edge/face pairs of interest. Typically the remaining set is much smaller than the complete d-edge/face matrix. In addition, it becomes more sensitive to output size than input size, which is important in bounding the number of processors required for the remainder of the intersection algorithm.

6.2 Phase I — Constructing D-Edges of the Intersection Curve

The first group of d-edges of the new solid that we identify are those that lie on the boundary of both of the input solids. (See Figure 5(a).) These d-edges are formed by the intersecting faces of the two input solids. Recall that each processor is associated with a d-edge/face pair. Let f be a face of solid A and g a face of B. Denote an intersection of d-edge $e_f^\bullet$ with the plane of g as $e_f^\bullet/g$. The intersection points $e_f^\bullet/g$ and $e_g^\bullet/f$ lie along the line defined by the intersection of the planes of f and g. We reassign these intersection points to processors whose physical adjacency reflects this ordering along the line. Communication between neighboring intersection points along these intersection lines becomes a nearest neighbor communication, which is particularly efficient. We then identify those regions of the intersection line that are on both faces, and create the new d-edges that lie on the intersection curve. Determining which d-edges get built requires analyzing many cases, depending on whether or not the faces intersect in their interiors or on their boundaries. The key to efficient implementation is to structure the cases so that as much as possible is in common. While this step is conceptually similar to that used in a serial algorithm, there are major implementational differences, especially in how we use the ordering of points along the intersection line to make the computation efficient.

When the first phase of the algorithm is complete, we have identified precisely which d-edges intersect the faces (as opposed to the planes) of the other solid. This allows us to further refine the number of intersection points that we carry around. This in turn reduces the number of processors required (and hence possibly the virtual processor ratio) to one more tightly bounded by output size, and reduces the time required for future computations.

6.3 Phase II — Constructing D-Edges Incident to the Intersection Curve

The second group of d-edges of the new solid to be identified are those that are incident to the intersection curve at one or two points (see Figure 5(b)). These output d-edges are segments of d-edges of one of the input solids that intersect boundary elements of the other. In Phase I of the intersection algorithm we determined which of the d-edge/plane intersections actually intersect the solid boundary. Furthermore, we determined whether each d-edge intersects a vertex, edge, or face. What remains to be determined is which portions of the d-edge are in the interior of the other solid, and which are not. The most difficult of these occurs when a d-edge intersects a vertex. This case is refined to an equivalent d-edge/edge or d-edge/face test. The d-edge/face incidence test is simple (just a dot product). For a d-edge/edge test, we determine if the d-edge points between a volume-enclosing pair of faces incident to the edge. While conceptually similar to algorithms used at this stage on serial machines, the parallel implementations of the d-edge/edge and d-edge/vertex incidence tests are necessarily quite different.

6.4 Phase III — Construct D-Edges Interior to an Input Solid

The final set of d-edges of the new solid that we identify are those d-edges of one input solid contained in the interior of the other, as shown in Figure 5(c). These are built in a manner completely unlike that typically used on serial machines. We can divide these d-edges into two groups; those contained in lines that intersect the other solid, and those that are not. For the first group, we use the results of the incidence tests already computed in Phase II to determine whether the d-edge is interior or exterior to the solid. If in the second group, it is sufficient to know whether the other input solid is unbounded.

This ray-casting approach contrasts with the methods typically used in serial algorithms. Since the incidence tests of Phase II of the algorithm are comparatively expensive, they are not, in general, applied to rays. Instead, in a serial algorithm, Phase III is done using a transitive closure. It is possible that isolated regions of the input solid boundaries cannot be classified by transitive closure. In that case, the serial algorithm must use a ray-casting procedure to classify some point on that isolated portion of boundary. Then another transitive closure step is needed to classify the rest of that portion. Transitive closure is very inefficient on a parallel machine, since its execution time is proportional to the distance (in a graph sense) of a d-edge from the closest d-edge intersecting the boundary.

Sections 7 through 10 describe the details of the intersection algorithm.

7 Preprocessing Details

The goal of the preprocessing step is to intersect the d-edges of each input solid with the planes of the faces of the other. This computation can be viewed as constructing two matrices. The rows of each matrix are indexed by the d-edges of one of the solids. The columns are indexed by the faces of the other. This is shown below.

$(e^\bullet_0/f_0)$	$(e^\bullet_0/f_1)$	$(e^\bullet_0/f_2)$	$\cdots$	$(e^\bullet_0/f_n)$
$(e^\bullet_1/f_0)$	$(e^\bullet_1/f_1)$	$(e^\bullet_1/f_2)$	$\cdots$	$(e^\bullet_1/f_n)$
......	......	......	...	......
$(e^\bullet_m/f_0)$	$(e^\bullet_m/f_1)$	$(e^\bullet_m/f_2)$	$\cdots$	$(e^\bullet_m/f_n)$

The first stage of the preprocessor constructs axis-aligned bounding boxes for each of the input solids. Then the d-edge/plane intersections are calculated by building matrices containing the intersection points. Intersection points are characterized as to how the d-edge intersects the plane. We then use the bounding boxes then to cull intersection points. Finally, we construct a sparse-matrix representation for the surviving intersection points.

7.1 Calculating the Bounding Box of a Solid

The bounding box of a solid is an axis-aligned box containing it. The box is found by taking the extrema of the x, y, and z components of the coordinates over the vertices of the solid.

The Connection Machine provides global min and max operators. The function ***max**(a!!) obtains the maximum value, over all active processors, returning a value on the front-end processor. Using the min and max operators, the bounding box calculation is:

```
*when $(e^\bullet.SolidLabel = SolidA)$
        $ABoxMin.x = *\mathbf{min}(e^\bullet.InitialVertex.Coordinates.x!!)$
        $ABoxMax.x = *\mathbf{max}(e^\bullet.InitialVertex.Coordinates.x!!)$
        ...
end when
```

***when** selects the active context of the Connection Machine. Thus, the first line selects as the active set all processors storing a d-edge of solid A. The next two statements find the extremal values of the x coordinate of the parallel variable (pvar) $e^{\bullet}.InitialVertex.Coordinates!!$. This operation is performed in constant time.

Serial implementations of solid modellers calculate bounding boxes of each face, and then cull non-intersecing face-pairs by intersecting the bounding boxes. This reduces the expected time to compute the intersection curve (Phase I of our algorithm). This is a common algorithmic strategy on serial machines — cheap culling of most candidates, followed by expensive testing of the remainder. On a SIMD machine this strategy increases overall computation time. In fact, we do our bounding-box calculations to reduce the size of the virtual processor set, and hence reduce the computation time.

7.2 Intersecting D-Edges with Planes

In order to intersect two solids we intersect the d-edges of one with the faces of the other and vice-versa. The first step in doing this is interesecting the d-edges of one with the planes of the faces of the other (and vice versa, which we will stop saying now). We construct two matrices containing the Cartesian product of the d-edges and planes of the two solids.

The matrix containing the d-edges of solid A and the planes of B is constructed using the following procedure:

1. Allocate (number of d-edges of A) × (number of faces of B) processors for the matrix.
2. Copy d-edge i to the first column of the i^{th} row. This is done using the ***pset** communications operator and is collision-free.
3. Use the segmented-scan operator to propagate the d-edge across the row.

Although the Connection Machine supports two dimensional (or higher) processor addressing, the size of each dimension is constrained to be a power of two. Because our matrices do not, in general, satisfy this condition, we model our matrices as two-dimensional arrays of data, but program the Connection Machine as a one-dimensional array.

The plane data is placed in the array in a similar manner.

Since we do not use the two-dimensional processor addressing, we cannot scan down columns. We circumvent this by copying the planes to the first column of the matrix "transpose." We then scan the plane data across the rows of the transpose. Finally we transpose the plane data in place using a collision-free ***pset**.

The only data we need in order to calculate the intersection of a d-edge with a plane are two endpoints and a plane equation. Hence the (single processor) data requirement for each matrix entry is 40 bytes. (If a matrix entry is $e^{\bullet}/f$, then four 4-byte floating point numbers represent the plane equation of f, and three 4-byte floating-point represent the coordinates of each vertex of $e^{\bullet}$.) A full-up Connextion Machine processor contains 128K bytes; A VP ratio of 1024 (which doesn't even begin to tax the memory available) allows matrices with 64M entries. This allows us to intersect solids with many thousands of faces.

7.3 Characterizing the Intersection Points

Subsequent phases of the intersection algorithm require knowledge about the way in which a d-edge and plane intersect. Each matrix element (processor) intersects its d-edge with its plane, computing an intersection point (unless the d-edge is parallel to the plane). We classify the intersection in one of six ways (see also Figure 6).

transverse The interior of the d-edge intersects the plane at a point.

ray The line containing the d-edge intersects the plane at a point but the d-edge does not intersect the plane.

Isolated vertices don't have a second vertex to define a line. For isolated vertices we use the ray rooted at the isolated vertex, pointing in the $+z$ direction.

contained The d-edge is contained in the plane

vertex1 The initial vertex of the d-edge lies in the plane.

vertex2 The terminal vertex of the d-edge lies in the plane.

extraneous Initially these are the d-edges not contained in, but parallel to the planes. This will later be expanded.

The ray and extraneous classes are used only to determine whether d-edges that do not intersect the boundary of the other solid are contained in the intersection-solid. As will be explained in Section 10, most of the ray intersections are superfluous. We can identify most of these superfluous matrix entries at this stage of the algorithm, possibly reducing the VP ratio required to perform further computations.

The superfluous rays are: (1) The intersection point is outside the bounding box of the other solid. (Thus, they do not intersect a face.); (2) Either vertex of the d-edge inducing the ray lies outside the bounding box. (The d-edge either lies outside the other solid (X) or intersects the boundary of X. In the first instance the d-edge is copied to the output solid only if X is unbounded.); (3) The intersection point lies beyond vertex 2. (We define the ray to be rooted at vertex 1 and points backwards along the line containing the d-edge.)

7.4 Creating a Sparse-Matrix for the Intersection Points

It is generally the case that a large fraction of the d-edge/plane intersections are extraneous. By discarding them we can reduce the number of processors required for the remainder of the intersection algorithm. At this point we construct a sparse matrix representation of the two matrices, with each matrix element containing a d-edge and a face. The combinatorial structure of the d-edges is obtained from the input solids, and the intersection points from the full matrix. (Recall that the full matrix created in the preprocessor contained only geomtric data.)

We use a segmented-scan operation to find the non-extraneous matrix entries. We enumerate them in order to generate the addresses of the processors that will contain them in the sparse matrix representation. With each sparse matrix entry we record the row and column of the original matrix entry.

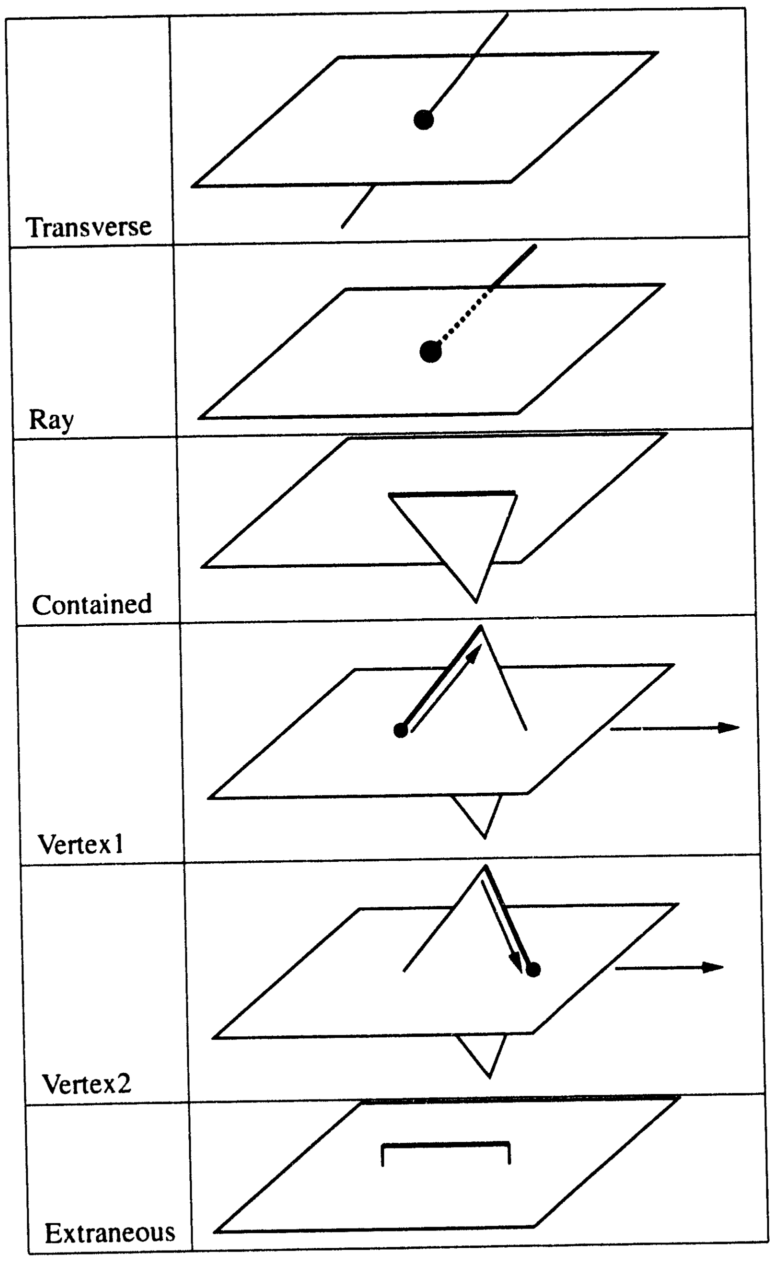


Figure 6: Characterizing d-edge/plane intersections

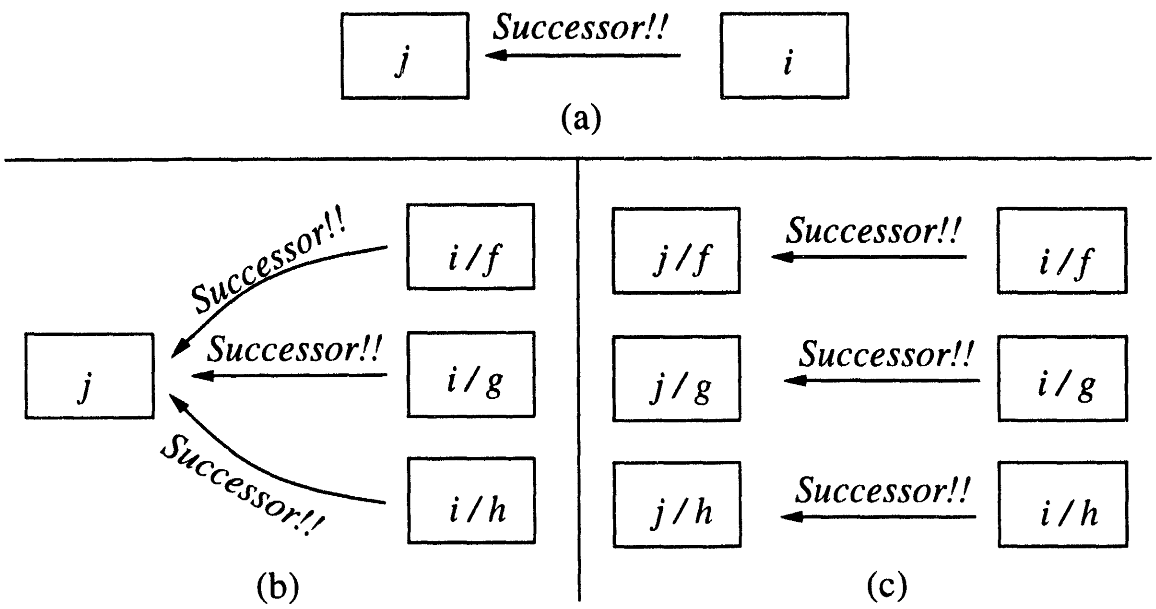


Figure 7: Reducing collisions in the matrix.

The combinatorial structure of a solid is given by the processor numbers in d-edge fields like *InitialVertex.Successor*!!. Suppose the d-edge (processor) labelled i in Figure 7(a) accesses a datum from d-edge j, its *Successor*!!. For the input solids, this is always collision-free because every d-edge is the successor of exactly one other. In creating the matrix representation, we create many copies of d-edge i. In part (b) of the figure, these many copies of i collide when they access j. In fact there are many copies of j in the matrix as well. In part (c) of the figure we show that by associating i/f with j/f, etc., we can eliminate the collisions. This means that in creating the matrix we have to update processor references to refer to processors in the same column.

In creating the sparse matrix, we must guarantee that these processor references are again updated and that they point to sparse matrix entries. Serendipitously, any pointer reference needed by the remainder of the intersection algorithm indeed points to a valid sparse matrix entry.

8 Phase I Details

The first major phase of the algorithm generates the portion of the output solid that lies on the intersection curve of the two input solids. We refer to these d-edges as *cross-face d-edges*. In Figure 8, we show the intersection of face f of solid A with face g of solid B. Cross-face d-edges are the result of these face-face intersections. The endpoints of the cross-face d-edges are the intersections of the d-edges of face f with face g and vice versa. (These are the intersections found earlier in the preprocessing step).

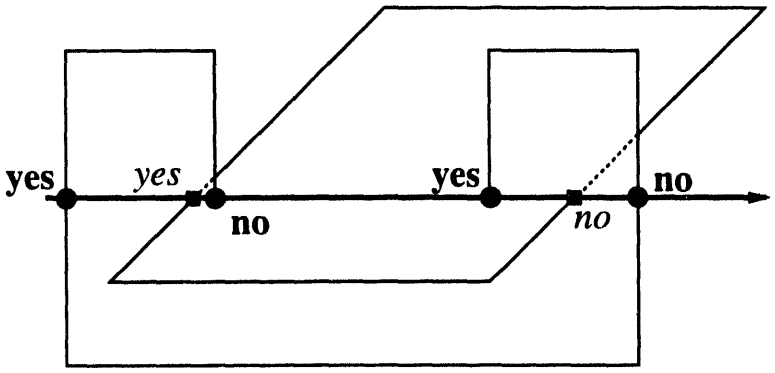


Figure 8: The intersection points of the two faces lie along the line l defined by the intersection of the two faces. Each dedge/face intersection point $e^{\bullet}_{f}/g$ is marked as to whether it might begin a cross-face (t points into the interior of the face of the intersection-point d-edge).

8.1 Classifying Intersection Points

Consider the line l, shown in Figure 8, defined by the intersection of f and g, and the direction t defined by $normal(f) \times normal(g)$. For each intersection of a d-edge of f (respectively g) with g (f), the intersection point lies on l, and may either begin or end a cross-face d-edge in direction t. Given an intersection point $e^{\bullet}_{f}/g$, we wish to determine whether the point may begin a cross-face d-edge along l in direction t. There are two cases:

1. If the intersection point is transverse and t points into the interior of f, then it may begin a cross-face d-edge.
2. If the intersection $e^{\bullet}_{f}/g$ is at the initial vertex of $e^{\bullet}_{f}$, and $e^{\bullet}_{f}$ is the d-edge that is immediatedly counterclockwise to t, then the intersection point may begin a cross-face d-edge. (See Figure 9.)

Intersection points that meet either of the above criteria only begin cross face d-edges if they are interior to the face of the other solid. We show how to test this containment is Section 8.3.

This is not quite correct. In fact, these intersection points can also either lie in the interior of edges of the other solid or coincide with intersection points from the other solid. We show how to deal with the first case in Section 8.2. The second case is dealt with by recognizing the coincidence (see Section 8.3.1) and then arbitrarily ordering the coincident points along the intersection line. Thus only one point can appear to be interior to the face of the other point.

The first case is dealt with by computing the sign of $t \cdot faceDirection(e^{\bullet}_{f})$.

The second case is more complicated. If vertex v of face f lies on the plane of face g, there are two or more d-edges of f that intersect g at v. Each of these intersection points constructs a local 2D coordinate system in which t is the x-axis and $normal(f) \times t$ is the y-axis. Each intersection point then computes the angle counterclockwise from the x-axis to its d-edge. The intersection point $e^{\bullet}_{f}/g$ whose d-edge is the first counterclockwise from the x-axis determines whether t points into f, and hence whether v begins a cross-face d-edge. Intersection point $e^{\bullet}_{f}/g$ begins a cross-face d-edge if its initial vertex is v.

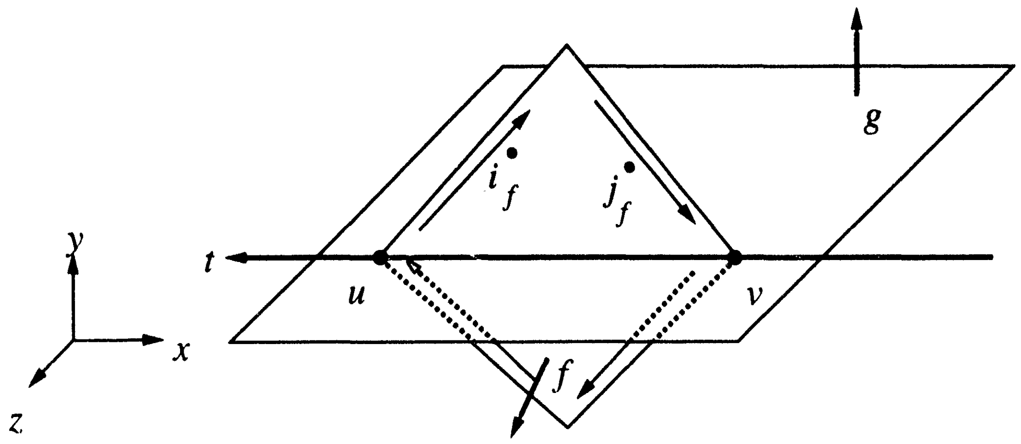


Figure 9: At vertex u, the d-edge that is immediately counterclockwise to t (on the plane of f) has an intersection point ($i_f^\bullet/g$) at its initial vertex, and so may begin a cross-face d-edge. On the other hand, the one counterclockwise to t at v ($i_f^\bullet/g$) intersects g at its terminal vertex, and so does not begin a cross-face d-edge.

Intersection points that have determined whether or not they may begin a cross-face d-edge are called classifiers. (In Figure 9 the two classifiers are $i_f^\bullet/g$ and $j_f^\bullet/g$.)

Although the above procedure is described in terms of angle computation, it is unnecessary to do the work implicit in the inverse trigonometric calculation. Because the angle is used to radially order the d-edges, any function that provides this ordering will suffice. We use the psuedo-angle of [6]. Given a local coordinate system with axes x and y, the pseudo-angle of a vector v is

$$sign(v \cdot y)(1 + v \cdot x).$$

The pseudo-angle is used every place in the algorithm where an angle is needed for creating a radial ordering.

As described above, only vertex1 intersection points may begin cross-face d-edges. Furthermore, each vertex1 intersection point tests to see if vector t lies radially between it and its radial predecessor (around its initial vertex). Although there is no radial predecessor field in the parallel d-edge data structure, a d-edge is the radial successor to its radial predecessor. Thus in the actual implementation, a vertex2 intersection point computes the angle and sends that angle using a collision-free ***pset** to its successor. This demonstrates how (1) "back-pointers" are eliminated from the data structure, and (2) careful structuring of the implementation allows the more efficient ***pset** operation to be used in place of a **pref!!** operation.

In fact, there is an additional complication when an intersection point $e_f^\bullet/g$ lies in l. If $tangent(e^\bullet) = -t$, then $e_f^\bullet/g$ behaves as a vertex1 intersection in the manner just described. However, if $tangent(e_f^\bullet) = t$, the vertex1 intersection point that receives its angle from $e_f^\bullet/g$ gets the wrong angle ($e_f^\bullet$ computed its angle with respect to the other

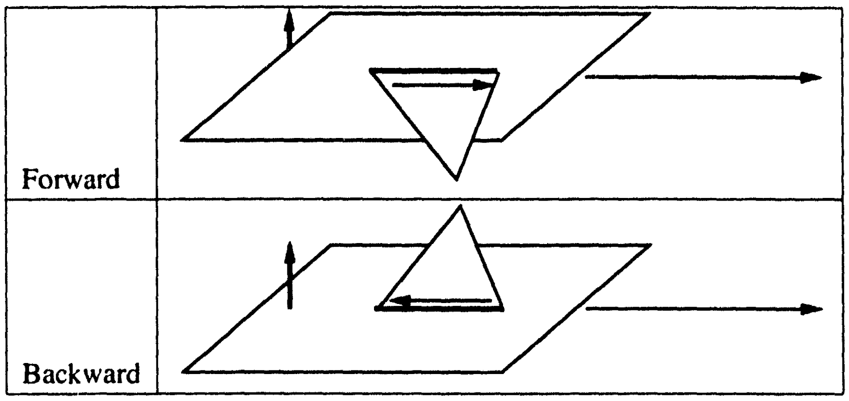


Figure 10: D-Edges contained in the intersection curve can be oriented either with or against the curve.

vertex). This case is specially handled by the vertex1 interesection point recognizing the (impossible) angle and adding π. (See Figure 10.)

Isolated vertices actually are simple for once. They always begin a cross-face d-edge.

8.2 Classifying D-Edges that Overlap the Intersection Curve

Intersection $e^{\bullet}_f/g$ is called a *contained* segment if $e^{\bullet}_f$ is (partially) contained in the intersection curve. A contained segment may contribute to the intersection curve at most one d-edge in the plane of f and at most one in the plane of g. The d-edge on the plane of f is built if $e^{\bullet}_f$ is interior to g and the face direction of $e^{\bullet}_f$ points below the plane of g. There is another d-edge $e^{\bullet}_{f'}$ of the underlying edge that is oppositely oriented and forms a volume enclosing pair with $e^{\bullet}_f$. The processors associated with these intersections each calculate a pseudo-face-direction using the plane of g and their d-edge tangent. By radially classifying these pseudo-face-directions against the volume-enclosing pair $[e^{\bullet}_f, e^{\bullet}_{f'}]$ we decide whether or not to build output d-edges on the plane of g. (See Figure 11.) These intersections are also referred to as classifiers.

If $e^{\bullet}_f/g$ overlaps a d-edge of face g, $e^{\bullet}_f$ does not determine whether to build a d-edge on the plane of f, as described above. In fact, $e^{\bullet}_f/g$ determines whether to build a d-edge of the plane of g, and d-edge $e^{\bullet}_g/f$ decides whether to build a d-edge on the plane of f.

There is a special case if f and g are coincident. This special case is further broken down by whether f and g are identicaily or oppositely oriented. If oppositely oriented, there is no volumetric overlap, and hence no d-edges to build. If the faces are identically oriented, it is necessary to decide on which face to build the output d-edge. The choice is arbitrary, so we always select the face (f) of solid A.

There is an additional complication associated with these tests. The classifier of an intersection d-edge is the intersection point that is at the beginning of the segment (with respect to direction t). However if a d-edge is contained in l but oriented oppositely to t then this test is done at the "wrong end" of the segment. If processor $e^{\bullet}_f/g$ contains

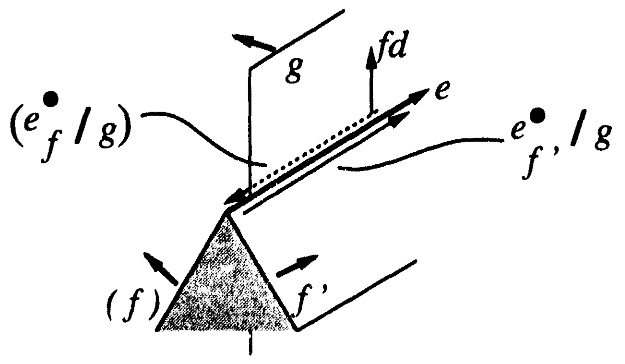


Figure 11: Edge e lies in the plane of g, and so induces two intersections $e^{\bullet}_{f}/g$ and $e^{\bullet}_{f'}/g$. The two processors associated with these intersection points classify pseudo-face-directions with the plane of g in order to determine whether or not to build d-edges on the plane of g. Here vector fd shows one of the pseudo face-directions. The other points in the opposite direction, into the interior of the volume enclosing face-pair.

such an intersection, it recognizes this case, and sends the result of the test, collision-free, to $e^{\bullet}_{f'}/g$, the processor named by $(e^{\bullet}_{f}/g).TerminalVertex.Successor!!$.

8.3 Finding Endpoints of D-edges of the Intersection Curve

Before we can build d-edges of the intersection curve we need to determine whether an intersection point lies in the interior of the other solid's face. An intersection point interior to the other solid's face begins a cross-face d-edge along l in direction t that ends at the next intersection point. Thus, if we sort the points along l we can easily access all the information required to build the new d-edge.

In order to construct the correct combinatorial structure of the intersection solid, we have to recognize that different intersection points correspond to the same vertex.

We begin by ranking the intersection points along the line l in direction t. In the event of ties (coincident points), it is necessary to insure that the highest ranked intersection point of a cluster is a classifier. (Recall that classifiers are the intersection points that can mark the beginning of a d-edge of the intersection curve.) Because communication is much more efficient when the routing corresponds to the physical connectivity of the underlying network, we reassign the ranked intersection points to neighboring processors.

8.3.1 Naming Vertices of the Intersection Curve

We need to insure that all copies of an intersection point are given the same name. By naming the intersection point with the concatenation of the names of the intersecting boundary element of solids A and B, each intersection point has a unique name because of the intersection property of the parallel d-edge representation. This requires recognizing the boundary element of the other solid with which the point coincides.

Recall that the intersection points have been sorted along lines of plane-plane intersection. We use the coordinates of the intersection points to determine whether points adjacent in the rank are different. Numerical differences between point coordinates delimit clusters of identical points (see Figure 12). We use a segmented scan to propagate the boundary element names contributed by each input solid to each

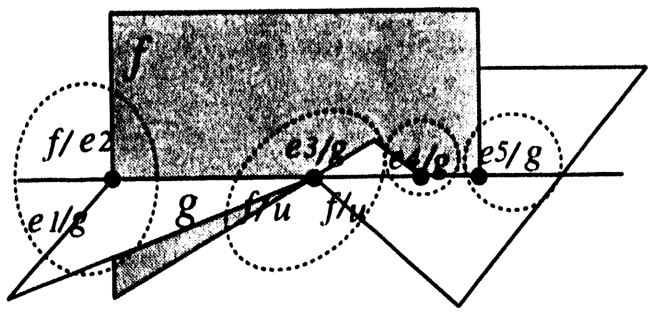


Figure 12: The intersection of faces f and g has four clusters of intersection points. Initially each intersection point is named by an (A-name, B-name) pair — one of the names corresponds to a face of the other solid, and the naming procedure may refine that name to be the name of a vertex or an edge. For example, after the naming procedure the name-pair of the first intersection point is $e1/e2$.

intersection point in the cluster. We concatenate the computed names of the intersection points to form a key, rank the keys, and count off the number of different keys using a segmented scan. The index in this count becomes the new name for this vertex.

This procedure fails to recognize intersection points interior to d-edges of the other solid which are contained in the line of intersection, l. This case is detected by ranking the intersection points according to the inducing edge of each solid as the primary key and the name of the intersected face of the other solid as the secondary key. Within a cluster, intersection points that recognize themselves as coincident with an edge of the other solid are forced to be first. (Because edges are defined by at least two planes and vertices by at least three, every cluster of intersection points in the interior of an edge of the other solid has one intersection point that recognizes itself as coincident.) We use a segmented scan to propagate its name as well as the name of the containing edge to all intersection points in the cluster.

8.3.2 Constructing the D-Edges on the Intersection Curve

The preceeding sections described how to identify endpoints of potential cross-face d-edges. Only those candidates which are interior to the face (as opposed to the plane of the face) of the other solid cause d-edges to be built. Additionally d-edges lying on l, the intersection of a pair of planes may overlap a d-edge of the other solid. Recall that intersection points are sorted along l. The last point in a cluster of coincident points is a classifier; that is, it determines whether the segment of l in direction t is interior to its face or edge. By using a segmented scan we propagate classification tags ("in face" and "in edge") along l to the intersection points of the other solid (see Figure 13).

One endpoint of a cross-face d-edge is a classifier whose own classification tag is true, and who received a true tag from an intersection point of the other solid. The other endpoint is the next point along the line l that is not a ray intersection. This is found using a scan in direction t along l.

Finally, we build the d-edges. There are three classes of d-edges that can be built at this stage; cross-face d-edges (those that are interior to faces of both solids), d-edge/face segments (those that are interior to an edge of one solid and a face of the other solid), and d-edge/d-edge segments (those that are interior to edges of both solids). Based on the propagated classification tags, each intersection point determines

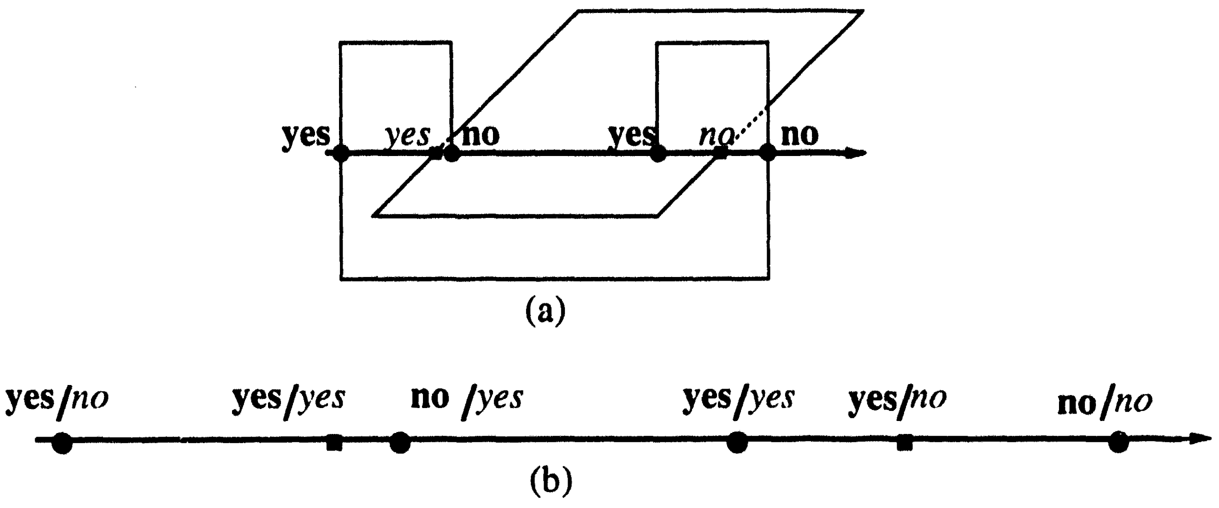


Figure 13: (a) In order to know whether to construct a d-edge of the intersection curve, we scan the classifier tags ("in face," "in edge") along the intersection curve between two faces. (b) the classifier tags have been propagated to the intersection points of the other solid by a segmented scan.

how many and what kind of d-edges to build. It then builds the d-edges to the endpoint identified in the reverse scan described above. Orientation of the contructed d-edges is defined to be positive for d-edges with terminal-vertex name greater than initial-vertex name.

It is very difficult to identify isolated vertices of the intersection solid. An isolated vertex arises in two ways; it was isolated in one of the input solids (and remained so), or it was a vertex of an input solid incident to the interior of a face of the other. The second case cannot occur if a d-edge built at this stage is incident to the vertex. Therefore, we record the faces incident to each vertex of the intersection solid, and whether or not there is a d-edge incident to the vertex each of these faces. This is stored as two bits in a "matrix" with rows corresponding to the d-edges of one solid and columns being the faces of the other. One bit indicates a vertex/face incidence, and the other indicates that a d-edge was built incident to the vertex on the face.

8.3.3 Eliminating Extraneous Intersection Points

To this point, we have not distinguished between d-edges of one solid (say A) that intersect the boundary of solid B and d-edges of A that merely intersect planes of faces of B. In the steps just completed, we made this distinction. We can now identify extraneous intersections, and eliminate them from the sparse intersection-point matrix. This reduces the processor requirements to an output sensitive size. In practice this leads to a dramatic reduction in the virtual processor ratio.

There are a number of places in the algorithm where the virtual processor ratio is reduced. The operation is conceptually simple, although there are a few tricks required to implement it. All relevant elements of the pvar are identified and marked. We then count them up and determine whether they can be fit into a smaller vp-set, and what that set's size is. We then create a new vp-set of that size, create a duplicate pvar in the

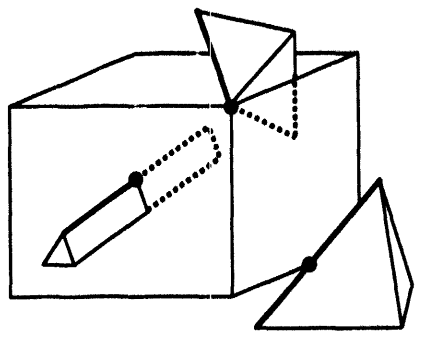


Figure 14: The kinds of edge-incidence tests that are necessary are edge/face, edge/edge, and edge/vertex tests.

new vp-set, and copy the marked elements into the new set. Depending upon how this data is represented and utilized it is sometimes necessary to update pointers in the pvar. Other times the data is accessed associatively, and no pointer updates are required. There are some subtleties in creating and destroying improperly nested vp-sets, but it can be done.

9 Phase II Details

The second phase of the intersection algorithm constructs d-edges of the intersection-solid that have one or both endpoints on the intersection curve, but whose interiors are contained in the interior of the other solid. Each intersection point is the intersection of a d-edge (or vertex) of one solid with a vertex, edge, or face of the other. These three cases are shown in Figure 14. Once this incidence testing is complete, the intersection points are sorted along the input d-edges, and the intersection d-edges are constructed. The incidence tests are intricate; the d-edge construction is straighforward. Note that vertex/boundary incidence tests are incorporated into the d-edge/boundary tests, and so they are not described explicitly.

The result of any of these incidence tests is a "build forward" bit associated with the intersection point. If the bit is set on, then we will eventually construct a d-edge from the intersection point either to the next intersection point along the d-edge or to the terminal vertex.

9.1 Face Incidence Testing

An intersection point $e^\bullet/g$ in the interior of face g builds a d-edge of the intersection solid on that portion of $e^\bullet$ which is below g (see Figure 15). The vector $tangent(e^\bullet)$ points below g if $tangent(e^\bullet) \cdot normal(g)$ is negative. This is a local computation. As a result of this test, the "build forward" bit associated with the intersection point is set on or off.

9.2 Edge Incidence Testing

Edge incidence testing uses the radial ordering of the faces around an edge to determine if an incident d-edge points into the interior of the subtended volume. For intersection point $e^\bullet/g$ incident to an edge E, Phase I of the algorithm has identified the intersected edge, the name of that edge, and its tangent. It is necessary to determine whether or not $\pm tangent(e^\bullet)$ points into the interior to the other solid at E (see Figure 16).

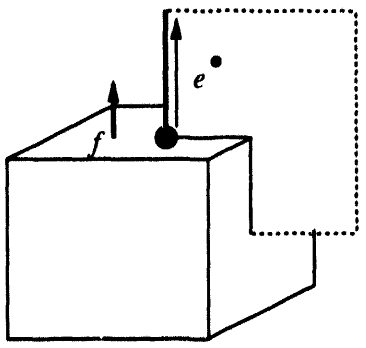


Figure 15: The tangent of d-edge $e^\bullet$ points above face f, and so we set the "build forward" tag to false.

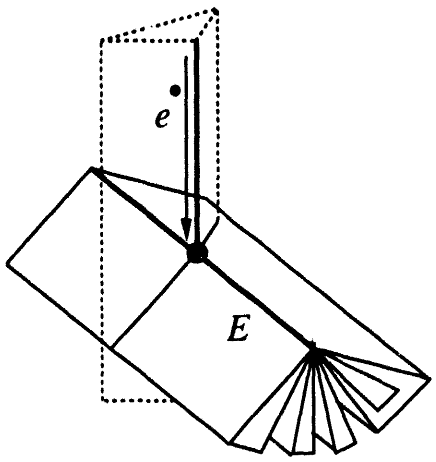


Figure 16: The vector $-tangent(e^\bullet)$ does not point between a volume-enclosing face pair adjacent to edge E, but vector $+tangent(e^\bullet)$ might.

Because there is always another d-edge $e'^\bullet$ with the same underlying edge as $e^\bullet$ but oriented in the opposite direction, we only test in the direction $+tangent(e^\bullet)$.

In order to determine whether the d-edge $e^\bullet$ enters the interior of the other solid it is necessary to determine whether $e^\bullet$ lies between a pair of volume-enclosing faces incident to the intersected edge E. Let $E_1^\bullet, E_2^\bullet, \ldots, E_3^\bullet$ be the d-edges of edge E. We create a local coordinate system in which the z-axis is $tangent(E)$. For each $E_i^\bullet$ we calculate the angle counterclockwise from the local x-axis to vector $faceDirection(E_i^\bullet)$. Candidate d-edge $e^\bullet$ calculates the counterclockwise angle from the local x-axis to the projection of $tangent(e^\bullet)$ onto the local xy-plane. We rank the intersection point $e^\bullet/g$ and the d-edges of E by this angle. If the d-edge of E that preceeds $e^\bullet$ in the rank is positively oriented, then $e^\bullet$ points into the solid (see Figure 17). The result of this test is used to set "build forward" on or off.

All candidate d-edges $e^\bullet$ that intersect E must create the same local coordinate system as the d-edges of E. We do this by selecting the component of $tangent(E)$ with the smallest magnitude, and setting the x-axis to the projection of the corresponding principal direction onto the plane with normal $tangent(E)$. The y-axis is obtained from $tangent(E) \times x$.

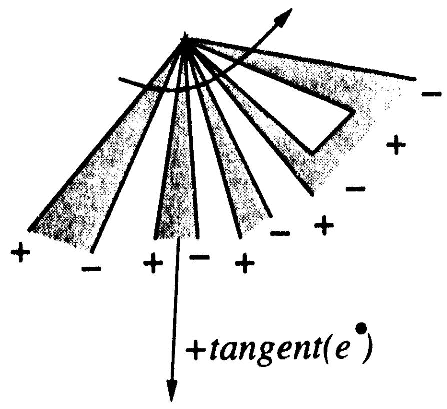


Figure 17: The vector $-tangent(e^\bullet)$ is preceded in the rank by a vector that is the face-direction vector of a postively oriented d-edge of E, and so the "build forward" tag of the candidate intersection point to be true.

Although there may be many copies of the d-edges of E, we only need one copy of each d-edge to compute the angle from the x-axis to its face direction vector and then participate in the circular scan. In creating the sparse matrix representation of the intersection points, we guarantee that at least one such d-edge survives. If there is more than one, the edge incidence test uses only a single copy. This allows us to reduce the size of the virtual processor set for the edge incidence test. Experience has again shown that this is a valuable savings.

Because the $E_i^\bullet$'s and the $e^\bullet$'s that intersect E are radially arranged, there is circularity in the ranking. In fact, the radial predecessor of the object with smallest rank is the one with largest rank. A scan is needed for an $e^\bullet$ to obtain the orientation of the preceeding $E_i^\bullet$. This is because many d-edges $e^\bullet$ may lie radially between consective d-edges of E. In order to deal with the circularity we use a segmented circular scan described earlier.

9.3 Vertex Incidence Testing

The vertex incidence test finds the edge or face that is angularly closest to the candidate d-edge. Then an incidence test with this closest object determines the containment of the d-edge. For intersection point $e^\bullet/g$ incident to a vertex V, Phase I of the algorithm has identified the intersected vertex and the name of that vertex. It is necessary to determine whether or not $\pm tangent(e^\bullet)$ points into the interior of the other solid at V. Just as for edge-incidence testing, we only need to test in direction $+tangent(e^\bullet)$ (see Figure 18).

Face g is one of the faces incident to vertex V. Let $E_{g,1}^\bullet, E_{g,2}^\bullet, \ldots$ be d-edges of g incident to V. The angular distance from $e^\bullet/g$ to g is the solid angle from $tangent(e^\bullet)$ to the plane of g if the tangent projects into the interior of g. Otherwise, it is the minimum of the solid angles from the tangent to each $E_{g,i}^\bullet$. Just as we did for edge incidence testing, we find the solid angle by constructing a local coordinate system on g. We calculate the angle counterclockwise from the local x-axis to each d-edge $E_{g,i}^\bullet$ and to the projection $P(t)$ of the tangent onto g. We rank the angles. By looking at the the radially closest d-edge of g to the

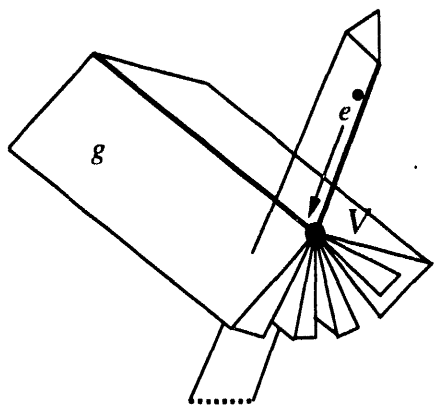


Figure 18: We need to determine whether or not $+tangent(e^{\bullet})$ points into the interior of the solid of vertex V. We do this by finding the edge or face bounded by V that is angularly closest to $+tangent(e^{\bullet})$, and then using an edge- or face-incidence test.

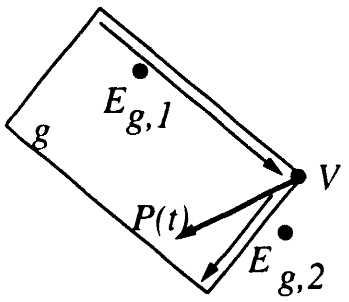


Figure 19: The projection (t) of $tangent(e^{\bullet})$ points from V into the interior of g, and so the radial angle between g and $tangent(e^{\bullet})$ is the angle between t and $tangent(e^{\bullet})$. If t had pointed outside of g, then the radial distance to g would have been the angle from $tangent(e^{\bullet})$ to $E^{\bullet}_{g,1}$ or $E^{\bullet}_{g,2}$, the d-edges of g that bracket t in the radial ordering.

projection, we can determine whether $P(t)$ points interior to g, or which d-edge of g is closest. From this we can calculate the solid angle.

By selecting the minimum over all the faces g incident to V, we obtain the face or edge that is closest to $e^{\bullet}$. If the closest is a face, we perform a face-incidence test. If the closest is an edge, we perform the edge-incidence test. The result of these tests is used to set the "build forward" but on or off.

The SIMD implementation of the intersection algorithm does all vertex-incidence testing before any edge or face-incidence testing, so that the vertex-incidence-induced edge- or face-incidence testing can be done with the other testing.

Just as for the edge incidence test, there may be many d-edges $e^{\bullet}$ intersecting vertex V, and because the radial ordering is circular, we again use a segmented circular scan.

Although there may be many copies of the d-edges of g incident to V, we only need two copies of each d-edge — one for calculating angles around the initial vertex, and one for the terminal vertex. In creating the sparse matrix representation of the intersection points, we guarantee that at least two such d-edge survives. If there are more than two, the vertex incidence test uses only two copies. This allows us to reduce the size of the virtual processor set for the vertex incidence test. Experience has again shown that this is a valuable savings.

9.4 Sorting Intersection Points Along D-Edges

Once one endpoint of each Phase II d-edge has been identified by the incidence tests, we need to identify the other endpoint and then build the d-edge. The other endpoint is either the terminal vertex of the d-edge inducing the intersection point, or the next intersection point along the d-edge (towards the terminal vertex). We sort all the non-extraneous intersection points along their inducing d-edges using as a key the d-edge name (*i.e.*, the row number of the matrix entry containing the intersection point), the distance along the d-edge, and the build forward tag. By making the build forward tag the low order bit of the key, we guarantee that an intersection point with the bit set (one that builds a d-edge) is last in a cluster of identical points and is therefore adjacent (in the rank) to the intersection point that defines the other endpoint of the d-edge to be built.

9.5 Constructing D-Edges Incident to the Intersection Curve

We have seen that the incidence tests provide a tag indicating whether there is a d-edge to be built in the "forward" direction along an input d-edge. It is also possible that we may need to build backward along the input d-edge. (This only occurs if the initial vertex of the input d-edge is interior to the solid.) Any d-edge with the build forward tag set notifies its radial successor around its underlying edge. A d-edge receiving such a message determines (based on the sort along the d-edge) whether it is the first intersection point. If so, it sets a build backward tag.

In order to build the intersection d-edge, an intersection point with the build forward tag set looks at the next-ranked intersection point. When the build backward tag is set we are always building to the initial vertex, and hence have no dependence on the sort order. The intersection points with the build-tags set build d-edges on the plane of their face using the appropriate endpoints.

There are certain configurations where our incidence tests might determine that a d-edge should be built, when in fact the d-edge has already been built by Phase I of the intersection algorithm. Such a configuration is shown in Figure 20. In order to deal with this case, the d-edge construction part of Phase I of the algorithm saves the names of all constucted d-edge endpoints. Then in Phase II, if an incidence test indicates that a d-edge should be constructed, we test to see if the d-edge was constructed in Phase I of the algorithm.

10 Phase III Details

In Phase III we identify d-edges of the input solids that do not touch the boundary of the other solid, but are part of the intersection solid. These d-edges are of two types, distinguished by whether their rays contact the boundary of the other solid or not. If the ray contacts the boundary, the ray intersection has already been classified by one of the boundary incidence tests. Since we restricted rays to those whose intersection

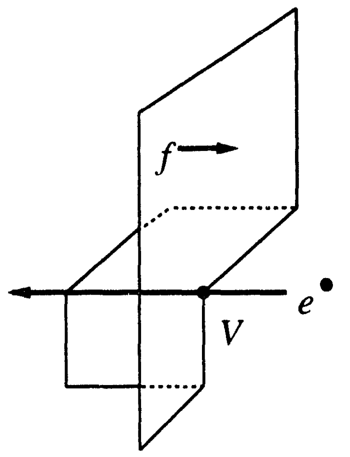


Figure 20: Because d-edge $e^\bullet$ is collinear with an edge of the other solid, the only intersection point that participates in the vertex-incidence test with V is $e^\bullet/f$. We evetually do a face-incidence test with f, and set the "build forward" tag to be true. We must check for this case and prevent the d-edge from being constructed — the d-edge was constructed, if necessary, in Phase I of the intersection algorithm

immediately preceeds its initial vertex (*i.e.*, there are no other intersections between the ray intersection and the initial vertex), the positive result of the incidence test implies that the entire d-edge is interior to the other solid. (See Figure 21).

Any d-edge that neither touches the boundary nor has a ray that touches the boundary of the other solid survives if and only if the other solid is unbounded.

We can finally determine which vertices of the intersection solid are isolated. If vertex v is isolated in the input solid, then v is isolated in the output solid if and only if it is interior to the other solid (which is determined by the ray test — we use the ray $+z$ for isolated vertices) or if the face of the input solid on which it is isolated is coplanar with a face of the other input solid, and is interior to that face. (That it is interior to the coplanar face is determined from data stored during the initial cross-face building process.) A non-isolated vertex of one of the input solids may become an isolated vertex of the intersection solid. This occurs when the vertex is interior to a face of the other solid, the local neighborhood of the face surrounding the vertex survives, and there are d-edges incident to the vertex, but not in the face. Determining that there are incident d-edges is done by saving a bit when d-edges are built. (One bit is saved for d-edges built on the face in question, and another bit for d-edges out of the face.) That the neighborhood survives is determined using a test similar to the edge-incidence test. The edge of the vertex that is angularly closest to the face in question is used to classify a vector pointing from that edge to the plane of the face. If the vector points into the solid bounded by the edge, then the local neighborhood of the face survives. The face-neighborhood classification required for this isolated vertex computation in intermingled with the edge incidence tests done in Phase II of the algorithm. This is done for efficiency on a SIMD processor.

There is a small amount of post-processing that we have not implemented – namely eliminating redundant edges and vertices. These are edges that are not defined by the intersection of two or more faces and vertices

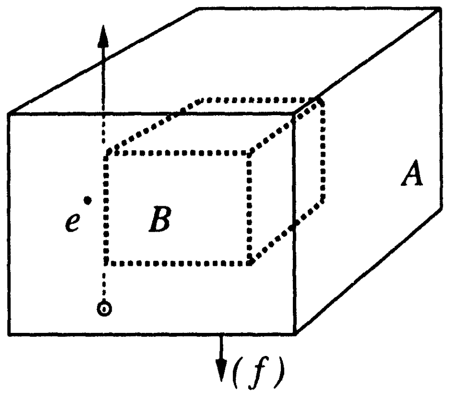


Figure 21: The ray intersection of d-edge $e^{\bullet}$ of the inner cuboid (B) is in the interior of bottom face f of the other solid, and so d-edge $e^{\bullet}$ is interior to that other solid. On the other hand, no ray intersections of solid A intersect the boundary of B, and so the d-edges of A are part of the intersection if and only if B is unbounded.

that are not defined by the intersection of three or more.

11 Performance Measurements

The parallel solid modelling algorithm described in this paper has been implemented on a Connection Machine CM-2. (Since these tests were made (1990), a faster model of the CM-2 has appeared.) In order to evaluate the performance of our algorithms, we compared our modeller to GDP, a production modeller internally developed at IBM Research. GDP is executed on an IBM/3090 600S. (This is the only supercomputer for which we have been able to identify a general purpose polyhedral solid modeller.)

This comparison is very difficult to perform, as it is in many respects an "apples and oranges" kind of comparison. The CM-2 is constructed of a large number of very simple (bit-serial) processors connected by a powerful routing capability. On the other hand, the IBM/3090 is a classic "big-iron" supercomputer which achieves it's speed through the use very high speed components, pipelines, vector-processors, etc. We believe the points to focus on are cost effectiveness and pathways to higher throughput. Machines like these are notoriously difficult to price accurately. Nonetheless, it is safe to assume that the IBM/3090 is the more expensive of the two machines tested here, by a factor of probably 2 to 5. In terms of achieving higher throughputs, the CM-2 offers continued growth possibilities, at least by several factors of 2, at roughly linear cost for the overall machine. In addition, the CM-2 does not use aggressive IC technology, so *processor* speed (though perhaps not communications speed) can be increased by more aggressive use of high speed IC technology. The IBM/3090 already uses some of the most aggressive technology to achieve high speed. Future speed increases come at the cost of sigficant complexity and high-risk technology.

Algorithm comparison is also an apples and oranges case. The algorithm presented here is designed specifically for a SIMD architecture massively parallel processor. It would be pointless to attempt to execute a version of it on a single processor machine. We believe the relevant issues are that GDP is a highly optimized modeler executing essentially the same functionality in these tests. That is, the representation is a vertex, edge, and face boundary representation, and the intersection is calculated by intersecting the boundary components of the two input solids. GDP development represents more than 100 times the effort devoted to the new algorithm presented in this paper.

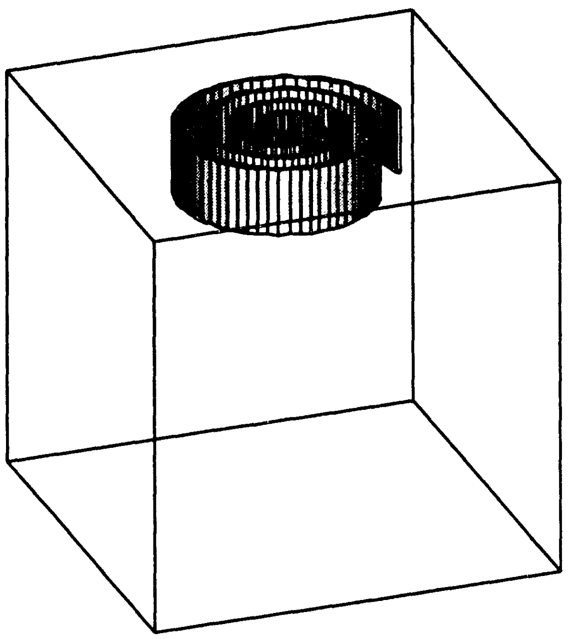


Figure 22: Toolpath test problem — block - spiral.

Because the performance of an intersection algorithm is so dependent on the two input solids, it is hard to define a representative problem for benchmarking purposes. The algorithm causes the vp-ratio to grow roughly proportionally to the product of the number of faces in the two input solids. In turn, run-time is a function of the vp-ratio. In addition, the code will skip (large and time-consuming) portions if no intersections of certain types occur, such as edge-vertex intersections). To show the range of performance, we have generated two examples at roughly opposite ends of the range. The first example has all the faces of one solid intersecting a single face of the other. Thus, "complexity" grows linearly with the problem size. In addition, all the intersections are in "general position." That is, they are all edge-face intersections, and therefore particularly easy to analyze. The second example balances the sizes of the two solids, so the number of face-face pairs grows like n^2. In addition, all the intersections are edge-edge intersections, which require many of the time-consuming procedures to be executed.

The first example is chosen to idealize a toolpath verification problem or a technique used for robot motion planning. In this problem, we take a faceted approximation of a toolpath and use it to "mill" a path in the top of a cube. A sample toolpath are shown in Figure 22, and the results of comparing our algorithm to GDP is shown in Figure 23(a). The graph shows speedups of up to 35 times. For 65K processors, performance has not even reached the linear domain since the machine is not yet being fully utilized. If we were to extrapolate performance of GDP to larger problems (which it cannot execute because of array addressing restrictions in the current implementation), speedups of 50 or better are predicted. For problems which fully occupy the machine, the data as well as data for 32K prrocessors (which is not shown here) show a nearly perfect linear speedup in the number of processors.

The "milling" problem is easy for both serial and parallel intersection algorithms because all of the

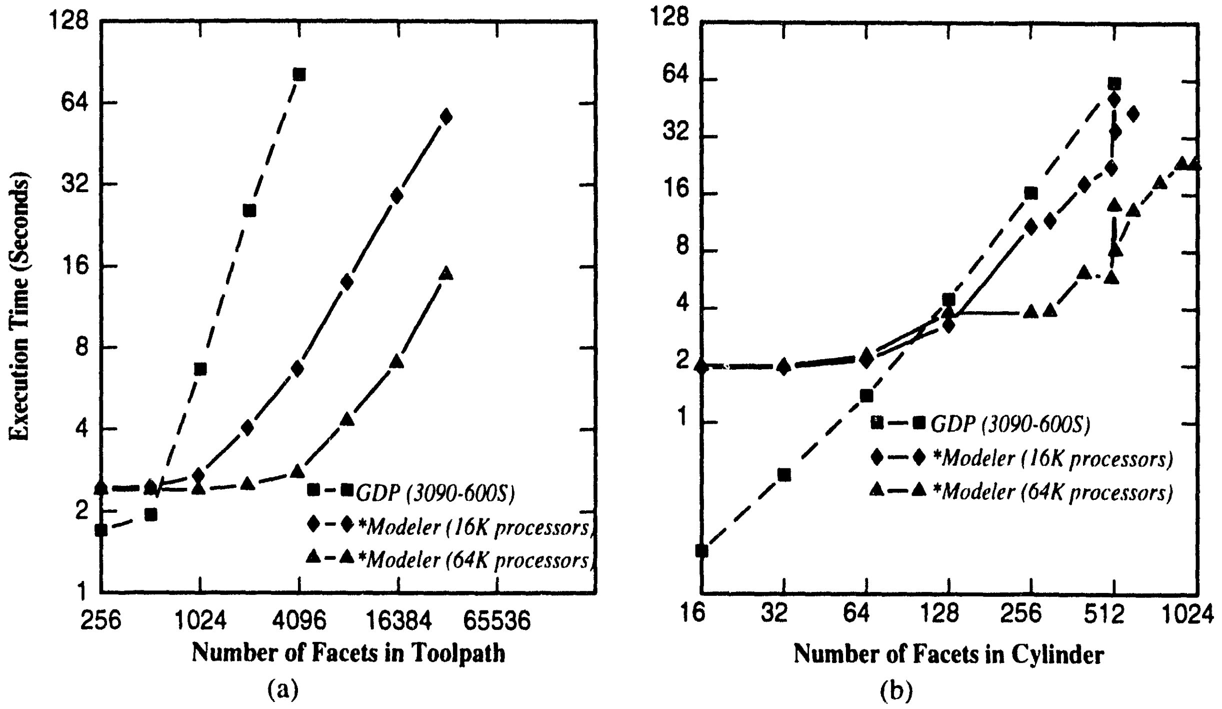


Figure 23: (a) "Milling" a path on a cube with a n-faceted approximation to a spiral. (b) Making a $2n$-faceted approximation to a cylinder by intersecting two n-faceted ones.

intersection points are non-singular, edge/face intersections. As the number of singular intersections increases, both serial and parallel intersection algorithms will slow down. We generate our second, somewhat pathological case by taking a faceted approximation to a cylinder, duplicating it, rotating the duplicate one half-facet, and then intersecting the two solids, obtaining a faceted approximation to a cylinder with twice the number of facets. All of the intersection points are singular — they are all edge/edge intersection points. As the number of singular intersection points increases, we expect our algorithm to degrade relative to a serial intersection algorithm because more interprocessor communication is necessary. Performance of the parallel algorithm compared to GDP is shown in Figure 23(b). The intersection algorithm is still faster on the Connection Machine, but the disparity is smaller. The discontinuities induced by changing VP-ratio are very apparent for this example. (The spike in the Connection Machine data is be due to a known router utilization bug for n = 512 facets.)

It is common in analyzing parallel algorithms to try to determine a speedup factor in comparison to execution on a single processor. Because our algorithm is designed explicitly for execution on a massively parallel SIMD processor, this sort of comparison is not possible. In the future, just as specialized parallel algorithms with no serial analog become more common, so will the sort of functional comparison we use. Nonetheless, our results do show nearly linear speedup in the number of processors as we move from 16K to 64K processors. We believe that the most relevant comparison will be computational cost effectiveness. On that score, our implementation compares very favorably with a more traditional serial algorithm on a serial supercomputer.

12 Conclusions

Solid modelling is a computationally demanding technology that underlies many key technologies in modern engineering design. We have shown in this paper how an intersection algorithm for solid modelling can be implemented on a massively parallel processor. Computational experiments have demonstrated the speed of this algorithm compares favorably to that of a serial algorithm on a supercomputers.

There are three main contributions to this work. First is the parallel d-edge data structure. In contrast to the multi-linked data structures common to serial solid modellers, we have shown how to embed the rich combinatorial structure in a distributed, uniform structure appropriate to a massively parallel processor. Our implementation of solid modelling algorithms is tailored to this data structure so as to frequently reduce or eliminate interprocessor communication. Second, we have dealt with all of the singularities that arise in an intersection algorithm. Dealing with them is difficult even for serial solid mdellers, and we have presented algorithms that efficiently incorporate the treatment of singular cases into a SIMD algorithm. Third, our avoidance of transitive closure procedures anywhere in the intersection algorithm allows us to have an algorithm whose execution time is independent of input solid topology. We use a variety of techniques whose efficiency is attributable directly to the SIMD nature of our algorithm as well as the data structures that we use.

Because the best implementations of solid modelling algorithms are primarily combinatorial, there is no meaningful measure of the megaflop throughput achieved. Furthermore, our implementation is structurally different from that used in serial algorithms, substituting a larger, homogeneous data-structure for serial computations. Hence, the only meaningful comparison of performance between these two approaches requires the use of benchmark comparisons of standard problems. Since many of the applications built on solid modeling are interactive in nature, the most important measure is how fast the algorithm runs, not an abstraction like megaflops or mips.

We have shown in our tests that our parallel intersection algorithms outperforms a serial one by a factor of 35 or better. As the models become bigger, the disparity increases. Furthermore, the nature of our intersection algorithm and the architecture of the Connection Machine allow scaling to essentially arbitrarily large arrays of processors with linear scaling in algorithm performance. To provide a reference point on model size, we have examined applications in a variety of industries. Solid models are used in integrated circuit process modeling. These models require tens to hundreds of facets per transistor. Thus, even small models of very limited portions of chips contains hundreds to thousands of facets. Ideally one might want to model a complete integrated circuit. A complete chip can have as few as several hundred transistors in the case of very simple "glue chips" to as many as a million transistors or more in state of the art microprocessors and memory chips. Obviously these chips are not currently modeled because of the impossibility using existing technology. In the aerospace field solid models are used to represent aircraft and their various components. Applications of these models range from finite element analysis to testing feasiblity of maintenance procedures. These models, even for small aircraft such as cruise missles, can have a hundred thousand facets or more. Thus, there are industrially interesting models in the size range at which our modeler appears to have a major advantage.

References

[1] M. J. Attalah and M. T. Goodrich. Efficient parallel solutions to geometric problems. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 411–417, State College, PA, 1985.

[2] B. Chazelle. Computational geometry on a systolic chip. *IEEE Transactions on Computers*, pages 774–785, September 1984.

[3] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molner, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, pages 79–88, July 1989.

[4] M. T. Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 127–136, Santa Fe, NM, 1989.

[5] C. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer*, pages 31–41, March 1989.

[6] M. Karasick. *On the Representation and Manipulation of Rigid Solids*. PhD thesis, McGill University, Montreal, Quebec, 1988. (available as Cornell Univ. Dept. of Computer Science 89-976, Ithaca, NY).

[7] G. Kedem and J. Ellis. The raycasting machine. In *ICCD '84*, pages 533–538, October 1984.

[8] D. E. Knuth. *The TEXbook*. Addison-Wesley, New York, 1984.

[9] D. Laidlaw, W. Trumbore, and J. Hughes. Constructive solid geometry for polyhedral objects. *Computer Graphics*, pages 161–170, August 1986.

[10] Y. Nakashima, H. Niimi, K. Shibayama, and H. Hagiwara. A parallel processing technique for set operations using three-dimensional solid modelling. *Trans. Info. Proc. Soc. Japan*, 30(10):1298–1308, October 1989. translated from the Japanese.

[11] C. Narayanaswami. *Parallel Processing For Geometric Applications*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1990.

[12] A. Requicha. Mathematical models of rigid solid objects. Technical Report Tech. Memo 28, Univ. of Rochester Production Automation Project, Rochester, NY, 1977.

[13] A. Requicha. Representations of rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, pages 437–464, December 1980.

[14] A. Requicha and H. Voelcker. Constructive solid geometry. Technical Report Tech. Memo 25, Univ. of Rochester Production Automation Project, Rochester, NY, 1977.

[15] A. Requicha and H. Voelcker. Solid modeling: A historical summary and contemporary assessment. *IEEE Computer Graphics and Applications*, pages 9–24, March 1982.

[16] A. Requicha and H. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, pages 30–44, January 1985.

[17] J. Rossignac and M. O'Connor. *SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries*, pages 145–180. Geometric Modeling for Product Engineering. Elsevier Science Publishers, 1990. M.J. Wozny, J.U. Turner and K. Preiss, editors.

[18] M. Segal and C. Séquin. Partitioning polyhedral objects into nonintersecting parts. *IEEE Computer Graphics and Applications*, pages 53–67, January 1988.

[19] D. R. Strip and M. S. Karasick. Solid modeling on a massively parallel processor. *International Journal of Supercomputer Applications*, to appear 1992.

Distribution: (UC-705)

MS	9018	Central Technical Files, 8523-2 (1)
MS	0899	Technical Library, 7141 (5)
MS	0619	Technical Publications, 7151 (1)
MS	0100	Document Processing for DOE/OSTI, 7613-2 (10)
MS	0320	Laura Lopez, 1011 (5)
MS	0951	David Strip, 2121 (10)
MS	1110	Cynthia Phillips, 1423 (5)