UCRL-LR-154063
Thesis

A Parallel Algorithm for Contact in a Finite Element Hydrocode

Timothy G. Pierce
Ph.D Thesis

June 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

A Parallel Algorithm for Contact in a Finite Element Hydrocode

by

Timothy G. Pierce

B.S (California State Polytechnic University) 1974

A dissertation submitted in partial satisfaction of the requirements for the degree
of
Doctor of Philosophy

in

Engineering — Applied Science

in the

Office of Graduate Studies
of the
University of California
Davis

Approved:

_____
Chair

_____

_____

Committee in charge
2003

Timothy G. Pierce
June, 2003
Engineering - Applied Science

A Parallel Algorithm for Contact in a Finite Element Code

ABSTRACT

A parallel algorithm is developed for contact/impact of multiple three dimensional bodies undergoing large deformation. As time progresses the relative positions of contact between the multiple bodies changes as collision and sliding occurs. The parallel algorithm is capable of tracking these changes and enforcing an impenetrability constraint and momentum transfer across the surfaces in contact.

Portions of the various surfaces of the bodies are assigned to the processors of a distributed-memory parallel machine in an arbitrary fashion, known as the primary decomposition. A secondary, dynamic decomposition is utilized to bring opposing sections of the contacting surfaces together on the same processors, so that opposing forces may be balanced and the resultant deformation of the bodies calculated. The secondary decomposition is accomplished and updated using only local communication with a limited subset of neighbor processors.

Each processor represents both a domain of the primary decomposition and a domain of the secondary, or contact, decomposition. Thus each processor has four sets of neighbor processors: a) those processors which represent regions adjacent to it in the primary decomposition, b) those processors which represent regions adjacent to it in the contact decomposition, c) those processors which send it the data from which it constructs its contact domain, and d) those processors to which it sends its primary domain data, from which they construct their contact domains. The latter three of these neighbor sets change dynamically as the simulation progresses.

By constraining all communication to these sets of neighbors, all global communication, with its attendant nonscalable performance, is avoided. A set of tests are provided to measure the degree of scalability achieved by this algorithm on up to 1024 processors. Issues

related to the operating system of the test platform which lead to some degradation of the results are analyzed.

This algorithm has been implemented as the contact capability of the ALE3D multiphysics code, and is currently in production use.

Acknowledgements

Thanks to my thesis advisor, Garry Rodrigue, for encouraging me to continue on, and for putting up graciously with more waffling than an International House of Pancakes.

Thanks as well to the other members of my thesis committee, Bill Hoover and Ping Wang, for generously offering their time to help me out.

Thanks to Evi Dube and Tom Adams, for allowing me the time to work on what has been a long process, and for finally paying me to finish it.

Thanks to Jim Maltby, my colleague and collaborator through much of the work.

Thanks to my many friends in the ALE3D group, most especially to Richard Sharp and Brad Wallin for their encouragement and high regard for my work, and to Rob Neely, who apparently knows every line of the quarter-million line code.

Thanks to the UC Davis Department of Applied Science and its fine staff, which presented me with the opportunity to stretch myself.

Thanks to my family, who make everything worth while: my mother and father, who really wanted this for me, my many wonderful and unique siblings (Fran, Dolly, Terry, Judi, Pauli, and John), and my children Claire and Travis, who frequently checked in with "Are you almost done, Dad?" Now I can say yes.

But most of all, my love and gratitude goes to my loving wife Bette. Everything else is secondary.

1 INTRODUCTION

Hydrocodes may be loosely defined as computer programs used to model large deformation transient problems that occur on a short time scale [1]. Historically there are two main classes of hydrocodes. Lagrangian codes use meshes that move with the material, while Eulerian codes have fixed meshes through which the material moves. ALE (Arbitrary Lagrangian Eulerian) methodology is a technique that attempts to incorporate the strengths of both approaches, while avoiding the weaknesses of each. In practice, ALE hydrocodes are essentially Lagrangian codes, with an added capability to adjust the mesh between Lagrange steps, either for increased resolution in certain regions or to avoid mesh tangling.

The notion of sliding interfaces, also called slidelines, slide surfaces, and contact surfaces, is native to the Lagrangian approach. If two or more separately-meshed bodies (or fluid media) are in contact, the region of contact between the two bodies is referred to as a sliding interface. The term "sliding interface" emphasizes the surfaces in persistent contact, forming a contact discontinuity. Contact discontinuities are material interfaces across which the component of material velocity orthogonal to the interface is continuous,

though densities and tangential material velocities may be discontinuous (i.e. the sides may slide across each other).

A related concept is that of contact-impact, where two bodies not in initial contact later come into contact, transferring momentum at the moment of impact. An example would be a projectile striking a target. After impact, the bodies may remain in persistent contact, as described above, or may separate.

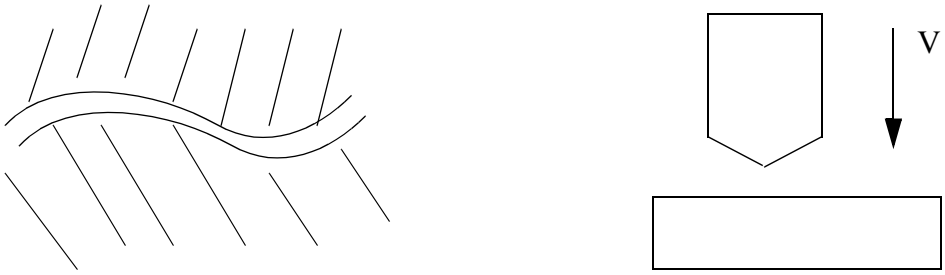


FIGURE 1-1

In Figure 1-1 (a) persistent contact between two surfaces is depicted, while in Figure 1-1 (b) the upper body travelling at velocity V is about to impact the lower body. Problems of both types may be studied with hydrocodes.

Though the above discussion assumed two or more bodies, a single deformable body may also undergo contact with itself, as in automobile crash simulations or column buckling. For simplicity, two bodies will continue to be discussed, with the understanding that they may be different regions of the same body.

The most fundamental requirement of a slide interface is that the nodes on the surface of one body not penetrate the boundary faces of another body (or itself). This is known as the ***impenetrability constraint***. If a surface node is already in persistent contact with the sur-

face of another body, then the impenetrability constraint constrains motion to be along the surface, i.e. along some particular face of the surface, unless a tensile force develops which pulls the two surfaces apart. On the other hand, if the node is not currently in contact, then its motion must be checked to see if it penetrates any surface faces of the other body during the current timestep. If so, its free motion is allowed for the portion of the timestep before the penetration. For the remainder of the timestep the impenetrability constraint is enforced.

In either case, sliding contact or impact, of all the faces on the other surface, the one where contact occurs (if contact does occur) must be determined. This has been referred to as the contact detection problem. Contact detection is potentially an $O(N^2)$ problem, if for each node of one surface the potential intersection with each face on another surface must be considered. For a parallel implementation this also creates the problem of massive global communication as the positions and velocities of all faces are sent to all processors each cycle.

Several solutions have been offered to reduce the order of the contact detection, with varying degrees of success. These will be summarized in Section 2.4. This thesis will describe a particular approach to the parallel contact detection problem, and its implementation in a modern multi-physics hydrocode (ALE3D). The approach has some restrictions, described in Section 2.4.4. Fortunately these limitations are irrelevant to the class of problems typically addressed by ALE3D.

Once contact has been detected, i.e. nodes on one surface are paired with faces on the other surface, contact enforcement can be performed. Several viable approaches to contact

enforcement are described in Section 1.3. The particular method used by ALE3D is described in much greater detail in Section A.11.

1.1 Overview

The first chapter of this thesis provides some general background for the setting in which the work was done. Specifically it surveys the development over the years of a class of software programs called hydrocodes, discussing variants that have evolved. It then takes a look at the general problem of contact, from a computational perspective.

In the second chapter the issues involved with parallelism are brought up, and several modern approaches to parallel contact are discussed.

The third chapter presents the method which is the primary original contribution of this work. It discusses the problem of evolving a communication pattern scalably, and presents a solution that involves three different communication topologies, each of them local and scalable it itself.

Chapter 4 presents a number of example applications, together with their performance on a large parallel machine. These applications are typical of the kind of problems to which the parent code, ALE3D, is applied.

Chapter 5 provides a brief summary and discussion of futute directions.

The first appendix provides a mass of hard-core detail in the implementation of the basic contact algorithm in ALE3D, and should primarily be of use to someone tasked with working on ALE3D itself, or a similar code.

The second appendix surveys extensions to the design of chapters 3 and and appendix A made to the ALE3D implementation to accommodate complex real-life applications. These are described at a higher level than the discussion of the appendix A.

1.2 Hydrocodes

1.2.1 Etymology of the Term "Hydrocode"

The term "hydrocode" came into being in the 1950s at the national defense laboratories to describe a type of software program based on fundamental conservation laws, which could be used to study the shock dynamics of materials under extreme pressure: i.e. explosions and hypervelocity impact. The term "hydrodynamics" is synonymous with "fluid dynamics", and refers to the branch of physics that studies the forces on and motion of fluids. Hydrodynamics is itself a branch of continuum mechanics. The original expression "hydrodynamics simulation code" was shortened to "hydrocode" [2], where "code" is synonymous with "software program".

It was never intended that hydrocodes be used to study only what are commonly thought of as fluids. Rather, in extreme pressure regimes the forces of compression and expansion are so much greater than the resistance of the material to shear stresses that shear stresses were ignored as a simplification, even for materials such as metals with significant "strength", or resistance to shear, in more typical regimes. This simplification leads directly to the hydrodynamic equations of motion, where pressure replaces the more general stress tensor.

Hydrocode technology soon evolved beyond this simplification with codes such as TENSOR [3] and HEMP [4], as designers began to explore later stages of energetic events, at which time strength of materials did become significant. Thus the hydrodynamic assumption was replaced by more realistic material models. However, the name stuck, and is still in common use today, confusing newcomers to the field.

Perhaps a more appropriate name, but one that is much less often seen is the term "wave code", emphasizing the central role in this technology of the accurate modelling of shock waves.

1.2.2 A Brief Survey of Hydrocode Evolution

The governing equations for hydrocodes are hyperbolic and nonlinear in nature. This implies that solutions can include discontinuities along "characteristics", or wave fronts. One approach to handling these discontinuities is to track them directly, via the Method of Characteristics [5]. This method works well for simple situations, but as shocks multiply and interact, becomes intractable. von Neumann and Richtmyer [6] developed a method of introducing an artificial "viscosity" into the equations which spread the shock front out over several grid zones, removed unphysical oscillations at the shock front, and still maintained good accuracy for the entire solution. Shocks of any complexity can thus be handled automatically without any explicit tracking. This method of artificial viscosity, with suitable refinements for higher dimensions, has become a standard approach in hydrocodes.

In the late 1950’s one dimensional codes (in spherical and cylindrical geometry) were developed at the national laboratories to study weapons and weapon effects. In one dimen-

sion the Lagrangian approach (Section 1.2.6) was clearly the method of choice. The primary difficulty with the Lagrangian approach, mesh tangling, only appears for two or higher dimensions. The KO code [4] was typical of this class of 1D Lagrangian codes.

As machine capacity grew, 2D codes (which better represented the underlying problems) became more feasible. In 1964 Wilkins [4] published a description of the model for a 2D (with axial symmetry) Finite Difference Lagrangian code, HEMP, which included an elastic-plastic strength model and slide interfaces. Another influential code of the same era with similar capacities was TENSOR [3].

By this time the problem of mesh distortion in the Lagrangian approach had become painfully familiar to researchers. One way to get around these problems, at least for a while, is to rearrange the mesh either by hand or with semi-automatic tools, during the course of a run, in order to allow the run to proceed further. This is known as a ***remap***. The remap has two phases. In the ***relaxation*** phase the positions of the internal nodes and corresponding zones are moved to provide for a more regular zoning. In the ***advection*** phase the extensive physical quantities such as mass, momentum and energy are reapportioned to the new zones in such a way that the overall quantity is strictly conserved. Little, however, could be done for material boundaries that themselves became severely distorted, as in turbulent flow or projectile impact.

To overcome this problem Eulerian, or fixed grid, hydrocodes were developed. The early Eulerian hydrocodes such as OIL [7] were an outgrowth of a technology called Particle-In-Cell, or PIC. The PIC method used a fixed mesh through which discrete particles moved. The particles were assigned mass, and density was simply the sum of mass of the

particles in a cell, divided by cell volume. In the PIC approach the momentum and energy equations (eq. 1-2., eq. 1-3) were solved for each cell ignoring the transport term

$v_i \frac{\partial}{\partial x_i}[\Phi]$ (where Φ is the transported energy or component of momentum)

(Section 1.2.3). Then the particles were moved (advected) based on the computed velocities from the first step. Finally the momentum and energy carried by the particles was used to update cell velocity and internal energy [8]. From this basis the first continuous Eulerian codes were built by replacing the discrete particles with a continuous density within the cells.

The 1970's saw the advent of widespread usage of the Finite Element method (FEM). As originally formulated, FEM when applied to dynamic problems led to a semi-discretized set of differential equations that were coupled in two ways: via the "mass matrix" and the "stiffness" matrix. Because of this coupling, advancement of the solution each cycle required the solution of a large system of equations. For the small timesteps required by hydrocodes to capture detailed shock behavior, this was prohibitive. However, Belytschko [9], [10] showed that above approach could be replaced by a lumped mass matrix and explicit time integration, and still produce accurate results for shock physics. This opened the door for the use of the Finite Element method for hydrocodes. Early efforts in this direction were HONDO [11], DYNA [12], and EPIC [13]. The latter two have spawned families of codes which are still widely used today.

Over the ensuing years the various classes and families of hydrocodes have undergone a number of evolutionary improvements and extensions, stimulated largely by the needs of the user base of the given code. For instance, structural elements (e.g. beams and shells)

were added to support structural analysis, 3D versions of the more successful codes were developed, sophisticated interface trackers written to shore up a weakness in that area of Eulerian codes, and various mergers of technologies such as Coupled Eulerian Lagrangian (Section 1.2.8) and Arbitrary Lagrangian Eulerian (Section 1.2.9) were developed. Meanwhile the implementations were constantly evolving to take advantage of emerging hardware technology, such as vector machines and shared and distributed memory parallel machines.

1.2.3 Governing Equations

The fundamental equations governing dynamic behavior of continuous media are derived from conservation laws; specifically, the conservation of mass, momentum, and energy. Expressed in differential form, these equation may be written as:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(\rho v_i) = 0 \qquad \text{Conservation of Mass} \qquad 1\text{-}1$$

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = f_i + \frac{1}{\rho}\frac{\partial \sigma_{ij}}{\partial x_j} \qquad \text{Conservation of Momentum} \qquad 1\text{-}2$$

$$\frac{\partial e}{\partial t} + v_i \frac{\partial e}{\partial x_i} = f_i v_i + \frac{1}{\rho}\frac{\partial}{\partial x_j}(\sigma_{ij} v_i) \qquad \text{Conservation of Energy} \qquad 1\text{-}3$$

In these equations, ρ is the density as a function of position (i.e. mass/volume), v_i is material velocity, f_i is the body force per unit mass, σ_{ij} is the stress tensor, and e is the specific energy.

These equations may be derived by considering an infinitesimal control volume, and applying the principle that the change of a quantity (e.g. mass) within the control volume

is dictated by the amount of the quantity swept into and out of the control volume by the movement of the material, plus any source or sink of the quantity within the control volume. Many standard texts contain a derivation of the conservation equations. For instance, see Panton [14].

Equations 1-1 to 1-3 are known as the Eulerian formulation of the conservation equations. If a special type of derivative:

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + v_i \frac{\partial}{\partial x_i} \qquad 1\text{-}4$$

is defined (usually called the material derivative, substantial derivative, or total derivative) then the equations can be written in the slightly simpler form:

$$\frac{D\rho}{Dt} + \rho \frac{\partial v_i}{\partial x_i} = 0 \qquad \text{Conservation of Mass} \qquad 1\text{-}5$$

$$\frac{Dv_i}{Dt} = f_i + \frac{1}{\rho} \frac{\partial \sigma_{ij}}{\partial x_j} \qquad \text{Conservation of Momentum} \qquad 1\text{-}6$$

$$\frac{De}{Dt} = f_i v_i + \frac{1}{\rho} \frac{\partial}{\partial x_j} (\sigma_{ij} v_i) \qquad \text{Conservation of Energy} \qquad 1\text{-}7$$

Equations 1-5 to 1-7 are known as the Lagrangian formulation of the conservation equations.

Mathematically there is no difference between the two forms: it is just a matter of definition. However, conceptually and computationally the distinction is a springboard for two fundamentally different approaches. In the Eulerian approach attention is focused on a fixed control volume. Flux into and out of the control volume is represented by the "trans-

port term" $v_i \frac{\partial}{\partial x_i}$. In the Lagrangian approach the conceptual focus is on a control volume that moves with the flow. In this frame of reference the convective part of the derivative disappears, and we are left with just the total time derivative $\frac{D}{Dt}$ (see eq. 1-4).

A helpful analogy is that of a boat on a river. The boat is the control volume. The Eulerian boat is anchored, and the river flows around it with some velocity, whereas the Lagrangian boat is adrift and moves at the speed of the surrounding river.

The conservation equations, in either form, do not constitute a complete set of solvable equations. For one thing a problem domain must be specified together with boundary conditions and initial conditions. But also, not enough is said about the stress tensor σ_{ij}. In order to produce a complete set of equations, the stress must be re-expressed in more fundamental unknowns related to displacements. Typically this is done through constitutive relations and equations of state.

A constitutive law ties the stress for a specific material to the strain, or relative displacement of nearby points, strain rates, and various "history variables" of the material such as damage and work-hardening. A simple example of a constitutive relation is Hooke's Law for elastic materials:

$$\sigma_{ij} = C_{ijkl}\varepsilon_{kl} \qquad 1\text{-}8$$

where the strain tensor ε_{kl} is defined as:

$$\varepsilon_{kl} = \frac{1}{2}\left(\frac{\partial u_k}{\partial x_l} + \frac{\partial u_l}{\partial x_k}\right) \quad 1\text{-}9$$

and u_k is a displacement of a material point from its equilibrium position. Thus, in this case, stress can be replaced by an expression in terms of displacement:

$$\sigma_{ij} = \frac{1}{2}C_{ijkl}\left(\frac{\partial u_k}{\partial x_l} + \frac{\partial u_l}{\partial x_k}\right) \quad 1\text{-}10$$

The equation of state expresses the relation between thermodynamic quantities. Typically it is used to model the relationship between the energy e, the pressure P, and the density ρ. The pressure P is a constituent of the stress tensor ($P = -\frac{1}{3}[\sigma_{11} + \sigma_{22} + \sigma_{33}]$), and so the equation of state helps eliminate stress from the unknowns in constructing a complete set of equations.

1.2.4 Time Integration

The conservation equations can be written in the general form

$$\frac{\partial f}{\partial t} = G(f, ...) \quad 1\text{-}11$$

where G may involve spatial, but not time derivatives of the scalar or vector function f. Consider the time interval from t^i to t^{i+1}. The value of G will in general vary over this time interval. The "best guess" of its mean value is some combination of its value at the two endpoints: i.e.

$$\varepsilon G^i + (1 - \varepsilon)G^{i+1} \quad 1\text{-}12$$

where G^i is G(f, ...) calculated at time t^i, G^{i+1} is G(f, ...) calculated at time t^{i+1}, and $0 \leq \varepsilon \leq 1$.

Assuming that the calculation has proceeded to time t^i, then G^i can be explicitly calculated; its functional argument values are known. However, if $\varepsilon < 1$, then evaluation of G^{i+1} requires values of the argument that are not yet known. This results in a system of equations (when all grid points are taken into account) that must be solved simultaneously. This is known as implicit time integration.

Implicit time integration (usually) has the virtue of unconditional stability. That is, no matter how large the time step, the solution will not be swamped by exponentially growing error terms. Of course, for a large timestep the results may be totally inaccurate, but that is a separate issue. For the same timestep as an explicit integration, the implicit integration will in fact be more accurate. However, the cost of an implicit scheme with its requirement for the solution of a system of equations, quite possibly iteratively for nonlinear problems, is far more expensive than the simpler explicit time advancement.

This cost may be acceptable when large enough timesteps can be taken. However, for the phenomena typically studied with hydrocodes, a separate consideration drives the choice of timestep. Since a primary goal is to accurately model shock waves traversing the material, the timestep must be short enough that the wave will not go further than the smallest dimension of the smallest zone in a single cycle, or it will be invisible to the code. Typically this results in timesteps on the order of microseconds. This restriction is essentially the same as the Courant stability restriction, so that nothing is lost by ensuring that the

timestep obey the Courant condition. Thus there is no good reason not to use explicit time integration, and that is what virtually all hydrocodes do.

That said, it is true that several of the most popular hydrocodes either have an implicit integration mode, or are part of a family of codes that also supports implicit time integeration. This is indicative of the widening applicability of hydrocodes as time has gone on, making them competitive in fields such as Structural Dynamics. For instance, DYNA3D has a sister implicit code NIKE3D, while ALE3D can be operated in "implicit hydro" mode. This feature can, for instance, be used in modelling phenomena that change slowly over a long time, followed by sudden explosive action, such as an explosive in a fire that slowly heats it to a detonation point. Implicit time integration coupled with long timesteps would be used over the long incubation period, then switched to explicit time integration and short (microsecond order) timesteps to model the actual explosion.

1.2.5 Finite Difference vs Finite Elements

The finite difference approach to discretizing the conservation laws starts directly with the strong form (Eq 1-1 to 1-3) of the conservation equations. These essentially describe local behavior: i.e. relationships between dependent variables and their derivatives at any given point $\overline{X}$ and time t. In the finite difference discretization, these local relationships are reduced from a continuum to a discrete set of grid points, and the dependent variables are tracked only at these points. Derivatives are replaced with finite differences based on values in the local neighborhood of the grid points.

A wide variety of finite difference formulas may be applied. For a fixed rectangular grid a simple forward, backward or central difference can replace first order derivatives, etc.

However, for more complicated grids such as a moving Lagrangian grid, a better approach is to use an approximation of the derivative based on a line integral about the surrounding cells [15].

It should be noted that, in discretizing nonlinear conservation equations, care must be taken that the approximation itself is still conservative. When the conservation equations are formulated, an essential idea is that the time rate of change of a quantity in an infinitesimal volume is the result of the flux of that quantity into the volume, plus source or sink terms internal to the volume. This balance must be maintained in the discretization, by evaluating the finite differences in such a way that the flux passing out of any cell is balanced by the same quantity passing into the adjacent cells. See for instance Hirsch [16].

The Finite Element Method (FEM) is conceptually a more global approach. The solution (for example, displacement from initial position as a function of time) is approximated by a continuous, piecewise smooth function, where each smooth patch is defined over a single cell, and the patches are "sewn" together continuously, but not necessarily smoothly, at the cell boundaries. Each patch is typically represented by a low-order polynomial function, such as a bilinear or trilinear surface. The continuous solution is dictated by the values of the solution at the nodes of the cell. Another way of saying this is that the nodal values are the "degrees of freedom" in the space of possible solutions constructed in this way.

The weak form of the governing equation is then used to find an optimal approximation of the above type to the "true" solution. For example the momentum equation, in its weak form, may be written as:

$$\int_{\Omega} T_i \left(\frac{Dv_i}{Dt} - f_i - \frac{1}{\rho} \frac{\partial \sigma_{ij}}{\partial x_j} \right) d\Omega = 0 \qquad 1\text{-}13$$

where the equality must hold for all T_i in the space of permissible functions (i.e. functions that satisfy certain boundary conditions, as well as conditions on differentiability).

Clearly if the displacement solution satisfies the strong form of the momentum equation it will also satisfy the weak form, since the term of the integrand in parentheses will be identically zero. But the value of the weak form lies in the possibility that useful solutions may be found that do not strictly satisfy the strong form, due to non-existence of derivatives at certain points, such as at patch boundaries in the solutions proposed above.

The weak form proves to be a useful starting point for finding a globally optimal approximate solution. The proposed, patched-together displacement field, with coefficients corresponding to the nodal displacements still unknown, is "plugged into" the integrand term $\frac{Dv_i}{Dt} - f_i - \frac{1}{\rho} \frac{\partial \sigma_{ij}}{\partial x_j}$. In order to do this, σ_{ij} must be expressed in terms of the displacement, via the constitutive stress-strain relation. V_i is itself a time derivative of displacement. Then this expression is multiplied independently by each of a set of functions T_i. In the popular Galerkin procedure [17], the trial functions are just the basis functions from which the approximate surface was patched together. Then the product is integrated numerically over the domain Ω. This leads to a set of equations that still contains time derivatives of the unknowns (called a semi-discretization). The set may then be integrated either explicitly or implicitly in time to approximate the dynamic behavior of the system.

1.2.6 Lagrangian Hydrocodes

In a Lagrangian hydrocode the mesh is fixed to the material and moves with it. Mesh nodes may be considered as representative material points within and on the surface of the material. The zones in the mesh move and distort, yet always contain the same material. Thus the mass in a Lagrangian zone is constant, though the volume may vary.

Figure 1-2 shows a metal bar before and after collision with a fixed wall. Each zone changes shape and potentially volume, but contains the same material at both times.

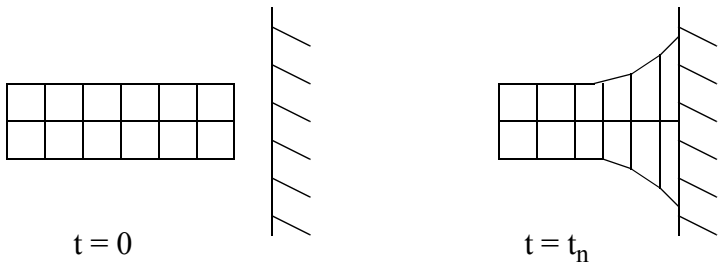


FIGURE 1-2

The mechanics of the Lagrangian formulation are analogous to Newtonian particle mechanics. The force at the mesh nodes is a resultant of the stress field in the neighborhood of the node (i.e. in adjacent zones). The mass of the body is treated as though it were concentrated at the nodes. Together the mass of, and force on, a node result in an acceleration of the node which, integrated over the time step, produces an incremental displacement. The new displacements of the nodes are then fed back into the material model to produce the updated stress field in the material, and the next step begins.

The Lagrangian approach provides several benefits. It is simple in that material never leaves or enters mesh cells. For the same reason it provides an excellent representation of material interfaces and boundaries. There is no need for "mixed cells", i.e. cells containing

multiple materials, with all the concomitant complexity of implementation. It also provides a natural way to track material history, such as damage or work hardening, which are properties of specific regions of the material, and may be recorded as zonal data. Such "history variables" are essential to realistic material modelling for many materials.

The chief drawback of the Lagrangian method is its tendency to produce mesh distortion, or "tangling". For instance, consider again Figure 1-2. At the later time, some of the cells have become compressed in one direction while expanding in another. For codes with explicit time integration (see Section 1.2.4), the time step is limited to the time it takes a signal (such as a wave) to cross the smallest cell of the problem. This is the well-known Courant (or CFL) Condition [41]. If it is violated the solution becomes unstable. As the cells become compressed the timestep must be reduced to satisfy the Courant condition. It is not unusual for the type of problem addressed by hydrocodes to produce so much compression that the time effectively stops advancing before the time of interest is reached.

Another typical case with still greater distortion occurs when a cell becomes inverted:

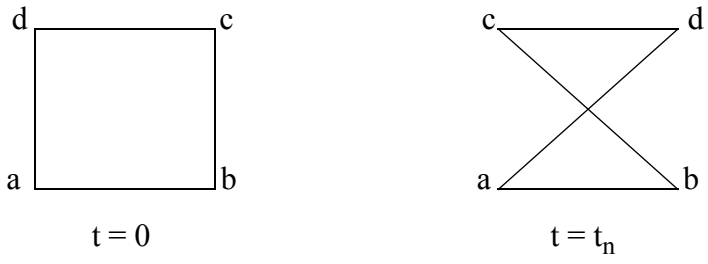


FIGURE 1-3

This will typically result in the calculation of a negative volume for the cell, followed shortly by a program "crash".

1.2.7 Eulerian Hydrocodes

The most obvious characteristic of an Eulerian hydrocode is that the mesh defining the problem does not move from one timestep to the next. A mesh is initially overlaid on the entire region over which the material will move during the course of the simulation.

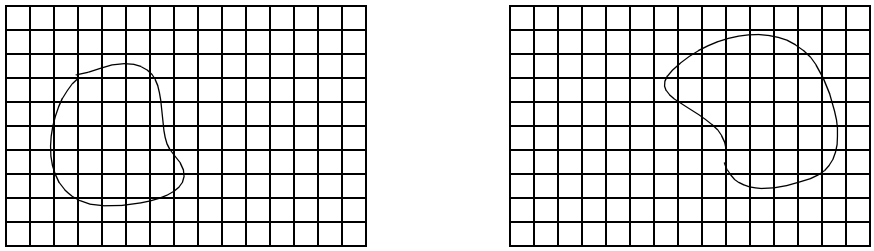


FIGURE 1-4

As time proceeds the body moves and potentially deforms through the mesh, leaving some zones and entering others.

One difficulty encountered with this approach is the tracking of boundaries and material interfaces. In Figure 1-4 a clear border is drawn between a body and its surroundings, but this gives an incorrect impression. Since Eulerian codes track material movement across zone boundaries, it is really only known that the material fills some zones, is absent from others, and is partially present in other ***mixed*** zones. The volume fraction of the material in mixed zones can be tracked, but is in a sense "constant" throughout the zone, so that the border of the material object is only known to the resolution of a zone. Figure 1-5 gives a more accurate picture of the actual simulation. The zones marked with "+" are pure zones

filled with the object material. The zones marked "o" are mixed: part object and part "void" material. The unmarked zones are again pure zones, filled with void material.

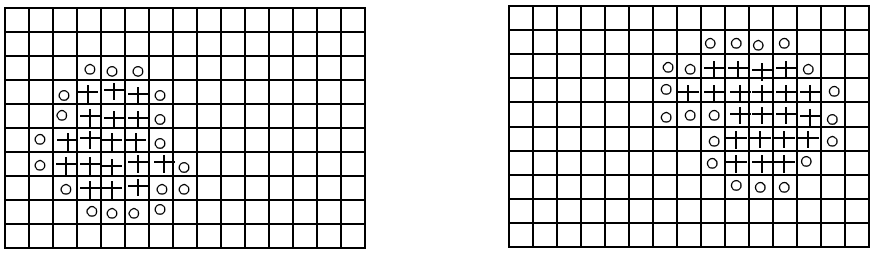


FIGURE 1-5

The use of mixed zones allows another problem to appear. As soon as material enters a zone in any fraction, it is present everywhere in the zone, so that the next cycle it may be convected to yet another zone. This rate of convection is non-physical, and causes the body to be smeared out over the whole problem space. To combat this problem Eulerian hydrocodes use a preferential advection scheme for mixed cells. If a "downwind" cell is pure (i.e. single material) and its upwind cell is mixed, then the material the mixed cell shares with the pure cell is advected preferentially out of the mixed cell, till it is used up.

A main strength of the Eulerian fixed grid approach is seen in high velocity impact simulations. At the meeting between two materials extreme deformation may occur. As discussed in the previous section, if the mesh is tied to the material it can become badly distorted, leading to extremely small timesteps or even a program "crash". Eulerian hydrocodes do not exhibit this drawback, since cell geometry is fixed

The term Eulerian hydrocode is a bit misleading. In the general field of Computational Fluid Dynamics, the Eulerian approach, which is nearly universal, is taken to include the

idea that the velocity field of the fluid is the fundamental unknown. Individual "particles", as tracked in the Lagrangian approach, are not present in the formulation. Only instantaneous velocities at various fixed points are relevant. However, the technology of the Eulerian hydrocodes is much closer to the Lagrangian method. Each cycle the material is advanced based on the stress in the local environment. The mesh temporarily moves with the material, creating a new mesh slightly offset from the old mesh. The quantities on the new mesh are then mapped back into the old mesh in such a way that mass, momentum and energy over the entire system is conserved.

1.2.8 Coupled Eulerian Lagrangian Hydrocodes

In some applications, especially those involving fluid-solid interactions, it is clearly advantageous to model parts of the problem with a fixed Eulerian grid, and other parts with a moving Lagrangian grid. As a simple example, consider an inflating balloon. It makes sense to model the expanding, possibly turbulent gas inside the balloon with an Eulerian grid, while modelling the skin of the balloon with a Lagrangian grid, possibly using shell structural elements. In this problem, the Eulerian grid covers all space that the balloon will eventually expand into, but the ***active*** part of the grid is delimited by the edge of the Lagrangian grid, which forms irregular zones along the edge of the Eulerian grid.

Coupled Eulerian Lagrangian (CEL) hydrocodes support this capability. The first CEL code [15] was written in the early 1960s, and was used to model 2D blast simulations. Some more recent examples of CEL codes are MSC/DYTRAN [18], MSC/PISCES [19], HULL [20], and the coupling of CTH with EPIC through ZAPOTEC [21].

Generally a CEL hydrocode has three modules: Lagrangian, Eulerian, and Coupling. At a given timestep t^i, the known state of the material, including the pressure of the Eulerian region acting on the Lagrange regions, is used to update the position of the Lagrange meshes to the next timestep t^{i+1}. The Coupling module then determines the new active area of the Eulerian mesh. Given the domain of the active Eulerian mesh at t^{i+1},the equations of motion are used to advance the state of the material in the active mesh from t^i to t^{i+1}. Finally a second phase of coupling determines the new pressures at t^{i+1} acting on the Lagrange meshes, and the next timestep may begin.

1.2.9 Arbitrary Lagrangian Eulerian Hydrocodes

Arbitrary Lagrangian-Eulerian (ALE) hydrocodes are an extension of the Eulerian hydrocode approach to non-fixed meshes. As discussed in Section 1.2.7, the Eulerian hydrocode first takes a Lagrange step, then remaps the mesh back to its original (Eulerian) position. An ALE code does the same, but has greater flexibility in deciding what mesh to map back to. It may do no remap at all, in which case it behaves like a Lagrangian code. It may map back into the Eulerian mesh. Or it may choose yet a different mesh, in the interest of maintaining zone regularity internal to the body, while preserving the boundaries as computed.

The simplest version of ALE, called "simple" ALE or SALE, restricts ALE mesh movement to within single materials [22]. The material boundaries are still treated in a strictly Lagrangian manner. The advantage of this is that it avoids the need for mixed zones, which add considerable complexity to the code. If the boundary between materials becomes distorted, then SALE suffers from the same mesh tangling problems as a pure Lagrangian code.

Multimaterial ALE, capable of supporting mixed zones, is more general than SALE, in that it removes the restriction on advection through internal material boundaries. The external boundaries (including slide interfaces) are still treated as Lagrangian. In other words, the limits of the mesh coincide with the limits of the bodies being modelled.

Some recent multimaterial ALE codes are CALE [23], CAVEAT [24], ALEGRA [25], and ALE3D [26].

1.3 Contact

The earliest scientific studies in contact were concerned with friction between rigid bodies. Leonardo Da Vinci studied frictional effects in the 15th century. They were put in the form of laws by Amontus in 1699, and further developed by Coulomb in 1781 [27].

However, these early studies did not consider deformable bodies, specifically bodies deformed by the force of the contact itself. Contact problems in elasticity were first considered in the 19th century, with early work done by Poisson, Saint-Venant, Voight, and Hertz [27]. Hertz [28] calculated the indentation of spherical elastic bodies into an elastic plane, and related geometries. This class of problems is now referred to as Hertzian contact problems.

A similar class of contact problem, known as the Signorini problem, consists of an elastic deformable body of arbitrary shape brought into contact with a rigid foundation. The actual surface of contact is not known ahead of time, but is computed as part of the solution. This problem was first studied by Signorini [29]. Even problems of such relative simplicity as Hertzian and Signorini problems prove to be a formidable challenge to solve analytically, and are still used today as prototype problems for new methods of solution.

In a numerical approach to the solution of contact problems, there are two main schools of thought. The first approach treats contact formally as an inequality. Consider the two bodies in Figure 1-6.

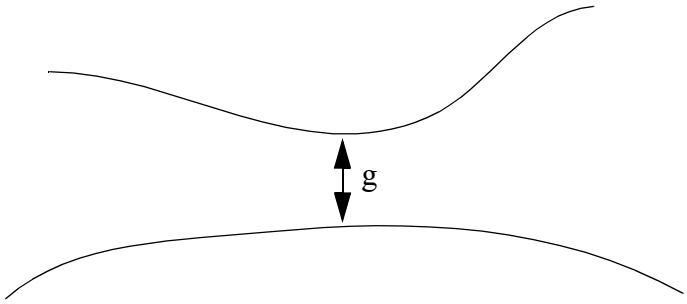


FIGURE 1-6

g (for gap) is the distance between the two bodies at one point along their boundaries. The fundamental constraint, that of impenetrability of two bodies, is written as $g \geq 0$.

This inequality is combined with the governing equations of the two bodies (note that it couples those equations) to form a complete mathematical description of the system. Function Theoretical methods, in particular the Calculus of Variations, is now employed in converting this system to its variational or "weak" form and solving it. In this conversion, treatment of the inequality is particularly problematical.

The result is that the system of differential and algebraic equations and inequalities is reformulated as a minimization problem: the function which minimizes an integral functional is the correct solution to the original problem.

Finally, a numerical method is applied to find an approximate representation of the minimizing function.

There is an enormous body of work supporting variational techniques in general, including results about the existence and uniqueness of solutions. The Finite Element method itself is grounded in the variational approach, and thus benefits from its solid theoretical underpinnings.

It is not the purpose of this work to discuss the variational approach to contact problems, which requires a substantial amount of mathematical machinery, but simply to point out that such a body of work exists. A good starting reference for this work is [27].

The second approach to modelling contact numerically is to treat the contact in a corrector step of a predictor-corrector formulation. During the predictor step the separate position of each involved body is calculated independently, from its governing equations, without any consideration of contact. Thus there is no coupling of bodies during the predictor step. During the corrector step the positions of the separate bodies are adjusted as necessary to enforce the impenetrability constraint, and possibly other conditions such as continuity of momentum normal to the contact interface. This is the approach taken by hydrocodes.

Three methods have emerged for handling contact within hydrocodes. These are usually referred to as the Lagrange Multiplier method, the penalty method, and the "hydrocode" method, so-called because it was the method first employed in early hydrocodes. All three methods are in use in present day hydrocodes.

Figure 1-7 depicts a sliding interface with most of the information abstracted away. For one side, all that is shown is one node, s_1. For the other side, only one face is shown, the face bounded by m_1, m_2, m_3, and m_4, and which is the closest face to s_1. Contact enforce-

ment in this case, for now ignoring the possibility that the surfaces may separate, amounts to ensuring that s_1 lies on the surface formed by m_1, m_2, m_3, m_4.

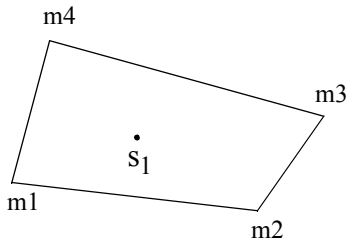


FIGURE 1-7

The Lagrange Multiplier method, which in its original form requires an implicit time integration method (Section 1.2.4), enforces the constraint by adding an equation to the set of equations that must be satisfied at the new timestep. To balance this it adds a new unknown, the "Lagrange multiplier", which turns out to equal the force required to satisfy the constraint. The added equation expresses algebraically that the new position of s_1 must lie on the surface bounded by m_1, m_2, m_3, m_4. Thus it links these new unknown positions together.

The penalty method avoids the addition of any new unknowns. Instead the motion of the nodes, neglecting initially any contact forces, is calculated as a predictor step. If this results in penetration of a contact face by a contact node of the other side then, as a corrector step, a nodal restoring force is applied which is perpendicular to the face, and proportional to the amount of penetration. It is as though there is a spring between the node and the surface, but only when the node penetrates the surface.

The penalty method is a simple, cheap method of contact enforcement for many structural problems. However, neither it nor the Lagrange multiplier method work well in shock wave applications where pressures greatly exceed the yield strength of the material. In this regime it has been found that the original method of the 1960's hydrocodes, with a number of improvements in the area of symmetry, void opening and closing (or impact) and more complex geometries such as intersecting and overlapping slides, is still the most accurate way to model shock waves as they cross slide interface boundaries.

The hydrocode method treats the two sides of the slide interface as though they were merged, during calculation of acceleration normal to the surface. That is, the mass from the two sides is lumped locally into a center-of-mass system, and the system accelerated by pressure exerted by each side on the other. In the absence of friction, tangential velocity for each node is treated independently.

A variant of the hydrocode method is used in the implementation described in the following chapters. It is described in more detail in Section A.11.

1.4 Examples

This section gives two examples of problems which involve contact surfaces. Ale3d was employed to solve both of them.

1.4.1 The Roller Problem

In this problem an aluminum ingot is fed between two steel rollers, during which its thickness is reduced by approximately 3% for this run. The contact surfaces are between the top and bottom of the ingot and the rollers, and between the top and bottom of the ingot and the table. The ingot is fed into the rollers with about the same velocity as the linear rolling speed. Once the ingot is in contact with the rollers frictional force pulls the ingot through. The thermal behavior of the coupled ingot-roller system is also modelled.

Figure 1-8 shows the computation just before the ingot touches the rollers, and after it is halfway through.

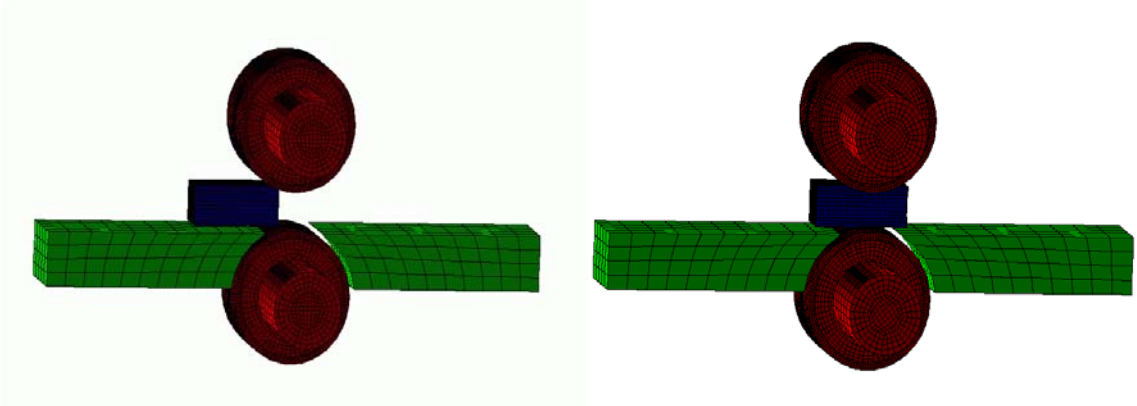


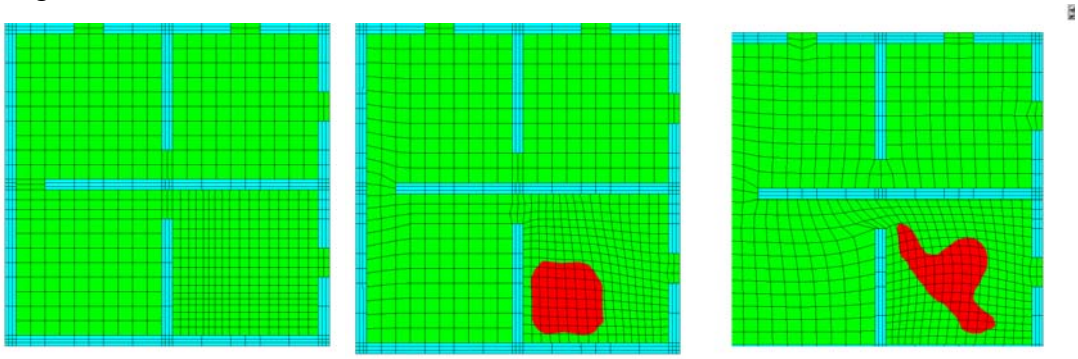
FIGURE 1-8

1.4.2 Explosion in Bunker

In this problem a four-room concrete bunker is modelled, together with the atmosphere inside the bunker. An explosion is set off inside the bunker, and the hot gases expand. The undisturbed air and the hot gases share the same mesh, and are represented by mixed

zones (Section 1.2.7). The hot gases are advected into the initially pure air zones. This problem uses the full ALE functionality. The gas-wall interface is represented by 29 separate contact surfaces.

The following figure shows the state at three different times. In these pictures a slice orthogonal to the vertical axis is taken.



t = 0.0 t = 150.0 usec t = 600.7 usec

FIGURE 1-9

2 PARALLEL CONTACT

2.1 Parallel Computers and Programming Paradigms

Over the years the evolution of computer systems has been relentless and rapid. In some cases the advances have been seemingly without cost to software developers. The newer machines were simply faster, memories larger, the operating systems more full-featured and easier to use. In other cases, to use a new class of architecture efficiently meant software changes, in some cases total rewrites, of leading-edge applications. For instance, when vector machines, almost synonymous with Cray computers, arrived on the scene, it was necessary to "vectorize" the code: i.e. arrange loops wherever possible so that the compiler could recognize the independence of separate iterations, and thus could pipeline the arithmetical operations. To do so led to improved performance by better than an order of magnitude. Vector machines achieve their performance through a form of parallelism. Arithmetical operations (e.g. floating point multiplications) are broken down into a number of substeps, and a separate functional unit designed to specialize in each substep. The overall arithmetic unit is then organized into a pipeline, and the functional units are kept busy simultaneously, each working on its stage with a different set of operands. So while

it may take 16 cycles to produce a given result, a new result is ready every cycle thereafter, for a parallel speedup of 16 (neglecting latency in "priming" the pipeline).

One fortunate aspect of the work of vectorizing applications was that it could be done incrementally. The most time-intensive loops could be tackled first, for the greatest gain, then more of the code gradually vectorized until it became not worth the effort to go further.

There is a fundamental limit to how far this style of heterogeneous parallelism can be taken, since an arithmetic operation can only be practically split into so many substeps. To grow beyond this it was necessary to seek parallelism at a higher level than the single instruction. This became possible through a new type of architecture which had multiple independent CPUs which shared the same memory. Today this type of machine is called a symmetric multiprocessor, or SMP, or simply a shared-memory machine.

With these machines it is possible to take advantage of the parallelism in loops at a higher level, by dividing the number of loop iterations into subsets, and assigning each subset to a different CPU. If the loop count is large (in the thousands or millions) and if each iteration is independent, as is also required for vectorization, then many CPUs can work in concert to achieve the computation.

This style of parallelism was sometimes called microtasking. It is still prevalent today on shared-memory machines, and is typically implemented via "threads", though threads are not limited to loop-level parallelism.

In processing the iterations of a loop the same set of instructions (i.e. the loop body) are executed over and over, but with different data. This observation became codified in the term SIMD (Single-Instruction-Multiple-Data) and led to a class of machines that had many processing units that all executed the same instruction simultaneously, each with its own data. SIMD machines enjoyed some success and popularity in the 1980s and 1990s, but ultimately proved to be too inflexible outside the loop level, and so did not deliver on their promise of performance for more complex real-world application.

It is difficult to design a machine with many CPUs sharing the same memory. Synchronization and bandwidth of memory access become bottlenecks. Practical SMPs today are on the order of 4-16 CPUs, although larger machines have been built. The memory bottleneck can be overcome by giving each CPU its own piece of the total memory, effectively partitioning the memory. Each CPU can then only directly read its local memory. In order to access data in another CPU's memory, the other CPU must read it, and then send it in a message. This kind of computer is called a distributed memory machine. A program that runs in parallel on a distributed memory machine is called a message-passing program. Unlike an SMP there is no inherent size limitation to distributed memory machines. The designer can simply keep adding CPUs and memory, hooking them together in some kind of communication network.

However, the simplicity of the hardware is balanced by the complexity of the software required to efficiently use the machine. Converting a serial code into an efficient scalable message passing program, when possible at all, is a major undertaking, much more labor-intensive than vectorizing or microtasking (threading) a serial code. Frequently the best

course is to start from scratch, retaining perhaps the computational kernels of the serial code.

A prevalent method of parallelizing simulations such as hydrocodes for distributed memory machines is spatial domain decomposition. Since the overall problem is discretized as a mesh in space, the mesh can be divided into more-or-less compact connected submeshes, and each submesh assigned to a different processor. That processor then has primary responsibility for the data in its submesh, such as calculating stresses, advancing nodal positions, etc. When necessary the processor communicates via message-passing with other processors, typically processors assigned to adjacent submeshes.

This style of development allows the developer to retain the original serial approach (if there is one) to the greatest extent possible, since the work performed in each submesh is for the most part the same work as performed on the whole, but for only a subset of the data. Of course, the parts must still be "glued" together with communications, and in this case the "devil is in the details".

In the past several years there has been a strong trend for the most powerful high-performance computers to combine the SMP and distributed memory models; that is, to combine a cluster of SMPs, each with a moderate number of processors, via a high-performance communication network. This gives software developers some added flexibility, as they can still use the machines as pure distributed memory (by ignoring the shared memory within the SMP and still employing message-passing, even to other processors on the same SMP) or they can capitalize of the shared memory by threading the code on each SMP so that multiple processors work in a microtasked fashion on a single submesh.

Employing both paradigms is optimal, but involves more work on the part of the implementer.

2.2 Scalability

Scalability refers to the performance behavior of an algorithm, as some aspect of a problem to which it is applied is increased in size. For instance, the summing of two vectors is said to scale linearly, since the operation count and time to completion are linearly related to the size of the vectors. Dense matrix multiplication, on the other hand, scales quadratically. A sequential algorithm is called scalable if it scales linearly with problem size.

In the case of a parallel algorithm, there are two types of size increase to consider. These are problem size and machine size - i.e. the number of processors applied to the problem.

If the problem size is held constant as the number of processors is increased, then the best one could reasonably hope for is a halving of run time for each doubling of CPU count. Insofar as an implementation accomplishes this it is said to be "perfectly" scalable. To express this more formally, define $T_{n,p}$ as the time it takes to execute a problem of size n on p processors. Then for a perfectly scalable code we would have

$$T_{n,\,p} = \frac{1}{p} T_{n,\,1}$$

In order to express the actual degree of scalability we define the ***fixed-size speedup*** to be:

$$S_f = \frac{T_{n,\,1}}{T_{n,\,p}}$$

For perfect scalability we would have $S_f = p$. To compare actual to perfect scalability we define the ***fixed-size efficiency*** as

$$E_f = \frac{S_f}{p}$$

The closer E is to 1 the more scalable the algorithm is. However, for a fixed problem size the efficiency must inevitably go down at some point, no matter how "embarrassingly parallel" the application, if for no other reason than because at some point the work will be spread so thinly that some processors have no work at all, while the rest have the smallest possible unit. Still, it is the most "honest" measurement of scalability, since any given problem is of some fixed size, and it is useful to know how much can be gained by running the problem in parallel.

Another measure of scalability considers the performance when problem size and processor count are increased proportionally. For instance, in a hydrocode this would mean holding the number of zones per processor constant while increasing the total problem size and processor count. In this case the ***scaled speedup*** is defined as:

$$S_s = \frac{T_{n,\,1} \cdot p}{T_{n \cdot p,\,p}}$$

Note that in this case we are comparing two separate problems; one of size n and one of size $n \cdot p$. ***scaled efficiency*** is then defined similarly as

$$E_s = \frac{S_s}{p} = \frac{T_{n,\,1}}{T_{n \cdot p,\,p}}$$

Scaled speedup is a more generous measure of scalability than fixed-size scalability, and so is considered suspect by some. But for the most part it is an accepted measure, particularly when considering very large problems that push the memory resources of even large parallel machines. For how is one to run such a problem on one processor, for comparison?

From the fixed-size point of view, the goal is to run the same problem in less time by adding processors. From the scaled point of view, the goal is to run larger problems in the same amount of time by adding processors. Which point of view is appropriate depends on which goal is sought.

The results of a scaled timing study for explicit time integration hydrocodes can be misleading. Typically when a problem is scaled, its overall dimensions do not change. Instead the mesh is refined, and each individual cell becomes smaller. There is a fundamental stability limit on maximum timestep that is dictated by the smallest cell size (the Courant or CFL Condition [41]), and so as the problem is scaled the timestep must be decreased. Even if an algorithm scales perfectly in its time to execute a single cycle, if its timestep must be decreased, then it will require more cycles to reach a given physical time. The global behavior is not scalable. If a timing study is run for a fixed number of cycles, and scalability results do not take into account the size of the timesteps, then the full picture is not there.

Nevertheless results for scaled timing studies are often given for a fixed number of cycles. The results in Chapter 5 of this document for scaled times, in fact, are given for a fixed number of cycles. The contact algorithm itself is not subject to the Courant Condition, but

is embedded in a code which is. Also there is another reason (Section A.11.12) why smaller contact faces require shorter timesteps. Thus any scaled timing results must be understood for what they are and also for what they are not.

Scalability issues may be separated into three classes: memory scalability, computational scalability, and communication scalability. Memory scalability requires that, as a problem is scaled, the data can be truly partitioned and stored in the distributed memory of the various processors. If, for instance, a problem size-dependent array must be stored locally by each processor, then the algorithm is not memory scalable, and at some point the per-processor memory will become saturated, even though the total distributed memory continues to grow as the problem is scaled.

Computational scalability requires that any computations applied to the entire problem space be factorizable into subtasks that can be executed in parallel on the various processors. For instance, if each processor is required to sort a problem size-dependent array, then the algorithm is not computationally scalable, since work per processor will grow as the problem is scaled.

Finally, communication scalability requires that, as the problem is scaled, the percentage of total time spent passing messages does not go up. If it does then the total time will go up, since it is the sum of computation and communication time (with some potential overlap). To be perfectly scalable, an algorithm must have no global communication since, even with optimal algorithms the number of messages sent by any given processor will grow as the log of the number of processors. Some global communication is usually inevitable as for instance, in calculation and communication of the next global timestep. How-

ever, any code that aspires to be even moderately scalable must employ strict discipline in minimizing global communication.

On the other hand local or "nearest neighbor" communication is entirely consistent with scalability since, as problem size increases the actual dimensionality of the problem doesn't change, and so the number of nearest neighbors remains bounded.

2.3 The Parallel Contact Problem

The problem of parallelizing a finite element or finite difference code without slide interfaces is fairly straightforward via a technique called ***Domain Decomposition***, and has been done many times. The first step is to decompose the mesh into N submeshes, such that each submesh contains about the same amount of computational work, and the number of nodes shared between submeshes is minimized (to ensure minimal communication requirements).The mesh decomposition process is typically performed as a preprocessing step using a tool such as METIS [30] or CHACO [31].

Each submesh is then assigned statically to a separate processor. Typically this involves assigning zones uniquely to processors. Since adjacent zones share nodes, zones on the partition boundaries will share nodes with other processors. It is determined during initialization which other processors share mesh nodes with this processor, and communication with these "nearest neighbors" is set up. The communication is needed because total force on these shared nodes will be the sum of the partial forces computed by each of the sharing processors.

Each cycle all the processors will swap partial forces on the shared nodes. Because both the mesh connectivity and the submesh-to-processor assignment are static, this communication pattern is also static and thus is relatively simple to set up.

When slide interfaces are introduced, the picture is much different. Consider Figure 2-1.

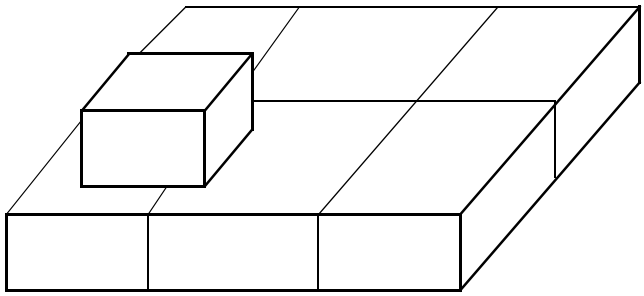


FIGURE 2-1

The upper block is free to slide around on the lower block under some external force. The grid in this figure represents the domain decomposition. That is, each grid element actually represents the entire submesh that is mapped to a single processor.

Just as in a single mesh, to calculate the force on a contact node it is only necessary to know its immediate environment. But unlike the single mesh, this environment is not dictated only by the mesh connectivity, but also by the current relative positions of the two contact sides. This relative position is not static, but must be determined as part of the problem which changes continually. In Figure 2-1 the processor owning the upper block need only communicate with the processors owning the two lower left submeshes. But in

Figure 2-2 at a later time it needs to communicate instead with the processors owning the two upper right submeshes.

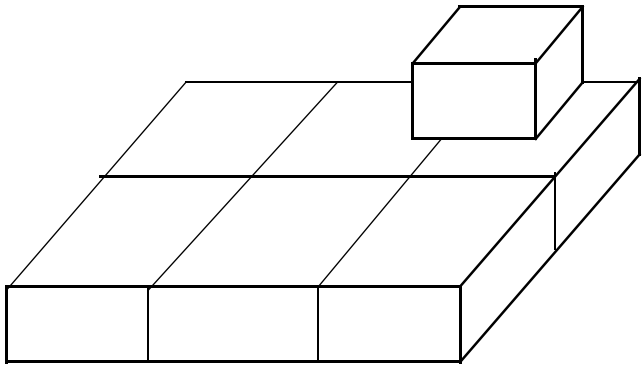


FIGURE 2-2

This shifting communication pattern makes contact a much more difficult problem to parallelize in a scalable fashion than the single mesh problem.

2.4 Recent Parallel Contact Algorithms

2.4.1 The Volume Checking Algorithm

In a pair of papers [32] [33], the authors describe an algorithm for parallel contact detections and enforcement. The problems they address consist entirely of structural shell elements, and any element can potentially come into contact with any other element.

The overall problem is decomposed into N subsets that are load-balanced with respect to the non-contact aspects of the computation. It is not possible to predict ahead of time where contact will occur, so this aspect cannot be included in the static load balance. Each processor "owns" an initially compact subset of the shell elements.

The following will focus on the work done by an arbitrary processor, called processor A. Each processor in the machine does the same, on its own data set. During a given cycle,

processor A determines a "bounding box", the smallest rectilinear region containing all its local elements. In a global communication pass, each processor then shares with all other processors the location of its bounding box, and receives the locations of theirs.

Processor A is then in a position to determine which other processors have bounding boxes that intersect its own. Where there is no intersection, no contact is possible. On the other hand, if there is non-empty intersection then contact is possible and must be further investigated. In order to do this processor A needs the details (positions, velocities) of all the elements contained in the intersecting bounding boxes. This is obtained via a communication pass whereby each of the intersecting regions is sent from its home processor and concatenated into a "contact domain" on processor A. Meanwhile in a symmetric operation, processor A sends all its data to every other processor that intersected it, and those processors each build their own contact domains.

Each processor now has all the data it needs to find any contact that occurs involving its own local elements, and to resolve the results of that contact.

The authors found, in running on 128 processors, that the chief obstacle to parallel efficiency in their implementation was due to load-balance. Only some subset of the total problem would be in contact at any point in time. Since the contact domains were tied to the original decomposition, those processors whose local element set were not in contact could do nothing but wait for the contact calculations to complete. Another problem was that, as time went on, the initially compact bounding boxes would tend to spread out, and more processors would have non-empty intersections, leading to increased communication.

2.4.2 DYNA3D

Dyna3D is a well-known explicit finite-element transient dynamics code [12]. In its serial version its treatment of contacts has been well-documented [34], [35]. The parallel version of the code, ParaDyn [36], has adopted dual strategies to deal with contact surfaces. In both these strategies a second decomposition, separate from the static bulk ("primary") decomposition, is employed to treat the contact.

In the first strategy, which they refer to as ***local contact***, regions of contact are somewhat predictable, as for instance when a tool bit meets a surface, and it is possible to specify pairs of surfaces, historically called master and slave, which will contact each other, even though specific details of which nodes and faces contact may not be known a priori, and in fact will change over time. For a complex engineering structure there may be many such pairs of surfaces.

The Dyna3D approach is to assign each such contact pair to a separate processor to be worked on in parallel. Since this assignment is static the communication, whereby nodal variables such as position and velocity are sent from the primary decomposition home to their assigned contact domain, can be set up during initialization, and is essentially no more complex in this respect than a finite element code without contact surfaces.

If the problem has many contact surfaces, this approach is certainly an assistance in speeding up overall execution time, but it clearly will not scale beyond the number of independent contact surfaces. Even within this range load-balance is beyond the control of the algorithm, and depends wholly on the problem definition.

The second Dyna3D strategy, referred to as ***arbitrary contact***, assumes a single surface (which need not be connected). In other words, the restriction that some part of the surface, designated the master, can only come into contact with some other disjoint part of the surface, designated the slave, is removed. This generalization is required to handle problems such as beam buckling, where the surface bends until it contacts itself, and many-body simulations, where it cannot be predetermined which bodies will collide with each other. In this case, the union of the surfaces of all the bodies is the single contact surface.

The Dyna3D parallel arbitrary contact strategy involves a spatial sort of the contact nodes in one, two or three dimensions depending on the problem geometry and the number of processors. The sort employed is known as a ***bucket sort*** and is documented (for the serial version) in [35].

The goal is to restrict the number of faces that must be checked for penetration by a given contact node. To accomplish this, the algorithm first searches for the nearest contact node (excluding itself and any other nodes immediately adjacent to it in the mesh connectivity). Then any faces connected to the nearest node are candidates for penetration.

The bucket sort consists of dividing the problem space up with a 1D, 2D, or 3D grid, then assigning each contact node to the grid zone, or "bucket" into which its coordinates place it. The size of the buckets are chosen so that contact is only possible between nodes and faces that are within the same or immediately adjacent buckets.

The contact decomposition then consists of assigning some compact subset of buckets to each processor, and is determined dynamically based on the current population of the

buckets, in order to preserve load-balance. During the contact phase a global communication pass is used to inform all processors of the bucket-processor assignments, and they can then determine where to send their local contact nodes for the detailed contact detection and enforcement calculations.

2.4.3 PRONTO3D

Pronto3D [37] is another explicit finite element transient dynamics code, similar in its application scope to Dyna3D. In addition it includes a ***Smoothed Particle Hydrodynamics*** (SPH) module that supports solid-fluid interface and fragmentation problems. Like Dyna3D it uses a separate problem decomposition for contact in order to load balance the contact calculations. The SPH calculations also employ a separate decomposition.

Pronto3D uses a method known as ***Recursive Coordinate Bisection*** (RCB) in order to decompose the problem into contact domains. The overall problem space is first divided by a plane perpendicular to one of the coordinate axes. The position of the plane is chosen so that half of the contact nodes and faces fall on each side of the plane. To accomplish this, a first guess at the plane's position is made, then each processor adds up the number of its local (i.e. primary decomposition) contact nodes and faces it has on either side of the split. All processes then communicate, determine how far off the first guess was, move the plane's position accordingly, and iterate. Typically a few iterations are sufficient.

Then the processors are divided into two equal groups, and one group assigned to each side of the split. If a given processor owns any nodes or faces that fall on the other side of the split, it sends them to a "partner" processor assigned to the other side.

After this step is done, the same operation is performed recursively for each of the half-spaces created by the first bisection, only now each side splits its set of half the processors into two sets, each with one quarter of the processors. Again, if a processor finds that it has nodes or faces that belong to the other half of the bisection (either from the primary decomposition, or because they got passed during the first bisection) it passes them now to a partner processor in the other half of the active bisection.

When the last recursive level of bisection is complete, each processor will have its own spatial contact domain, and will have all the nodal and face information that it needs in its local memory. However, because the nodes and faces are moving, it is necessary to also have data from surrounding regions, which are assigned to neighboring processors. To accomplish this, each processor needs to find out who its neighboring processors are. This depends on the result of the RCB: i.e. where specifically the cuts were made. Another round of global communication distributes this information so that all processors know the regions assigned to all other processors. From this they can calculate who their neighbors are, and exchange information about their respective contact nodes and faces.

At this point each contact domain has enough information to enforce contact for its local nodes and faces. It sorts its list of local nodes in each dimension, and uses these sorted lists to reduce the number of collision possibilities. For the contact enforcement, once it has determined the collisions, it uses a version of the penalty method (Section 1.3).

Finally, the results of the contact enforcement on a given node are sent back to the processor that "owns" the node in the primary decomposition.

At the beginning of the contact phase at every timestep (except the first) the contact nodal and face data are sent to the processor that they were assigned to, via the RCB, during the previous cycle. This provides a good "first guess" for the current RCB step, so that after this initial send from the primary domain to the old contact domain, only a small subset of the nodes will typically need to be moved again. Or from another perspective, the coordinates of the RCB cut this cycle should be close to the cut coordinates of the previous cycle.

The scalability of this algorithm has been demonstrated for some problems on up to 3,000 processors [38], [39].

2.4.4 ALE3D

The ALE3D code developed at LLNL is a multiphysics ALE finite element hydrocode that includes thermal diffusion, chemistry, and incompressible flow models as well as explicit and implicit continuum mechanics [26] [40]. In the late 1990s the serial version was completely rewritten as a message-passing parallel code. The contact algorithm described in detail in the following chapters has been implemented in ALE3D, and is currently in production use.

As in Dyna3D and Pronto3D, ALE3D makes use of a separate decomposition for contact calculations. This decomposition is partly static, partly dynamic. The master sides of the contact surfaces are permanently (statically) assigned to various processors in a load-balanced fashion (Section A.6.8). The slave sides are assigned dynamically: each cycle a given slave node is assigned to the contact domain of the corresponding master node clos-

est to it. Because of surfaces sliding this domain assignment will, in general, change over the course of time.

The slide interfaces treated in ALE3D are essentially the same as the ***local contact*** interfaces in Dyna3D. The serial ALE3D code also implemented ***arbitrary contact***, but there has been very little demand for arbitrary contact in the ALE3D user base, and it has not yet been implemented in the parallel code.

The assumption of local contact is that a node's nearest neighbor on the other side of a contact surface will not change very much, in terms of the connectivity of the surface, in a single cycle. This constraint is more formally defined in Section A.11.12. The difference between ALE3D and Dyna3D local contact is that, in ALE3D, there is no restriction of a given contact surface to a single processor. It is a fully general parallel implementation. While it is not as general as arbitrary contact, as implemented in Dyna3D and in Pronto3D, this loss (not an issue with ALE3D users) is balanced by the ability to create a truly scalable algorithm, as described in the following chapters. There is absolutely no global communication in the ALE3D contact algorithm. Thus it will never be a limiting factor in the overall scalability of ALE3D.

3 THE METHOD

3.1 Introduction

In this chapter we discuss a new method for implementing contact on a parallel distributed-memory machine using only local communication. This allows the algorithm to be scaled to an arbitrarily-large system. We believe this is the first implementation of contact surfaces that can make this claim.

Contact problems may be separated into two phases: ***contact detection*** and ***contact enforcement***. Contact detection determines when and where contact occurs. If the contact is persistent as in sliding surfaces, it determines which nodes are opposite which faces each cycle. In its most general setting, contact detection is an $O(N^2)$ problem. Each node on the surface must be compared against each face to see if contact has occurred during the current timestep. Once contact is detected, then contact enforcement is applied to model the forces which act across the contact interface.

The order of complexity of the detection problem can be reduced by several techniques. For instance, the nodes and faces may be sorted into a number of "bins" based on coordinates. Then contact need only be checked for in the same and neighboring bins. The sort

itself is O(NlogN). If N = m x p, where m is the number of nodes or faces in a bin, and p is the number of bins, then the search is O(p x m^2), which is only 1/p of N^2. Dyna3d uses this approach in its single-sided surfaces [35]. The approach taken by Ale3d, which is the method upon which this work is based, uses the mesh topology, or connectivity, of the contact surfaces, and makes the assumption that nodes and faces will only move a short distance in a single timestep with respect to the opposing side. Thus it can restrict its search for nearest nodes to the mesh neighborhood of nearest nodes the previous cycle. If contact occurs, it must be in the neighborhood of nearest nodes.

When the contact problem is moved to a parallel, distributed memory environment the detection problem becomes more difficult because, in general, a given node and the set of its potential contact faces are not on the same processor. Furthermore, as the problem evolves the processor(s) containing the potential contact faces may change. This is not because the nodes and faces themselves move from one processor to another, but because the sides of the surface (or portions of the single-sided surface) are in motion relative to each other.

On a given cycle we can define the ***communication topology*** to be the arrangement of which processors communicate with each other to send the node and face information from wherever it is permanently located to where the detailed contact detection and enforcement will be performed. Since the detection search is local, as described above for Ale3d, and since contact enforcement is inherently local (for explicit codes), the number of processors that need to communicate with a given processor is bounded. So the communication topology is of "local" form and the communication can be done scalably.

However, there is a second problem besides sending the contact information where it is needed, and that is figuring out where it is needed in the first place, or in other words, *evolving* the communication topology. It is not obvious how this can be done scalably using only local communication. If processor A has no communication with processor B at one time, how do they simultaneously decide at a later time to start communicating, without any shared global context?

The primary original contribution of this thesis is the demonstration that scalable evolution of the communication topology for parallel contact using only local communication is possible, by a description of one such implementation.

3.2 Authorship

The starting point for this work was the existing Ale3d serial contact algorithm, and a set of decisions about the parallel implementation:

1. The parallel contact calculations would use a decomposition independent of the static bulk decomposition. Each cycle all required data would be transferred from bulk domains to appropriate contact domains, and the results returned to bulk domains.
2. The same master nodes would be assigned to the same contact domains each cycle: i.e. the master side decomposition would be static.
3. Each slave node would be assigned to the contact domain of its nearest master node. Since the nearest master node will change as the surfaces slide, the slave decomposition would be dynamic and not known in advance.

Given this initial set of assumptions, the goal was to create a distributed, scalable algorithm that could determine what data needed to be sent where and orchestrate its transfer. This algorithm would be implemented in a new parallel version of Ale3d.

An initial version of parallel Ale3d slides was developed by the author in close collaboration with another engineer, Jim Maltby. Many hours were spent discussing the ideas which were eventually implemented. This first version, unfortunately, was not fully scalable. Its performance on large problems was acceptable on machine configurations up to about 256 processors, but between there and 512 processors there was a drastic rolloff in parallel efficiency.

This led to a serious reexamination of the original design and the search for a more scalable approach. This work was done independently by the author, and the resultant design is documented in this thesis.

It must be emphasized that the original work described here is concerned with the distributed nature of the parallel contact problem, and with solving the problem in a scalable fashion. The actual contact enforcement, once all necessary data has been transferred to the appropriate places, is simply an adaptation of the corresponding serial Ale3d contact algorithm.

3.3 Limitations

The method developed in this thesis is specific to 2-sided contact. A 2-sided contact surface is one in which contact is known ahead of time to only occur between specific pairs of surfaces. Also, movement between the two sides is assumed to be continuous. Thus, if a

node "a" on one side is the closest node of that side to a node "b" on the other side during a given cycle, then on the next cycle the closest node to "b" will be some node in the mesh neighborhood of "a" (see Section A.11.12). In the search for the closest point on the other side, we can always start with the closest point the previous cycle and just do a local search. Two examples of 2-sided contact problems are given in Section 1.4.

On the other hand, in what are sometimes called single-sided surfaces, any node may come in contact with any face of the same surface. A naive implementation would require an N^2 search each cycle to detect which nodes and faces are in contact. In practice, however, the nodes may be sorted in one or more dimensions as discussed in Section 3.1, in order to reduce the number of "buckets" that need to be examined for nearest neighbors. The sort may be not be necessary every cycle, if limits on the amount a node may move in a cycle can be established.

A prominent example of single-sided surfaces is the study of buckling, such as occurs in car crash simulations[35]. Another is the study of multiple colliding bodies, such as billiard balls. This problem can be posed using 2-sided contact surfaces. For instance, each ball's surface is the master side of its contact surface, and the surfaces of all other balls combined represent the slave side of that surface. Thus the surface of a given ball represents at the same time one master surface and many slave surfaces. But it is much more direct to simply treat all surface area as one single-sided surface.

Single-sided contact is more general than 2-sided contact, since any 2-sided contact problem can be reformulated as single-sided contact, but not vice-versa. However, for problems where contact between specific subsets of the surfaces involved can be predicted,

and in particular for surfaces where sliding rather than impact dominates, 2-sided contact is much cheaper. The serial Ale3d code had a single-sided contact capability, but this has not yet been ported to parallel Ale3d due to lack of demand by the Ale3d user community.

3.4 Definitions

The ***problem mesh*** is the discretization of one or more physical bodies into finite element zones (also called cells). An example problem mesh is shown in Figure 3-1. In this case there are three disjoint bodies, and three corresponding disjoint submeshes.

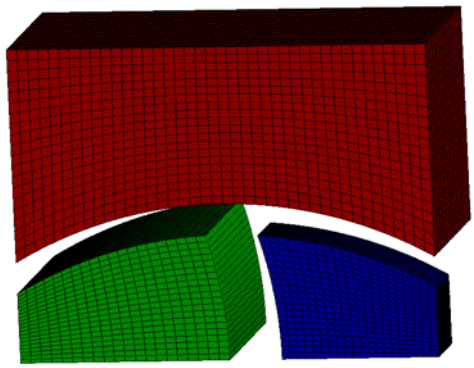


FIGURE 3-1

In order to simulate the dynamic behavior of the bodies on a parallel machine, the mesh is partitioned into a number of ***bulk domains***. Typically each bulk domain will be assigned to its own processor with separate memory. In order to share data, the processors send

messages to each other. Figure 3-2 shows a typical bulk decomposition into 8 domains, where each color represents a different bulk domain.

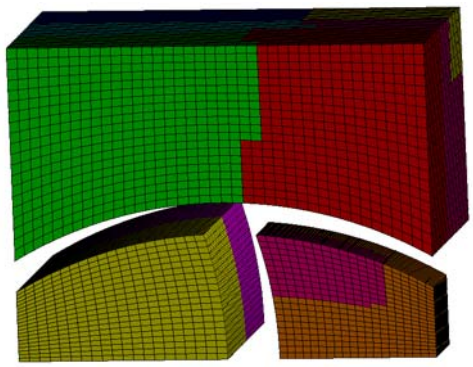


FIGURE 3-2

Some subset of mesh faces forms the boundary surfaces of each submesh. These are the ***boundary faces*** of the problem mesh.

A (two-sided) contact surface consists of a pair of boundary face subsets, called the ***master side*** and ***slave side*** of the contact surface. These subsets are specified by the user, and presuppose some knowledge of which parts of the bodies will come in contact. In Figure 3-3 the preceding example has been stripped down to two contact surfaces.

FIGURE 3-3

In this example the red and blue surfaces form the master and slave sides of one contact surface, while the green and yellow surfaces form the master and slave sides of a different surface. Each side of each contact surface is itself a 2-D mesh. The relation between the various faces and nodes of each mesh is fixed, and is called the ***connectivity*** of the mesh. Each face on a contact side always includes the same nodes, and each node is a vertex for the same set of faces.

The ***neighborhood*** of a node is defined for our purposes to include all the nodes within two faces of the specified node. Figure 3-4 shows the neighborhood of node "a" as all the circled nodes.

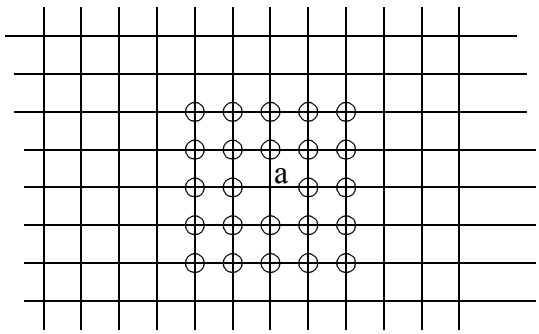


FIGURE 3-4

Contact enforcement in explicit codes is essentially a local process: Each "patch" on one side of a contact surface is only affected by a patch directly opposite it on the other side. But the opposing patch may be continually changing as the two sides move relative to each other. To make this more precise, we start with the definition of ***order node***. The order node of a given node is the closest node on the other side of the surface. Each contact node has an order node each cycle, which is in general constantly changing as the problem evolves. There is a limit to the amount of change allowed in a given cycle, however. The order node of a particular node next cycle must share a face with its order node

this cycle. In other words there must exist a face on the opposing side for which the previous and current order node are both vertices.. This allows us to limit the search for the new order node each cycle to a local search.

The ***locale*** of a contact node consists of the node itself, its neighborhood, its order node, and the neighborhood of its order node. The locale is a more precise definition of the "two opposing patches" mentioned earlier. The locale of a node is the minimum amount of state required in order to perform the contact enforcement and subsequent contact detection update on that node. In Figure 3-5 the locale of node "a" consists of all nodes marked by open circles.

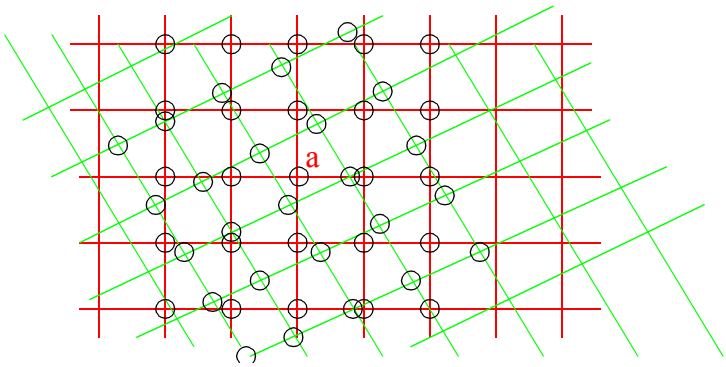


FIGURE 3-5

In order to parallelize the work of contact enforcement, the contact surface is partitioned into a number of ***contact domains***, and each contact domain is assigned to a processor. Note that a given processor will hold both one or more bulk domains, and a contact domain. The nodal data "lives" in the bulk domains, but is sent to the contact domains each cycle for contact enforcement. The updated nodal data is then returned to the bulk domains. The specific pattern of which bulk domains communicate with which contact domains on a given cycle is an example of a communication topology, which is further discussed below.

A contact node is ***assigned*** to a particular contact domain if its state is updated and returned by that contact domain, in the process of contact enforcement. The master nodes are statically assigned to contact domains: a given master node is assigned to the same contact domain each cycle. A slave node, on the other hand, is assigned to the contact domain of its order node. As the order node changes during the course of relative movement, so may the contact domain to which a slave is assigned.

By ***communication topology*** we mean the overall organization of which processors talk to each other. The cross-section of the topology for a given processor is all the processors it talks to.

The two characteristics of the topology that we are interested in are:

1. static vs. dynamic, and
2. local vs. global

If the topology is static, each processor talks to the same set of other processors (its "neighbors") each cycle. If it is dynamic its set of neighbors may change each cycle.

Suppose that we define a scaled series of related problems, arrived at for instance by increasingly refining the mesh for some set of interacting bodies, and suppose we run these on a proportionally increasing number of processors, such that in each instance about the same number of zones or the same number of contact faces are assigned to a given processor. Then the communication topology is called local (or nearest-neighbor) if there is a fixed upper bound to the number of other processors that a given processor talks to, regardless of problem size. For instance, if bulk domains supply nodes to a contact domain, and if the number of nodes per contact domain is fixed as the problem grows,

then the absolute upper bound on the number of bulk domains that need to talk to that contact domain is equal to the number of nodes sent to the contact domain. In practice data layout tends to have a geometric locality, and for 2-D contact surfaces the number of neighbors (per side) will be in the neighborhood of 9.

On the other hand, a global topology is one in which, as the overall number of processors grows, so does the number that a given processor talks to. The archetype of this topology is all-to-all communication.

3.5 Contact Domain Requirements

Some subset of the contact nodes is assigned to each contact domain, or ***CD***, each cycle. It is the job of the CD to calculate the force on each of its assigned nodes due to the other contact side, to advance the position of each of its assigned nodes, and to update its order node based on the new positions.

Each master node is assigned to the same CD each cycle. Slave nodes, on the other hand, are assigned to whichever CD their master node is assigned to, and this will change from cycle to cycle.

In the rest of this chapter, simplified 1-D diagrams will be used to represent the contact surface. Also, node neighborhoods will only include nodes within one face (or segment, in 1-D) of a given node. This makes the description of various scenarios less cluttered, while the extension to 2-D surfaces should be obvious .

In Figure 3-6 a contact surface is shown at two different times. At time t_1, the arrow from s_4 to m_3 means that s_4's order node is m_3 (alternate wording: s_4 ***orders on*** m_3.). m_3 is

assigned to CD 5, so s_4 is also assigned to CD 5. At a later time t_2 the slave side has moved and now s_4 orders on m_4, so s_4 is assigned to CD 6.

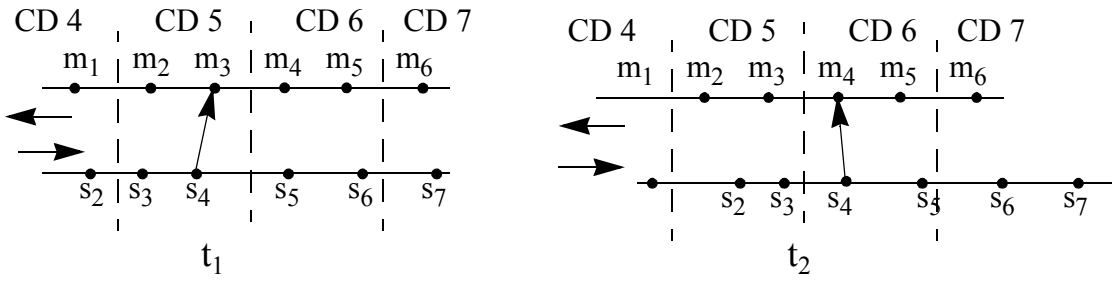


FIGURE 3-6

If a node is assigned to a CD, the CD also requires the node's locale in order to update that node. Figure 3-7 shows an example of the locale requirements. Given that m_9 is assigned to CD 5, all of the circled nodes must also be sent to CD 5.

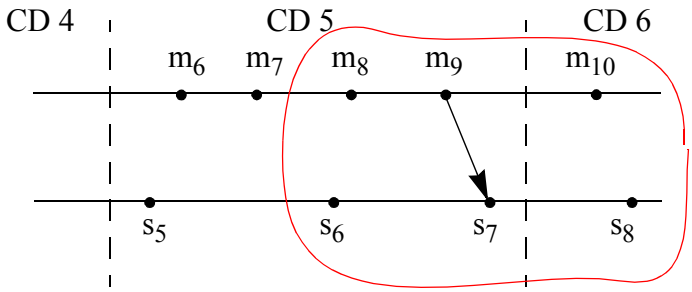


FIGURE 3-7

Remember that this is a "simplified" diagram. In a more realistic diagram each side would be a 2-dimensional mesh, and the neighborhoods of m_9 and s_7 would extend out two faces, not one .

Because of the locale requirements of m_9, CD 5 needs nodes m_{10} and s_8, even though they are assigned to a different CD. m_{10} and s_8 are called ***ghost nodes*** of CD 5.

3.5.1 Static Nature of Master Side CD Requirements

Focusing on the master side requirements of a given CD, there are four sets:

1. Assigned master nodes
2. Neighbors of assigned master nodes
3. Order nodes of assigned slave nodes
4. Neighbors of order nodes of assigned slave nodes

By design of the contact decomposition, the same master nodes are assigned to the CD each cycle, so the first set is static. Most but not all of the second set will typically also be in the first set (Section A.5.2). In any case, since the connectivity of the contact side is fixed and the first set is static, the second set is also the same each cycle.

The third set is not strictly static. However, it is always a subset (not always the same subset) of the first set, which is static. This is because a slave is only assigned to the CD if its order node is in the first set. Therefore the fourth set is always a subset of the second set.

The point is that each CD has exactly the same set of assigned and ghost nodes each cycle. As with all nodes, these nodes also have a static bulk domain home. Therefore the communication topology required to deliver the master nodes from the bulk domains to the contact domain and back is also static, and can be set up once for all during initialization. Because of its relative simplicity, we will assume the master side communication and focus on the slave side from here on.

3.6 Communication Topology Evolution

In order to effect the communication between bulk domains (BDs) and contact domains:

1. Each BD must determine which CDs to send to, and what to send them,
2. Each CD must determine which BDs it will receive from.

Both these steps are necessary because each communication requires the sender and receiver to explicitly cooperate. We cannot simply send a CD messages from various places and have it poll for any messages coming in. It would never know when it had waited long enough (unless it waited till it got messages from all BDs, which would imply global communication).

Suppose that in cycle *n* a certain bulk domain and contact domain had no communication between them, but due to the relative movement of the contact sides, during cycle *n+1* the BD needs to send the CD some of its nodes. Since they are not yet communicating, they need to simultaneously but independently decide to start communicating at cycle *n+1*.

3.7 Bulk Domain Work

Each bulk domain is responsible for determining where to send each of its local nodes for contact processing. As discussed above, we will focus on the slave nodes. Each of the BD's local slave nodes will be sent to one CD as an assigned node, and to zero or more CDs as a ghost node.

There are three overlapping reasons why a BD might send one of its local nodes to a particular CD:

1. Its current order node is statically assigned to that CD (in which case the slave node is also assigned there)
2. It is the order node of a master node that is assigned to that CD
3. It is in the neighborhood of a slave node that falls into the first or second class.

The second and third classes are a consequence of the locale requirements for assigned nodes. If a node is in the first class it is called a ***rule 1 generator*** for that CD, because it generates a list of other nodes (its neighbors) that must also be sent to the CD. A node is a rule 1 generator for exactly one CD, its assigned CD (which changes dynamically for slave nodes.)

A node in the second class is called a ***rule 2 generator*** for the CD. It can simultaneously be a rule 2 generator for multiple CDs (or none). Figure 3-8 gives an example:

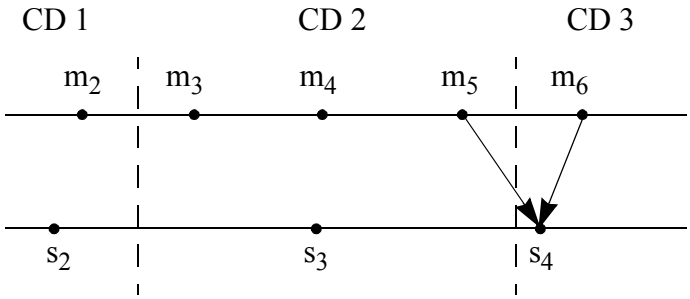


FIGURE 3-8

Even though s_3 and m_5 are both assigned to CD 2, node s_4, which is assigned to CD 3, is closer to m_5 than is s_3, so m_5's order node is s_4. m_6's order node is also s_4, so s_4 is a rule 2 generator for both CD 2 and CD 3 (and a rule 1 generator for CD 3).

To support all this, each node carries with it two additional items of state:

1. Its rule 1 CD
2. Its rule 2 CD list

How these become known to the node goes back to the end of the previous cycle. At that time the node was assigned to some CD (its rule 1 CD that cycle). After its position, and the position of all nodes in its locale, were updated, a new order node was selected from among the nodes in its old order node's neighborhood. The identity of the new order node

was then returned along with the rest of the node's updated state to its home bulk domain. Since the master side decomposition is static, the bulk domain just looks up which CD the new order node is statically assigned to, and that is the slave node's new rule 1 CD.

The rule 2 CD list is built a little less directly. The node was sent the previous cycle to some number of CDs as either an assigned or a ghost node. At the end of that cycle the assigned master nodes on each of those CDs determined new order nodes. If a particular slave node was chosen as order node of any assigned master node(s) on that CD, the slave node is flagged. In the return CD→BD communication at the end of last cycle, the CD sends to the BD a list of any of the BD's slave nodes that were flagged.

If a slave node is sent to multiple CDs, any or all of those CDs may returned flagged status on that node. Any CD that does so is added to the node's rule 2 CD list for the next (i.e. current) cycle. In the previous Figure 3-8, both CD 2 and CD 3 will return a flag on node s_4 to the BD(s) where s_4 lives, and s4's rule 2 CD list will be {CD 2, CD 3}.

3.8 BD <-->BD Communication

Consider Figure 3-9:

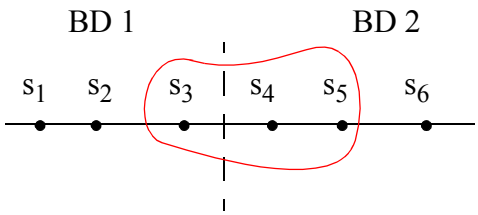


FIGURE 3-9

The circle represents the neighborhood of node s_4. Suppose that s_4 has rule 1 CD set to CD 3. This means that s_3, s_4, and s_5 must all be sent to CD 3. Since s_4 is on BD 2, BD 2 can

look at its rule 1 CD and its neighbors, and determine to send s_4 and s_5 to CD 3. But how does BD 1 know that it needs to send s_3 also to CD 3?

The solution we use is rather standard. We create a ghost layer of nodes around each bulk domain's local set. To avoid confusion with the contact domain ghost nodes, we will instead refer to these nodes as ***proxy nodes***. For a given BD, during an initialization phase the set of local nodes is used to generate the larger set of locals plus proxies, by taking the union of the neighbors (i.e. nodes within two faces) of all the local nodes. Then the locals are removed, leaving only the set of proxy nodes for the bulk domain, and these are sent to all the other BDs for matching against their own local nodes. When a match is found this is reported back to the originating BD.

The result is that a static communication topology is established between each BD and the other BDs where its proxy nodes originate from, or where its local nodes are proxy. Not only the topology, but also the list of nodes to be sent is static. At the beginning of each cycle each bulk domain sends to the other BDs in its "neighborhood" (i.e. the BDs that have proxy copies of any of its local nodes) the latest rule1 CD and rule 2 CD list for each of the shared nodes, and receives the same.

The BD↔BD communication topology is a local, nearest-neighbor topology because it is bounded by the number of native nodes times the number of neighbors per node. In practice each bulk domain will have on the order of eight neighbors (a 3 x 3 grid, where the central element represents the local BD, which doesn't send to itself).

3.9 Bulk Domain Preparation for Sending

After the BD↔BD communication pass, the bulk domain is ready to package the data to send to each of its neighbor contact domains. To do this, it loops through all its local and proxy nodes. For each node it looks at the node's rule 1 CD and rule 2 CD list. If this is the first time a CD is encountered it creates a new list for nodes going to that CD, and adds the CD to its list of CDs to send to.

If the node is local it is added to the list of nodes being sent to the rule 1 CD (if not already on the list) and flags the node as an assigned node. Then, whether the node is local or proxy, it adds all the local neighbors of the node to the lists of all CDs in the rule 1 and rule 2 CD lists (if not already there).

This satisfies requirement 1 of Section 3.6.

3.10 Contact Domain Preparation for Receiving

As mentioned earlier (Section 3.6), each contact domain must know ahead of time which bulk domains will send it messages, in order to post the receives of those messages. This determination is actually done at the end of the previous cycle.

After the order nodes have been updated at the end of the previous cycle, each contact domain determines which slave nodes have order nodes that are among its assigned masters (the CD's new rule 1 generators) and which slave nodes *are* the order nodes of its assigned masters (the CD's rule 2 generators). This is a start, but the CD will also receive all the neighbors of its generators as well, and these neighbors may not all be from the same BDs as the generators themselves (see Figure 3-9). To solve this problem, when a

node is sent to a CD some extra data is sent with it called the ***neighbor BD list.*** For a given node, its neighbor BD list is a list of all bulk domains on which the node is local or proxy. If a BD appears in a given node's BD list, then that bulk domain contains at least one node in the neighborhood of the given node, and thus that BD will communicate with the contact domain.

So the CD finds all slaves that order on its assigned masters, and all slaves that are ordered on by its assigned masters, and takes the union of all their neighbor BD lists in order to determine which BDs it will talk to next cycle. It does all this at the end of the previous cycle, and saves the resulting list in its ***next time BDs list*** for use the coming cycle**.**

3.11 BD --> CD Communication

In this step each CD simply issues receives from each BD in its next time BDs list. Each BD allocates a buffer for each CD on its CD send list (Section 3.9). It fills each buffer with the state information for each node in the list of nodes to send to that CD. This state includes the node's neighhbor BD list and either its assigned CD or list of ghost CDs (these are discussed in the next section), as well as its position, velocity, order node, etc. When the buffer is complete it sends it to the destination CD.

The CD waits until it has received all its expected messages, and then assembles the data from all buffers into its piece of the contact surface. Section A.8 and Section A.10 contain considerably more detail about this part of the operation.

3.12 CD<-->CD Communication

A slight difficulty was glossed over in the last section. The statement "the CD finds all slaves that order on its assigned masters" is the culprit. Consider Figure 3-10:

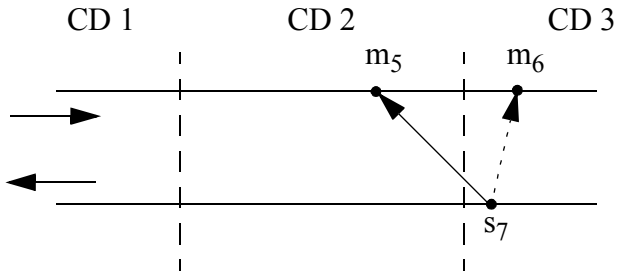


FIGURE 3-10

Suppose at the start of the previous cycle s_7's order node was m_6, but as a result of relative surface movement, at the end of the previous cycle s_7's new order node was m_5. Because s_7 intitially ordered on m6 it was assigned to CD 3, and it is CD 3 that calculated its new order node. However, it is CD 2 that needs to know s_7's new order node, so that it will know that it has to accumulate s_7's neighbor BD list into its next time BDs list.

To accomodate this we define another communication topology, between each contact domain and all other contact domains where any of its ghosts are assigned, or where any of its assigned nodes are ghosts. This is again a local topology, bounded by the total number of nodes in a contact domain, with the typical number of neighbors being on the order of eight.

Actually, this communication topology is used several times during the course of the contact calculation each cycle. For example in Figure 3-10, when s_7 determined that its new

order node was m_5, it needed the freshly updated coordinates of m_5, a ghost node to CD 3, in order to make that determination. This is discussed in greater detail in Section A.12.

To construct this topology, for each local node the bulk domain builds a list of all the contact domains it will send the node to as a ghost node. When it sends the node to its assigned CD, it sends this list with it. When it sends the node to any other CD as a ghost, it sends along with it the information of which CD it is assigned to. Each CD constructs from this information a set of CDs where its assigned nodes are ghosts, and another set of CDs where its ghost nodes are assigned. Unlike the BD↔BD topology, the CD↔CD communication topology is dynamic.

3.13 An Example

A simple example is provided to demonstrate how the communication pattern evolves. In a more realistic example the surfaces would be 2-D, and the node neighborhoods would include all nodes within two faces. As a further simplification, all the nodes on one side of the surface will exactly match up with a node on the other side, so that the rule 2 CD list will only contain the rule 1 CD. Nevertheless this example shows much of the mechanism by which BDs and CDs start and stop communicating with each other in the course of contact evolution.

Figure 3-11 shows the static layout of the contact surface in the six bulk domains of the example:

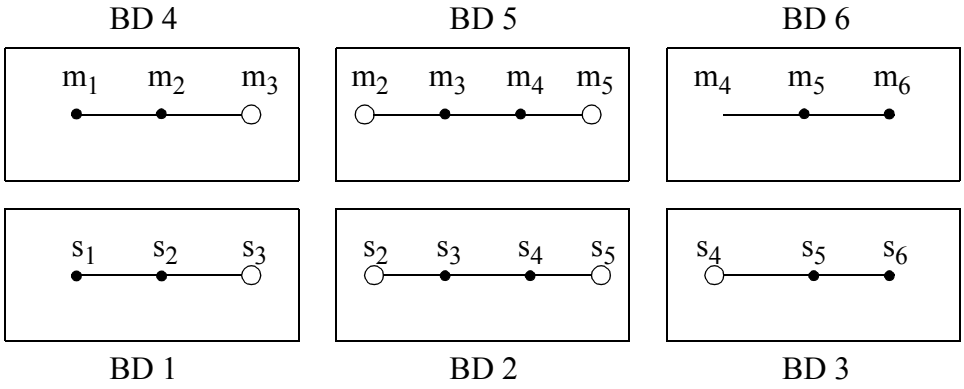


FIGURE 3-11

The nodes represented by open circles are proxy nodes. From this we can see that the static BD↔BD topology is:

BD 1 talks to {BD 2}
BD 2 talks to {BD 1, BD 3}
BD 3 talks to {BD 2}
BD 4 talks to {BD 5}
BD 5 talks to {BD 4, BD 6}
BD 6 talks to {BD 5}

The static master side contact decomposition is shown in Figure 3-12. The filled circles represent assigned nodes to that domain, the open circles represent ghost nodes.

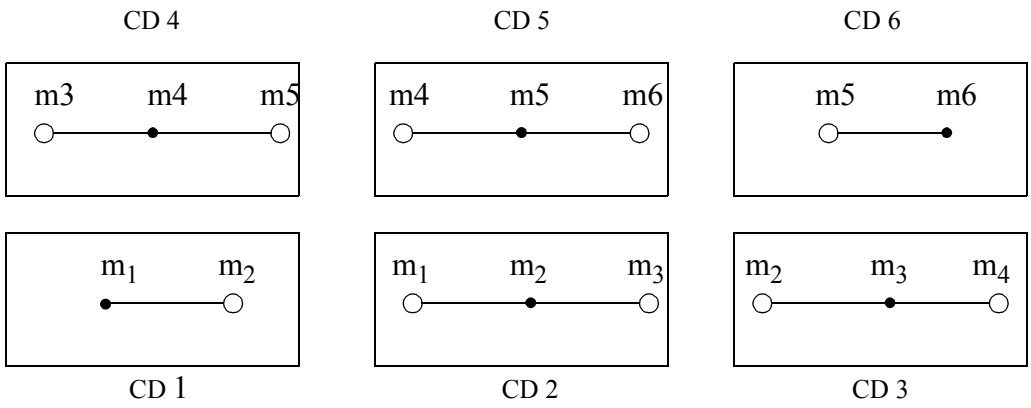


FIGURE 3-12

Suppose that at the start of cycle n-1 the contact sides are oriented as shown in Figure 3-13. Each node m_i both orders on and is ordered on by corresponding node s_i.

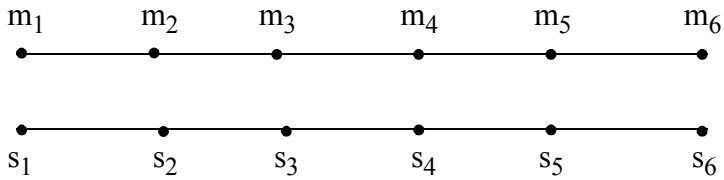


FIGURE 3-13

Then the contact domain configuration will be as shown in Figure 3-14:

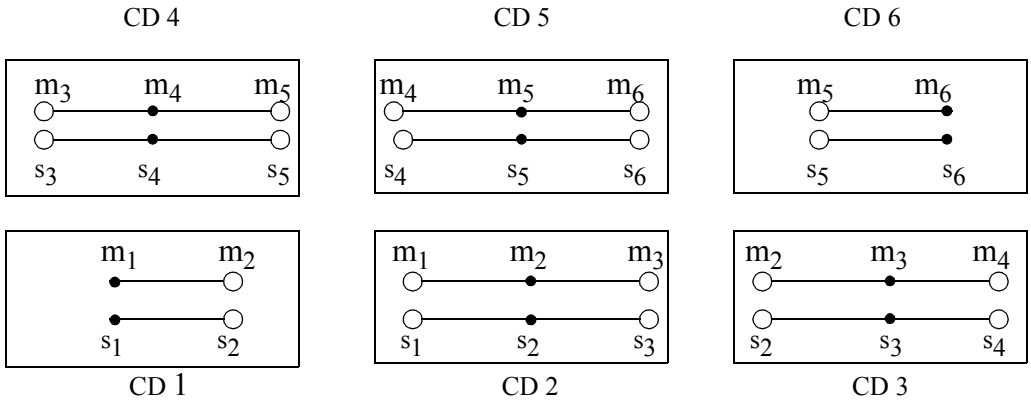


FIGURE 3-14

As a result of the relative movement and the update of the order node information, at the end of cycle n-1 the contact surface has changed to:

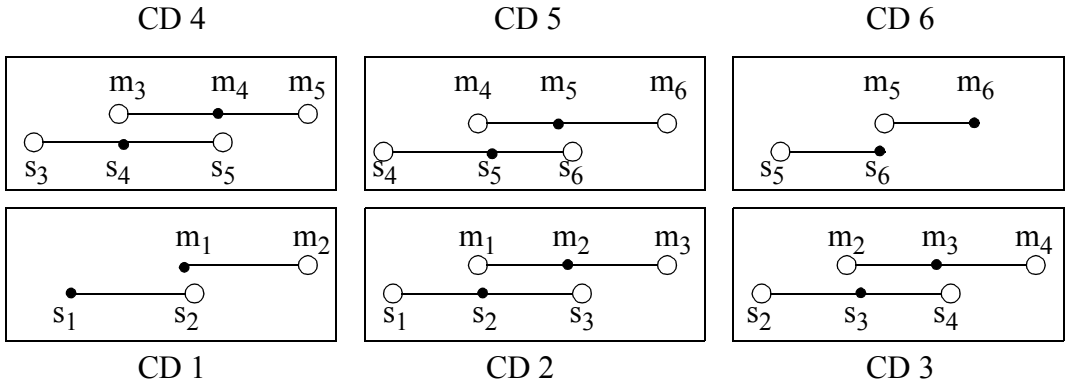


FIGURE 3-15

Or from a global perspective:

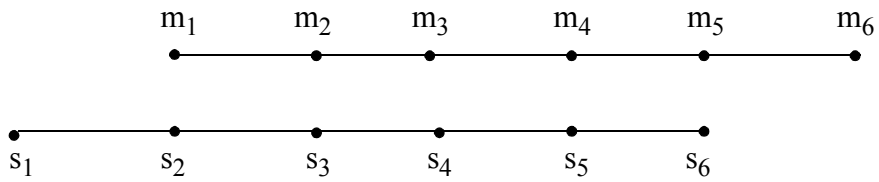


FIGURE 3-16

In response to the updated orientation, the contact domains during cycle n will be as shown in the following figure:

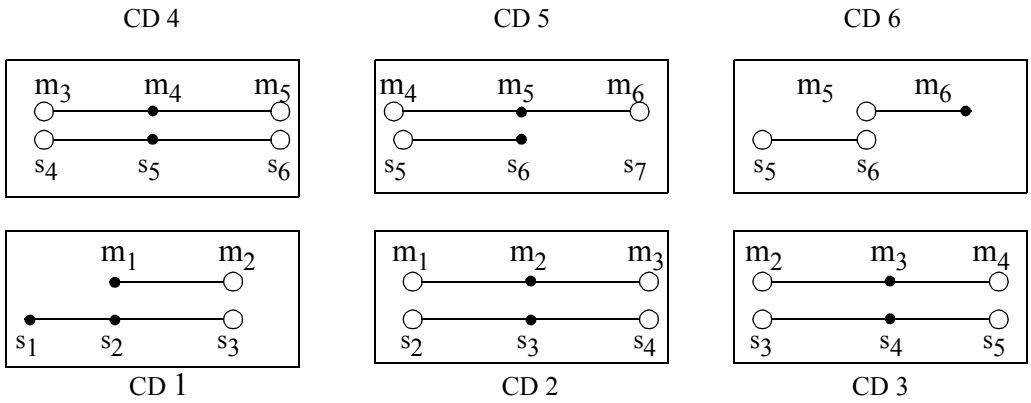


FIGURE 3-17

Now we will go back and examine what happened and how it happened. We focus on one particular event. During cycle n-1 BD 2 doesn't send anything to CD 1 (Figure 3-14). BD 2 → CD 1 is not part of the communication topology. During cycle n, BD 2 sends s_3 to CD 1, so now BD 2 → CD 1 *is* part of the topology. In bringing about this change the following sequence of events occurred:

1. During cycle n-1, CD 2 found that after the order node update s_2 ordered on m_1.
2. CD 2 found that m_1 was a ghost node assigned to CD 1.
3. during CD↔CD communication, CD 2 sent s_2's neighbor BD list (i.e. {BD 1, BD 2}) to CD 1, since it determined that s_2 would be assigned to CD 1 next cycle.

4. CD 1 included s_2's neighbor BD list (which included BD 2) in its next time BDs list of bulk domains to expect a message from next cycle.
5. During CD→BD communication, CD 2 returned s_2's updated state to BD 1, including the information that s2's new order node was m_1.
6. BD 1 looked up m_1's static contact assignment, found it was CD 1, and so set s_2's rule 1 CD to CD 1. This ends cycle n-1.
7. At the start of cycle n, during the BD↔BD phase, BD 1 sent the rule 1 CD of s_2 to BD 2, where s_2 is a proxy node.
8. BD 2 found that its proxy node s_2 had CD 1 as its assigned (rule 1) CD, found all s_2's local neighbors (just s_3 in this case), added CD 1 to the list of CDs to send to, and added s_3 to the list of nodes to send (as ghost) to CD 1.

In step 4, CD 1 found that it would get data from BD 2. In step 8 BD 2 found that it would send data to CD 1. So BD 2 → CD 1 gets added to the BD→CD topology at cycle n.

Applying the forgoing logic also causes communication paths to disappear when they are no longer needed. For instance, during cycle n-1 CD 3 received s_2 from BD 1. In the next cycle CD 3 no longer needed s_2, and consequently BD 1 → CD 3 dropped out of the topology.

3.14 Summary

The method described in this chapter creates a dynamically evolving communication pattern in support of contact enforcement on a parallel distributed-memory machine. This is accomplished by separating the communication into three communication topologies, each of local type.

The BD↔BD communication ties the bulk domains together by sharing information near mutual boundaries. This allows a given bulk domain to "sense" when it is about to become involved in the work of a specific contact domain, by involvement of its proxy nodes.

the BD↔CD communication supplies the contact domain with the data needed to perform the contact enforcement, to update the order nodes of the CD's assigned nodes, and to predict, on the basis of the updated order nodes, which bulk domains will communicate with it next cycle.

The CD↔CD communication supports the contact domain in determining which slaves order on its assigned master nodes, and thus which bulk domains the contact domain will communicate with next cycle. CD↔CD communication also permits less redundant calculation of ghost state, by trading communication for redundant calculation.

These three communication topologies work together to obviate the need for global communication in determining the constantly changing interface between the two sides of a contact surface.

This chapter provides a summary of the method developed during this research. A much more detailed description of the actual implementatin is provided in Appendix A.

4 TIMING STUDIES

In order to test the scalability of the contact algorithm as implemented in ALE3D several timing studies were done. The test geometries are rectangular, cylindrical, and spherical, and have been designed to isolate the issues related to contact scalability.

Both scaled efficiency and fixed efficiency (Section 2.2) are measured. For the scaled efficiency tests the mesh refinement strategy employed in the rectangular and cylindrical tests is unconventional, so a few words of explanation are in order. Ordinarily when scaling a problem, the geometry (i.e. the overall shape defined by the boundaries) is constant, and the mesh is refined by putting more, smaller zones in each dimension, while keeping the zone geometry as "regular" (i.e. cubic) as possible. In practice this can be difficult, since the geometry can change radically as time evolves, while the connectivity is constant. For example see Figure 4-14. One does the best one can.

However, in these tests (excluding the spherical tests) the mesh refinement is only done in two dimensions, while the problem geometry is shrunk in the remaining dimension in order to keep the zones more or less cubic. This is done to provide a more stringent test of the scalability of the contact algorithm. Since contact surfaces are two dimensional, and

the overall problem is three dimensional, in conventional scaling the number of contact surface elements grows only as $N^{2/3}$ of the overall element count N. For scaled speedup in which the overall amount of work per processor is held constant, contact work per processor actually shrinks. This clouds the issue of scalability of the contact algorithm itself. Ideally one would hope to see super-linear scaling of the contact phase in this case, but how should one quantify it?

The simplest solution is to hold the contact work per processor constant, in which case it is clear that the ideal speedup is 1. And the simplest way to accomplish constant contact work per processor is to perform mesh refinement only in the dimensions of the contact surface.

For the fixed efficiency tests a fixed size of problem is chosen, and then is run on various numbers of processors. Ideally each doubling of processors will halve the run time. Inevitably this ideal will fail at some point, as the amount of actual work per processor becomes small compared to the time spent in various fixed-overhead work, such as communication latency times. The problem sizes chosen exhibit this failure at the largest processor counts run. This is deliberate. In each case a larger problem would have fit into the minimum processor count used (8) and would have better concealed this behavior, but to what point? It is better to demonstrate and discuss what is an inevitable feature of the fixed efficiency measure of scalability.

All tests were performed on the LLNL ASCI "Y" machine, which is an IBM SP system with 488 nodes and 1.5 GBytes of memory per node. Each node consists of 4 PowerPC 604e processors, running at 332 MHertz with a peak of 664MFlops, that share the node's

memory, for a total of 1952 processors. The version of ALE3d used was Version 3.1.73, with some modifications to the static master decomposition (the modified version is described in Section A.6.8.4), and some additional timing instrumentation.

4.1 An IBM SP Scalability Issue

In running the tests described in the following sections it was found that, though the problems scaled well, they did not scale as well as expected. In going from 8 to 1024 processors, a factor of 128, the run times for the scaled tests approximately doubled or tripled.

All tests were run for 200 cycles. In running with the larger processor counts it was observed that there was both a steady increase in typical cycle times, and an increasing variance in the individual cycle times. Some cycles took up to four times as long as others. This occurred even in the fixed test, in which identical work and communication was performed each cycle. Furthermore, which cycles took longer on a given run was entirely unrepeatable, though the general pattern appeared on each run.

This behavior is believed to be a result of system "daemon" processes interrupting various processors occasionally to perform housekeeping tasks. If each processor ran entirely independently, assuming these system interruptions are uniformly distributed, then they would not affect scalability. However when the processors communicate, then if one processor is interrupted, all processors it communicates with must wait. This effect is most dramatic for global operations such as MPI_Barrier and MPI_Allreduce, but also affects local communications.

To see this, consider figure Figure 4-1.

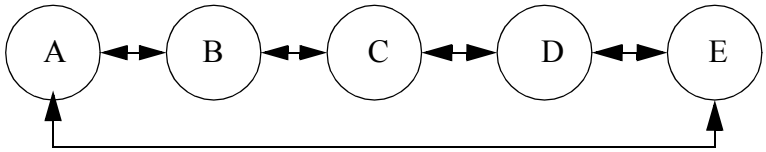
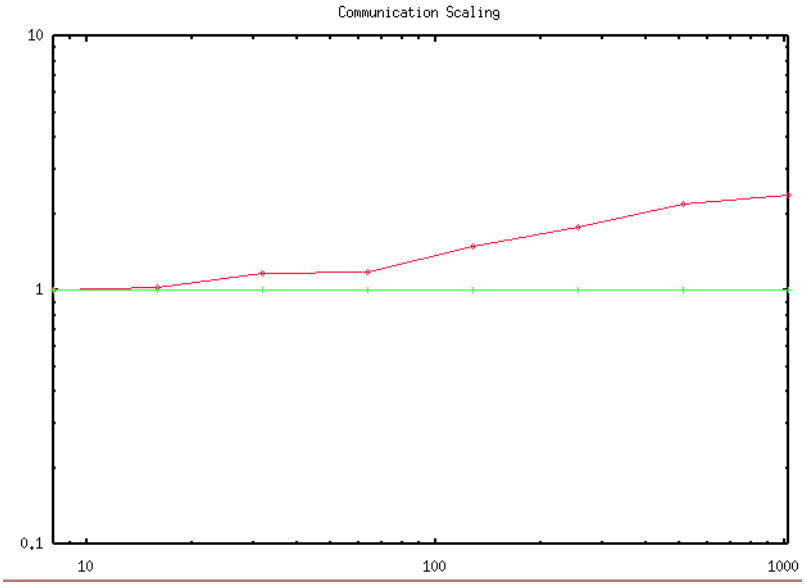


FIGURE 4-1

Each processor in this example communicates only with its nearest neighbor on either side. Suppose processor A is interrupted and temporarily cannot send to B and E the data they are waiting for. Then at some point (very soon!) B and E can go no further and cannot send to C and D the data they are waiting for. Thus C and D must also wait for A to return from the interruption. This "daisy-chain" can be of any length: all processors must wait for A. As the processor count is increased, the probability of a processor being interrupted by the system in any given time interval increases, resulting in the increased run times observed as well as the variance in cycle times.

To measure this phenomenon a simple test was constructed. In this test, each processor communicates with 6 neighbor processors. For simplicity the neighbors are the six with processor numbers (MPI rank) closest to the given processor. Each cycle each processor sends and receives 2000 floating-point numbers to its neighbors. The test is run for 1000 timesteps. Ideally this test should take the same amount of time regardless of the total

number of processors involved. The actual results are shown (normalized) in figure Figure 4-2. Each number is the average of 4 runs.

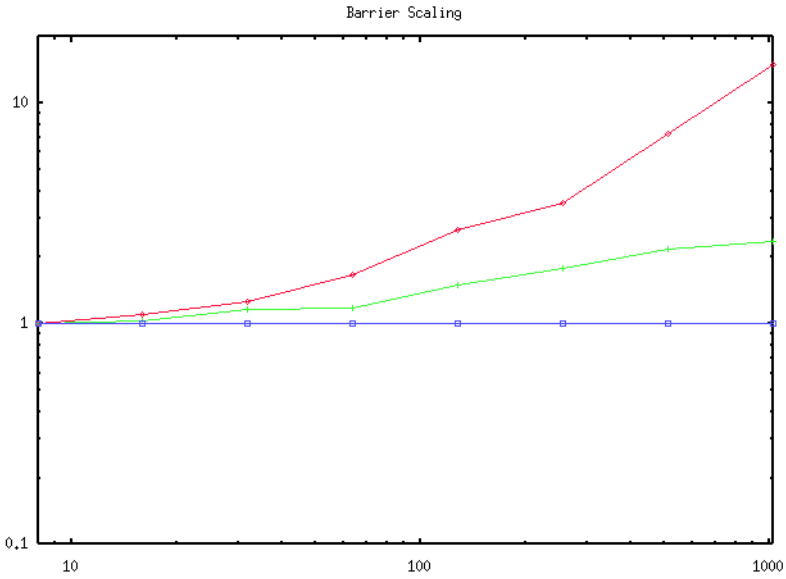


procs	time
8	1.0
16	1.03
32	1.17
64	1.18
128	1.49
256	1.78
512	2.18
1024	2.31

FIGURE 4-2

One may wonder if there is any point in making the effort to produce an algorithm that uses only local communication, if the result is no better scaling that this (at least on the current IBM SP). To answer this question a similar test was constructed which instead per-

forms a global operation (MPI_Barrier) within the loop. The results of this test are shown in Figure 4-3:



procs	time
8	1.0
16	1.10
32	1.26
64	1.66
128	2.65
256	3.52
512	7.31
1024	14.90

FIGURE 4-3

In this figure the times for local communications are reproduced, and the times for the barrier loop are added. Since barriers are the simplest form of global communication, in that they involve no data transfer, it is clear that global communication produces a much bleaker picture for scalability than local communication, and thus efforts expended in restricting communication to local interchanges only are worth while.

IBM has proposed two solutions to this problem, both of a draconian nature [42]. First, one can leave one processor of each node free to handle the system work, thus allowing the remaining processors to continue without interrupts. For the machine these tests were performed on, this would require giving up 25% of the CPUs. On the latest SPs with 16 processors per node this solution may be feasible. Second, one can set the priority of the user processes higher than the interrupting system processes. This solution is not available to the average user, and seems very dangerous in terms of system stability.

4.2 Fixed Planes

For the first set of tests the simplest possible geometry is used. Two rectangular boxes of equal size are oriented so that one sits on top of the other with perfect overlap. The contact surface is defined to be the surface where the blocks meet. The mesh for the two blocks is slightly different. There is one more zone in the x and y directions of the bottom block than the top block, so that the contact nodes do not coincide, except at the edges (the contact surface is orthogonal to the z direction). There is no movement in this test: the blocks just sit there. Nevertheless, the entire machinery of the contact algorithm, including all communication, is exercised.

This test is used to establish a baseline. Computational load balance should be very good, with any non-scalability attributable to communication. Figure 4-4 shows the geometry of the problem for a 1 processor run. Larger runs simply use a finer grid in x and y directions, while shrinking the box thickness in the z direction.

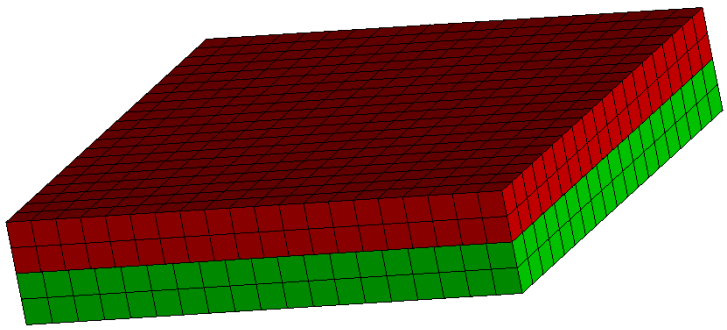


FIGURE 4-4

The following table gives the parameters and results for a set of timed runs. Each run is approximately twice the size of the previous run, and is run on twice as many processors. All times are normalized.

Procs	total zones	zones / proc	contact nodes	nodes / proc	contact time
8	25,282	3160	12,961	1,620	1.0
16	50,622	3164	25,763	1,610	1.11
32	101,762	3180	51,521	1,610	1.14
64	203,386	3178	102,597	1,603	1.24
128	408,322	3190	205,441	1,605	1.36
256	815,346	3185	409,481	1,600	1.51
512	1,635,842	3195	820,481	1,603	1.65
1024	3,264,994	3188	1,636,113	1,598	2.57

TABLE 1.

As expected from the discussion in Section 4.1, there is a gradual increase in time attributable to system interrupts leading to communication blocks. The overall increase in time is about 3x, for a 128x increase in problem size and processor count. The following figure compares graphically the actual scaled efficiency compared to ideal. The normalized maximum run time is shown, which is actually the inverse of scaled efficiency.

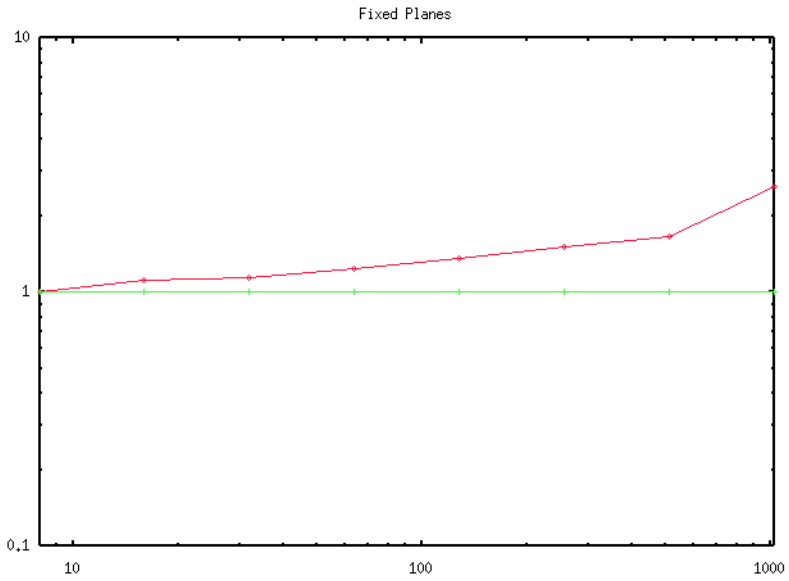


FIGURE 4-5

In addition to the scaled test, a fixed-size test was run with various processor counts. The problem chosen is identical to the one used for the scaled test with 256 processors. Results for the fixed test are compared with ideal scaling in Figure 4-6.

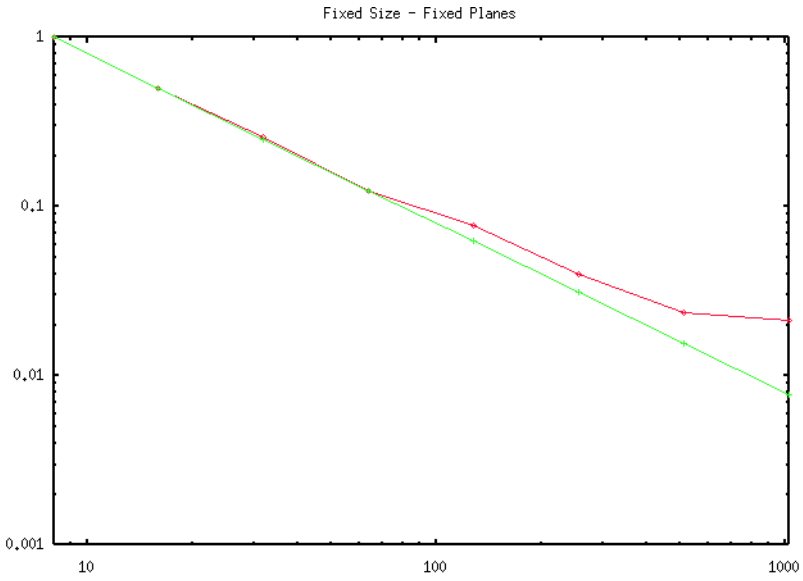
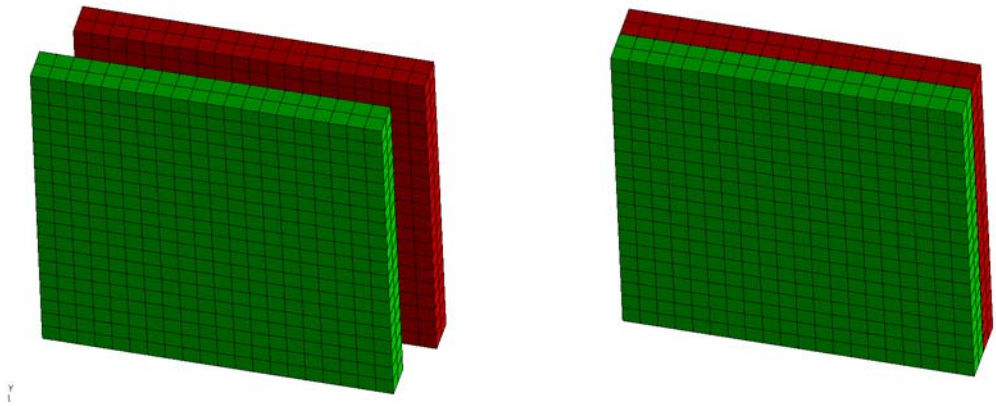


FIGURE 4-6

4.3 Sliding Planes

In this test two rectangular boxes are again modelled, but in this case they sit at an initial offset, as shown in Figure 4-7(a). One block is given a velocity, so that at a later time they are in the configuration shown in Figure 4-7(b).



(a) (b)

FIGURE 4-7

This test exercises the ability of the algorithm to change its communication pattern over time. The partitioning algorithm decomposes the problem into domains such that each domain is in one block or the other, not both. This is a consequence of the rule that attempts to minimize connectivity between domains. The most obvious cut is between the blocks, along the contact surface, because there is no connectivity between blocks.

As the block slides, the domains directly across from a given domain will change, altering the communication pattern. Aside from this, individual slave nodes are assigned to the contact domain of their associated master node, and as the block slides, slaves that were previously ordered on masters on the edge of a contact domain will change ordering to new masters in a different contact domain. Thus the individual data sent to contact domains changes each cycle, even when the pattern of communicating domains does not.

Table 2 and Figure 4-8 present the normalized results for the scaled sliding planes test:

procs	total zones	zones / proc	contact nodes	nodes / proc	contact time
8	25,600	3,200	13,122	1,640	1.0
16	51,072	3,192	25,990	1,624	1.10
32	102,400	3,200	51,842	1,620	1.20
64	204,288	3,192	103,050	1,610	1.40
128	409,600	3,200	206,082	1,610	1.78
256	817,152	3,192	410,368	1,603	2.15
512	1,628,400	3,200	821,762	1,605	3.05
1024	3,268,608	3,192	1,637,922	1,600	5.21

TABLE 2.

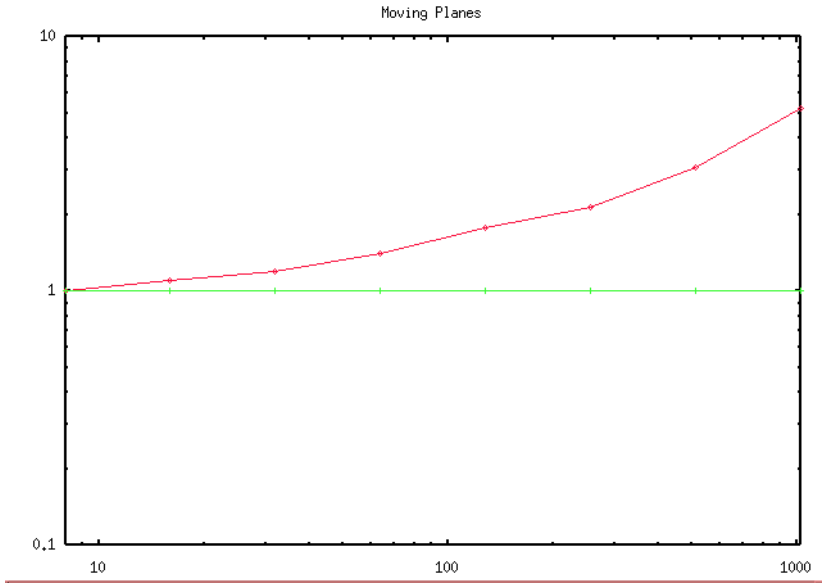


FIGURE 4-8

From the above results it is clear that this test does not scale as well as the fixed plane test. The source of the problem is poor load-balance. This is a consequence of the way in which the slave nodes are assigned to Contact Domains. As discussed in Section A.5, each slave node is dynamically assigned to the same Contact Domain as its statically assigned master order node. Since the blocks are offset, all the slave nodes in the offset region order on the

limited set of master nodes along the edge of the master side closest to the offset slave region. Figure 4-9 depicts a simplified one dimensional example. The slave side is on the bottom. Each partition along the top represents a different Contact Domain.

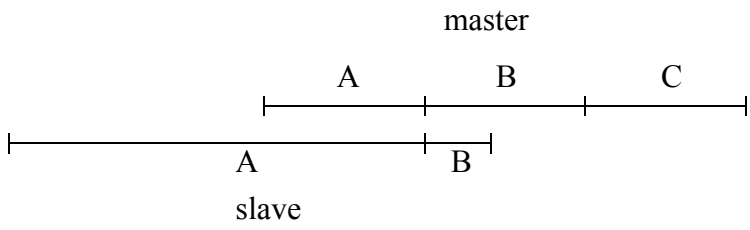


FIGURE 4-9

Though the master side is well load-balanced, most of the slave side is closest to master nodes in Contact Domain A. Thus most of the slave nodes are assigned to Contact Domain A. A much lesser number are assigned to B, and none are assigned to Contact Domain C. This results in severe load imbalance.

The imbalance gets worse as the problem is scaled up. In Figure 4-9 50% of the slave side lies to the left of the master side, and thus more than 50% of the slave side will be assigned to 33% of the Contact Domains. Now suppose that the mesh is refined 10 times, with a proportional increase in processor count, so that there are now 30 Contact Domains. More than 50% of the slave nodes will now be assigned to 1/30, or about 3%, of the Contact Domains.

This is a definite weakness in the current version of the contact algorithm. It can be "fixed" by using a different method of assigning slave nodes to Contact Domains, and is an area for future development.

Figure 4-10 shows the results for the fixed size scaling test for sliding planes. Again, the problem used is the same as the 256 processor scaled test. The straight line represents ideal scaling.

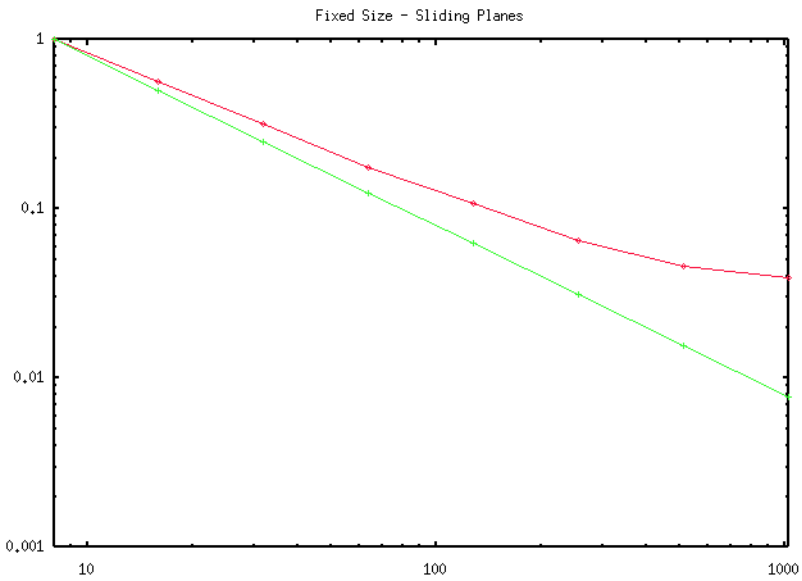


FIGURE 4-10

4.4 Cylinder Test

In this test two concentric cylinders are modelled, as shown in Figure 4-11. The outer cylinder is held fixed while the inner cylinder is given a constant angular velocity. In this problem each part of the contact surface eventually comes into contact with many other parts of the surface, so that there is a large amount of change in the communication pattern over time. It does not have the load-imbalance problem of the sliding planes in the last

section, so scalability should be improved. Over the course of the test one full revolution of the interior cylinder is modelled.

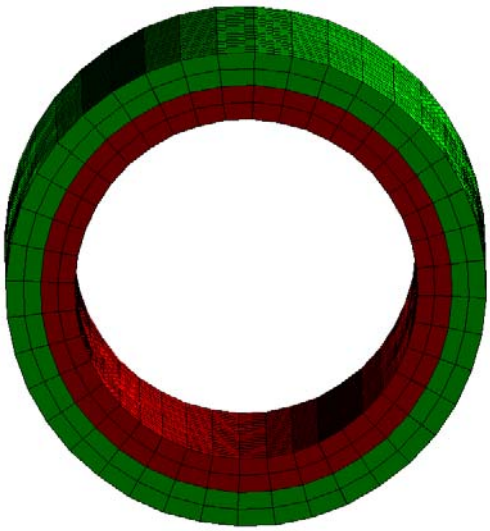


FIGURE 4-11

The results for this test are shown in Table 3 and Figure 4-12.

procs	total zones	zones / proc	contact nodes	nodes / proc	contact time
8	12,800	1,600	6,642	830	1.0
16	25,536	1,596	13,110	819	1.01
32	51,200	1,600	26,082	815	1.13
64	102,144	1,596	51,754	809	1.21
128	204,800	1,600	103,362	808	1.40
256	408,576	1,596	205,650	803	1.56
512	819,200	1,600	411,522	804	1.85
1024	1,634,304	1,596	819,847	801	2.37

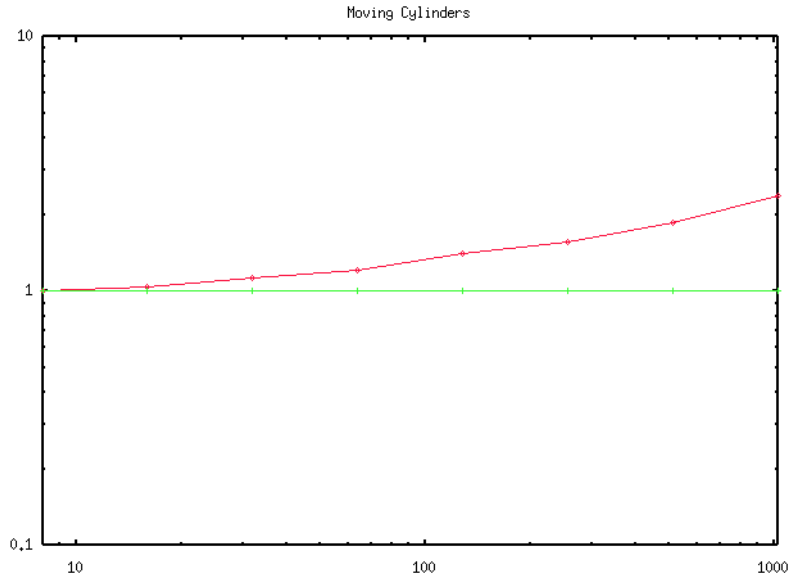


FIGURE 4-12

As expected, this test scales quite well, within the limits imposed by the scaling issues discussed in Section 4.1.

Figure 4-13 shows the corresponding fixed size results, for the problem used in the 256 processor scaled test:

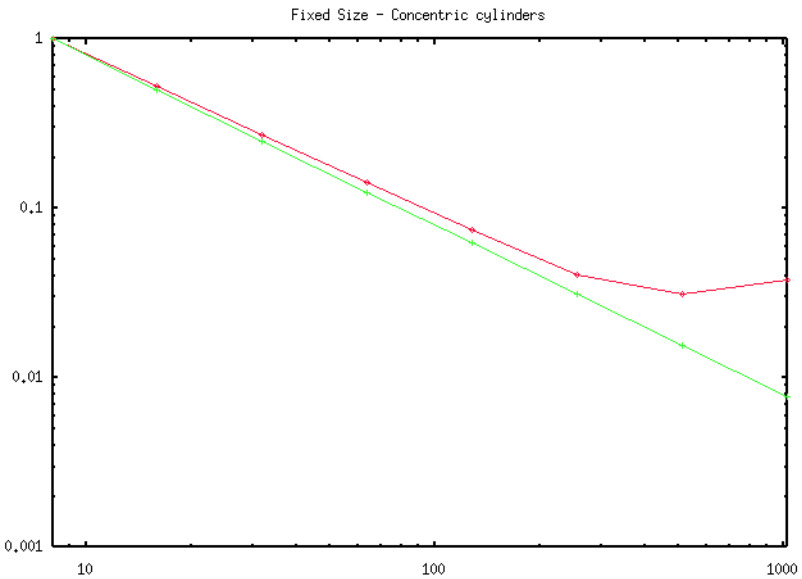


FIGURE 4-13

It can be seen from this that the fixed-size problem becomes communication-bound past 512 processors. This is not surprising, as there are less than 400 contact nodes per Contact Domain at this point, which leaves each processor without a lot of computation to balance the communication overhead.

4.5 Concentric Spheres Test

This last test involves more physics than the previous tests, and can be used to test the accuracy of the overall simulation. In this test there are two contact surfaces, separating three concentric spherical shells. The outer and inner shell are a metallic material, while the middle shell is an explosive material. Only one octant of the material is modelled, with symmetry boundary conditions.

At the initial time the explosive is ignited, leading to rapid expansion of the material. Figure 4-14 shows the initial problem configuration. While there is not a lot of movement along the contact surfaces, there is some, and to capture the correct physics it is necessary

to allow the free motion. The most important requirement on the contact surfaces in this application is the need to accurately model shock waves as they cross the surface.

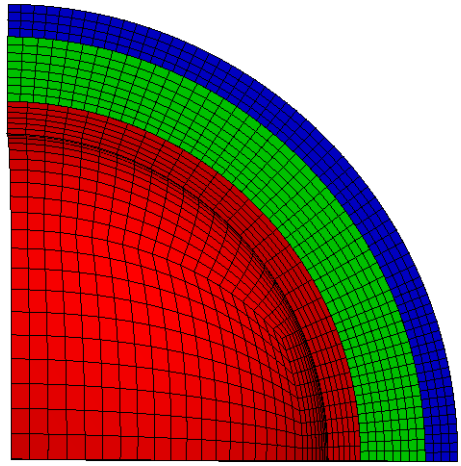


FIGURE 4-14

Unlike the previous tests, as this test is scaled up the ratio of contact nodes to overall zones decreases, as shown in the following table of problem parameters and results.

Procs	total zones	zones / proc	contact nodes	nodes / proc	contact time
8	21,240	2,655	5,312	664	1.0
16	42,350	2,646	7,669	479	0.90
32	84,860	2,652	12,936	404	0.90
64	169,920	2,655	20,740	324	1.09
128	338,800	2,646	30,064	235	1.11
256	678,880	2,652	50,948	199	1.37
512	1,359,360	2,655	81,956	160	1.62
1024	2,710,400	2,646	119,044	116	1.93

TABLE 3.

Initially the shrinkage in the amount of actual computation leads to a reduced run time in the above results, but as the processor count increases the amount of computation becomes

less significant relative to the communication time, which suffers the mild non-scalability discussed in Section 4.1. A plot of the timing results is presented in Figure 4-15:

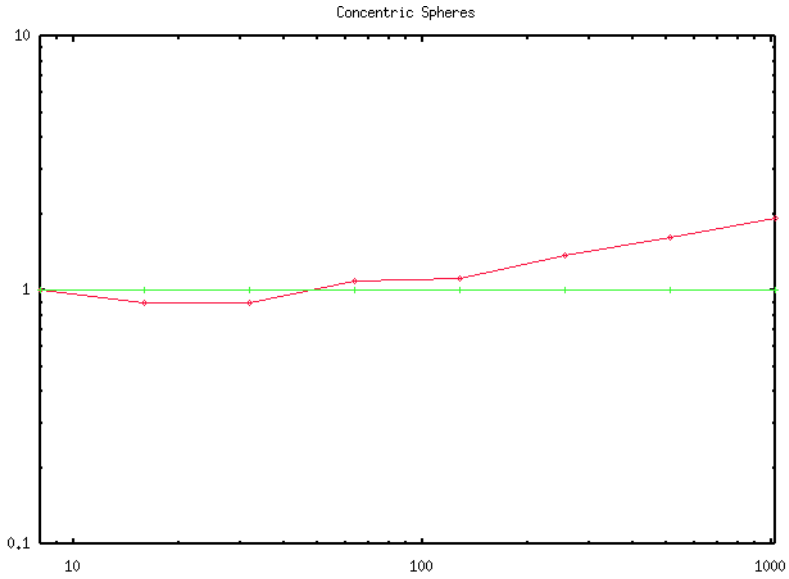


FIGURE 4-15

The following Figure 4-16 shows the results of a fixed size scaling test. As in the other tests, the problem size used above for 256 processors is used for all processor counts of the fixed size test.

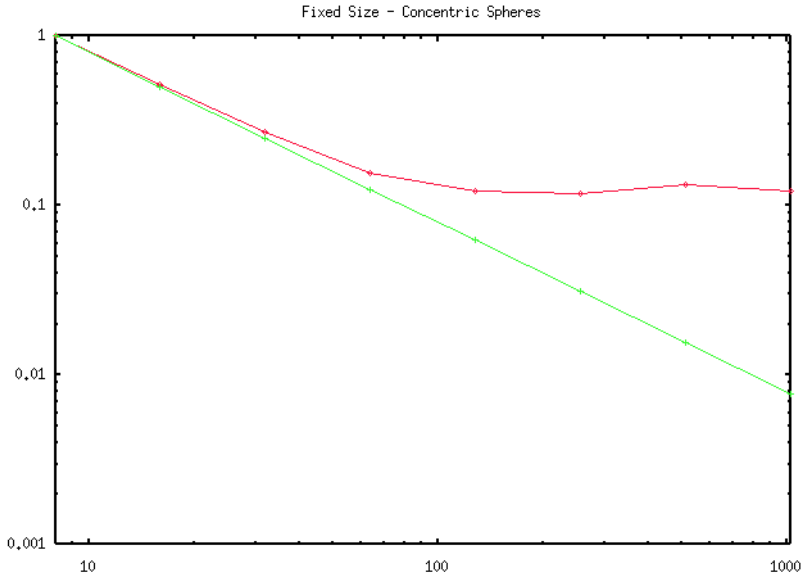


FIGURE 4-16

5 SUMMARY AND FUTURE DIRECTIONS

This dissertation has detailed the design and implementation of a parallel algorithm for modelling the behavior of multiple bodies in contact with each other. Unlike the fairly straight-forward task of parallelizing the computation of a single body, the modelling of multiple bodies results in a communication pattern which changes dynamically and unpredictably in time.

After presenting a survey of past work in modelling dynamic processes with the class of programs known as hydrocodes, with special emphasis on the handling of contact surfaces, and reviewing various efforts to parallelize this task, the dissertation launches into a specific implementation which is designed to accomplish this task on the current class of distributed memory parallel computers in a highly scalable manner. Successful implementation requires that all global communication be banished, and the tracking of the changing configuration be accomplished via local communications only. These patterns are themselves in flux. Success in this endeavor is demonstrated through a series of timing studies using various problem geometries.

The algorithm described herein has been incorporated into the ALE3D code system, a multiphysics finite element hydrocode that is under constant use and development at Lawrence Livermore National Laboratory. There are several areas in which the applicability and scalability of the basic algorithm might be extended in future development. One direction involves the use of computer memory. For the initial implementation, several arrays that are multiples of the length of the entire contact surface must be present in each processor. As the problem size is scaled up, a point must ultimately be reached where these arrays are too large to fit into memory. This problem was analyzed early in the development of the parallel contact algorithm, and it was determined that it would not be an issue for the current and projected problems to be studied by ALE3D over the next several years. This is due to the two dimensional nature of contact surfaces in three dimensional problems. For example, in a problem with 100 million zones, the number of contact faces is on the order of 500,000. Still, the history of computers has taught us that what is considered enormous at one point in time may become a typical size within a few years.

In order to deal with this issue it will be necessary to partition the contact surface during the preprocessing phase when the overall problem is partitioned, rather than performing the entire contact decomposition at run time. Each processor would then only have such bits and pieces of arrays such as the contact surface connectivity and master side decomposition as it required.

Another useful direction for future development is touched on in Section 4.3 when it was seen that in some circumstances the current master static / slave dynamic decomposition produced poor load-balance. One application area that highlights this shortcoming is the impact problem, since in the initially separated positions a large fraction of the slave nodes

may order on a small fraction of the master nodes, leading to disproportional distribution of the slaves across contact domains.

A potential solution to this problem would be to develop a decomposition which statically assigns slaves as well as masters to contact domains. In many situations this would be inefficient, leading to increased communication (see the discussion in Section A.5.2). However, in other situations the improved load balance would more than offset this. So the optimal solution would be to offer this capability as yet another decomposition that the user can choose to fit the circumstances.

ALE3D is a living code. Several enhancements have been developed since the initial work documented in this dissertation was done, and more will follow. However, it is fair to say that this work falls into the category discussed in the appendix on ALE3D extensions, and does not contradict the fundamental algorithmic descriptions in the body of the dissertation.

APPENDIX A - ALE3D IMPLEMENTATION

A.1 Overview

In its ALE3D implementation, the contact algorithm is part of a larger picture. First the problem geometry is defined via a mesh generator. Then the mesh is divided in a number of subdomains, called ***bulk domains***, and various material properties and initial and boundary conditions are specified.This is all part of the pre-processing stage.

At runtime, each bulk domain is assigned to a separate processor (see, however, Section B.1) and a number of initialization steps are performed to read in the problem definition, set up communication along bulk domain boundaries, initialize the slide decomposition, etc.

Once the main loop is entered, each processor independently calculates the forces on each node in its bulk domain, based on the stress in each mesh zone surrounding the node. From these forces and the zonal masses, which are a lumped representation of the mass of surrounding zones, nodal accelerations are arrived at.

Up to this point, nodes on the contact surface are treated as though they were on a free surface. The computed accelerations do not take into account force exerted on contact nodes by the opposing side of the slide interface.

During the contact phase, nodes on the slide surface are reassigned to ***contact domains***, where each contact domain is associated with its own processor (from the same set of processors as the bulk domains). If a node is assigned to a given contact domain then it, its immediate neighbors on its side of the surface, the closest node on the opposing side of the surface, and immediate neighbors of the closest opposing node (collectively referred to as the ***locale*** of the node in Chapter 3) must all be sent to the given contact domain. The purpose of this is to supply the contact domain with sufficient information to complete the acceleration calculation for the node, taking into account force exerted by the opposing side.

After the modified acceleration is computed for a contact node, it is used to calculate a new velocity and position for the node. At the end of the contact phase the velocities and positions of all contact nodes are returned to their bulk domains, which can be considered their permanent homes.

With updated nodal velocities and positions for all nodes (bulk and slide) available on the bulk domain, the new nodal stresses can be evaluated using the stress-strain relationship of the selected material model, and the next cycle can begin.

A.2 Definitions

In this section a formal approach is taken, in order to precisely define the terms that will be used throughout the rest of this appendix.

A.2.1 The Continuum model

Let Ω be a continuous, bounded, simply-connected closed region of R^3. Ω will be used to represent the space occupied by a single material body, solid or fluid, at a given time t.

For a point $x \in R^3$, let $N_r(x)$ be the set of all points $y \in R^3$ such that $|x - y| < r$. $N_r(x)$ is called the ***open neighborhood*** of x of radius r.

The boundary Γ of Ω consists of those points $x \in \Omega$ such that every open neighborhood of x contains points not in Ω: $\Gamma = \{x \in \Omega | N_r(x) \not\subset \Omega, \forall r > 0\}$

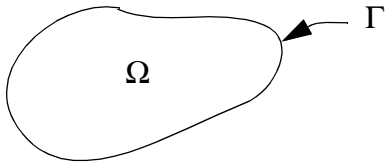


FIGURE A-1

The interior int(Ω) is defined to be $\Omega - \Gamma$, the complement of Γ in Ω.

$int(\Omega) = \{x \in \Omega | \exists r > 0 \ni N_r(x) \subset \Omega\}$

Let $\Omega_i | i \in \{0 \ldots N-1\}$ be a set of regions as defined above, such that $\bigcap_i int(\Omega_i) = \varnothing$.

Their corresponding boundaries are $\{\Gamma_i | i \in \{1 \ldots N\}\}$

Definition A.1: A ***Contact Surface*** $C_{i,j}$ *is an ordered pair of boundaries* (Γ_i,Γ_j) *along which contact may occur. At a given time t the actual contact is restricted to the set* $\Gamma_i \cap \Gamma_j$.

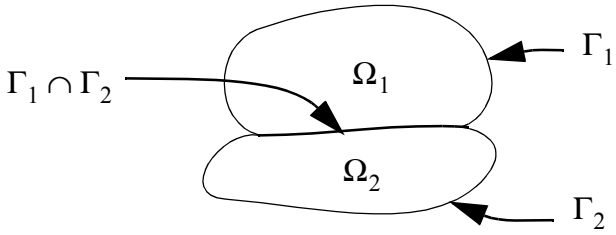
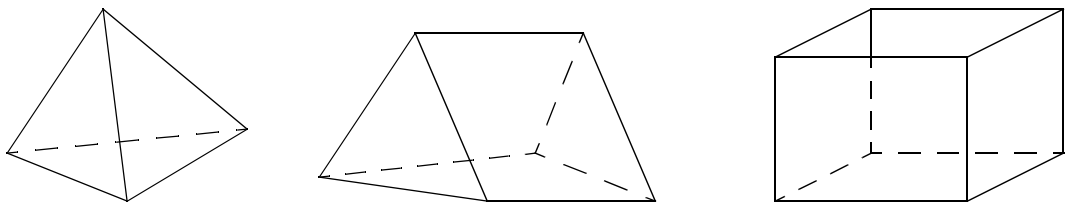


FIGURE A-2

A.2.2 The Finite Element Mesh

Polyhedra are 3-D objects with multiple distinct faces, edges, and nodes (also called vertices or corners). Common polyhedra in modelling applications are tetrahedra (Figure A-3a), triangular prisms (Figure A-3b), and hexahedra (Figure A-3c). A polyhedron is defined by its type and by the coordinates of its nodes. Note that while triangular faces are necessarily planar, quadrilateral and higher-order faces need not be planar.



(a) (b) (c)

FIGURE A-3

Definition A.2: Let $E = \{e_i | i \in I_E\}$ *be a set of polyhedra that approximately cover* Ω*, such that* $\bigcap_{i \in I_E} int(e_i) = \varnothing$ *and* $\bigcup_{i \in I_E} e_i \cong \Omega$*. E is called the* ***mesh*** *or* ***grid****, and the* e_i *are the* ***mesh elements****,* ***zones****, or* ***cells.*** I_E *is the*

index set for the elements: $I_E = \{0 \ldots N_E - 1\}$, *where* N_E *is the total number of elements in E.*

The mesh is constructed so that each element face either a) coincides with the face of a neighboring element, or b) is a boundary face of the mesh. Two faces coincide if all the nodes defining each face coincide. There is no partial overlap of faces.

Definition A.3: Let F_{e_i} *be the set of faces of element* e_i, *and* $F = \bigcup_{i \in I_E} F_{e_i}$ *be the set of all faces in the mesh.* $F = \{f_j | j \in I_F\}$ *is called the* ***face list*** *of the mesh.* I_F *is the index set* $\{0 \ldots N_F - 1\}$, *where* N_F *is the total number of faces in* F.

Definition A.4: Let V_{f_j} *be the set of nodes of face* f_j.

Definition A.5: Let V_{e_i} *be the set of nodes of element* e_i. $V_{e_i} = \bigcup V_{f_j}, \forall f_j \in F_{e_i}$

Definition A.6: Let $V = \bigcup_{i \in I_E} V_{e_i}$ *be the set of all nodes in the mesh.*

The preceding definitions lead up to the introduction of the connectivity and dual connectivity, which specify how the various elements are connected together in the overall mesh.

Definition A.7: The ***connectivity of the element set E*** *is defined to be the set* $\{V_{e_i} | i \in I_E\}$.

Note that the connectivity is an indexed set of sets. Consider two set members V_{e_p} and V_{e_q}. Each is a list of nodes. By comparing the nodes in each list it can be determined if the

two elements e_p and e_q are connected, and if so, whether they share a single node, an edge (two nodes), or a face.

Definition A.8: Let $v_k \in V$, *and define* $E_{v_k} = \{e_i | v_k \in V_{e_i}\}$.

For each v_k, E_{v_k} is the set of all elements that include the node v_k.

Definition A.9: The ***dual connectivity of the element set E*** *is defined to be the set* $\{E_{v_k} | k \in I_V\}$.

I_V is the index set $\{0 \ldots N_V - 1\}$, where N_V is the total number of nodes in the mesh E. The dual connectivity is also an indexed set of sets.

The element set connectivity expresses the relationship between elements and nodes. A related construct defines the relationship between faces and nodes.

Definition A.10: The ***connectivity of the face set F*** *is defined to be the set* $\{V_{f_i} | i \in I_F\}$.

Definition A.11: The ***Dual connectivity of the face set F*** *is defined to be the set* $\{F_{v_k} | k \in I_V\}$.

In the above definitions of connectivity and dual connectivity, the element set E and face set F may be any arbitrary element and face sets.

Definition A.12: A ***mesh path*** *between two nodes* v_k *and* v_l *is a sequence of elements* $\{e_i \ldots e_j\}$ *such that* $v_k \in V_{e_i}$, $v_l \in V_{e_j}$ *and for each adjacent pair of elements* e_p, e_q *in the sequence, there exists a node* v_m *such that*

$v_m \in V_{e_p} \cap V_{e_q}$. *The length of the mesh path is the number of elements in the sequence.*

In order to traverse a mesh path, the connectivity and dual connectivity are applied in alternating order. For example, a v_k is first selected. Then the dual connectivity is used to choose an $e_j \in E_{v_k}$. The connectivity is then applied to e_j to select a different node $v_n \in V_{e_j}$, and the process is repeated over the length of the path.

Definition A.13: The ***mesh neighborhood*** $[v_i]^r$ *of a node* v_i *of radius r, is the set of all nodes* v_x *for which there exists a mesh path between* v_i *and* v_x *of length* $L \leq r$.

In the above definition, r may be preselected and held constant throughout a discussion, in which case $[v_i]^r$ is simplified to $[v_i]$, with r implicit.

Let $int(F) = \{f_k \in F | \exists i, j, \quad i \neq j \ni f_k \in F_{e_i} \cap F_{e_j}\}$. $int(F)$ consists of the set of all faces in F that are shared by two elements.

Let $B = F - int(F)$. B is the set of boundary faces of the mesh, which have an element on only one side, the other side being a "free surface".

Definition A.14: Let E_1 *and* E_2 *be unconnected meshes, with boundary face sets* B_1 *and* B_2. *A* ***double-sided contact surface*** *is an ordered pair of face sets* (M, S), *where* $M \subset B_1$ *and* $S \subset B_2$

M is referred to as the ***master side***, and S is referred to as the ***slave side***, of the contact surface. Single-sided contact surfaces for which a single set of faces is defined, and contact permitted between faces of the single set, are not considered in this document, and so the term "contact surface" will be used to refer to a double-sided contact surface.

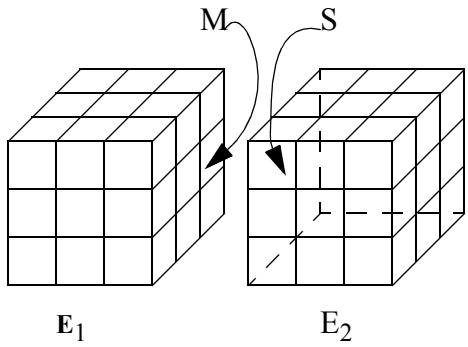


FIGURE A-4

In Figure A-4, two disconnected meshes E_1 and E_2 are shown. Each has 54 boundary faces. From these the 9 rightmost boundary faces of E_1 constitute the master side M, while the leftmost 9 boundary faces of E_2 constitute the slave side S.

A.2.3 Contact Proximity Relations

In order to enforce the contact constraints, it is necessary to know at all times the spatial relationship of the two sides with respect to each other. This is accomplished by determining at each timestep, for each node on each side of the surface, the closest node on the other side of the surface.

Let $V_M = \left\{ v_i \middle| v_i \in \bigcup_{f_k \in M} V_{f_k} \right\}$, and let $V_S = \left\{ v_i \middle| v_i \in \bigcup_{f_k \in S} V_{f_k} \right\}$. Note that

$V_M = \{ m_i \middle| i \in I_{V_M} \}$ and $V_S = \{ s_i \middle| i \in I_{V_S} \}$, where $I_{V_M} = \{ 0 \ldots N_{V_M} - 1 \}$, N_{V_M} is the total number of master nodes, and $I_{V_S} = \{ 0 \ldots N_{V_S} - 1 \}$, N_{V_S} is the total number of slave

nodes. The m_i, s_i nomenclature will be used in preference to the more general v_i to specifically refer to master or slave nodes, respectively.

Definition A.15: The ***order node mapping is a mapping*** $X:V \to V$ *such that:*

4. $X:V_M \to V_S$ *is defined by* $s_j = X(m_i)$ *such that* $\forall k \in I_{V_S}, |s_k - m_i| \geq |s_j - m_i|$.
5. $X:V_S \to V_M$ *is defined by* $m_j = X(s_i)$ *such that* $\forall k \in I_{V_M}, |m_k - s_i| \geq |m_j - s_i|$.

$X(m_i)$ is called the ***order node*** *of m_i. Similarly, $X(s_i)$ is called the order node of s_j. If more than one node meets the criterion to be an order node, one of them is arbitrarily selected.*

A.2.4 Domain Decomposition

Definition A.16: A ***Domain Decomposition*** *is a partitioning of a mesh into a number of submeshes, called domains.*

For the purpose of this discussion, multiple disconnected meshes will be considered as one generalized mesh E. Also, a single contact surface will be assumed. The extension to multiple contact surfaces is straight-forward.

Three different decompositions, and three resulting sets of domains, are considered.

Definition A.17: The ***Bulk Decomposition*** *partitions the set of elements* $E = \{e_i | i \in I_E\}$ *into a number N_{BD} of* ***bulk domains*** $\{E_j | j \in \{0 \ldots N_{BD} - 1\}\}$, *where* $\bigcup E_j = E$ *and* $\bigcap E_j = \varnothing$.

Typically one bulk domain, or BD, will be assigned to each processor of a parallel machine. However, this is not necessary, and in some situations multiple BDs may be assigned to each processor to achieve better computer cache behavior, in a method known as domain overloading.

Let $F_{E_i} = \{f \in F | f \in F_{e_j} \text{ for some } e_j \in E_i\}$. Note that F_{E_i} is the restriction of the face set F to faces in the bulk domain E_i.

*Definition A.18: the **Primary Decomposition** is a restriction of the Bulk Decomposition to contact faces and nodes. That is, each **primary domain** is the subset of the corresponding bulk domain restricted to contact nodes and faces.*

For a given bulk domain E_i, its corresponding primary domain is the face set

$F_i = \{f_k | f_k \in F_{E_i} \cap (M \cup S)\}$ with corresponding node set

$V_{F_i} = \{v_j | v_j \in V_{f_k} \text{ for } f_k \in F_i\}$.

*Definition A.19: $v_j \in V_{F_i}$ is called a **local node** of primary domain F_i.*

*Definition A.20: $v_j \notin V_{F_i}$ is called a **proxy node** of primary domain F_i, if*

$$\exists v_k \in V_{F_i} \ni v_j \in [v_k]^r, \text{ for a prespecified radius } r.$$

In words, a proxy node is a node that, while not a local node, is in the neighborhood of a local node.

For the purpose of the contact algorithm, the bulk decomposition and thus the primary decomposition are given. Each is statically determined at the start of execution. This is not true of the following decomposition.

Definition A.21: The ***Contact Decomposition*** *is a partitioning of the contact nodes* $V_M \cup V_S$ *into* N_{CD} ***contact domains****.*

Note that $V_{M_i} = \{m_j \in V_M | m_j \text{ is in the ith contact domain}\}$, and $V_{S_i} = \{s_j \in V_S | s_j \text{ is in the ith contact domain}\}$, so that the contact domain C_i is defined by $C_i = V_{M_i} \cup V_{S_i}$

Let $I_{CD} = \{0 \ldots N_{CD} - 1\}$, the index set of contact domains. Then $C_i \cap C_j = \varnothing$ for $i, j \in I_{CD}$, $i \neq j$, and $\bigcup_{i \in I_{CD}} C_i = V_M \cup V_S$.

Definition A.22: $v_j \in C_i$ *is called an* ***assigned node*** *of contact domain* C_i.

Definition A.23: $v_j \notin C_i$ *is called a* ***ghost node*** *of contact domain* C_i*, if* $\exists v_k \in C_i \ni v_j \in [v_k]^r$*, for a prespecified radius* r.

In words, a ghost node is a node that, while not an assigned node, is in the neighborhood of an assigned node.

A.3 Naming and Index Sets

In the preceding section a number of index sets were defined. For instance, the set I_{V_S} is the index set of nodes on the slave side of the contact surface, so that s_i represents a slave

node for $i \in I_{V_S}$. Index sets figure prominantly in the following discussion, so it is important to be clear about the definition and relation of the various index sets. This section expands on the role and relationships of these index sets, from a computationally-oriented point of view. In the following, the term "index set" will be replaced by the more computer-oriented term ***index space***, but the two terms are synonymous

A.3.1 Naming

In order to operate on data from memory, its address must be known. In most computer languages the user associates a name with the specific data (for instance, "temp1", "XPOS[125]"), and the operating software keeps track of the actual storage location. When there are a large number of the same type of entity, for instance the X coordinates of all the nodes in a domain, it is common to use "arrays" or "lists" to store the set of data. The name for a particular datum then consists of two parts: its array name and its index. The index specifies which item in the list it is. For instance, the array name may be XPOS for the X coordinates of domain nodes and the name for the X coordinate of the $126^{\underline{th}}$ node, XPOS[125] (starting at zero).

A given node has a variety of data associated with it: coordinates, velocity, acceleration, lumped mass, etc. There can be separate arrays for each of these, or all data can be collected into an array of structures. In the first case, we can insist that the same index in the various arrays be used for the various properties of a specific node. In the second case, the point is moot; there is only one array, the various properties ***must*** share the same index. It is only a small abstraction to define the index to be the name of the node, even if, as in the first case, there is no location at which the node itself exists, but only a number of array

elements comprising its properties. It will be seen below that there need not even exist property arrays for the notion of a node index to be useful.

A.3.2 An Example

In order to illustrate the use of the various index spaces to be discussed below, an example is offered. In Figure A-5 a mesh is shown for two simple blocks with a contact surface between them. The upper surface of the lower block is defined to be the master side of the contact surface, while the lower surface of the upper block is the slave side. The bulk decomposition of the problem into four bulk domains is shown by the bold lines. The lower left bulk domain will be called P_0, the lower right domain P_1, the upper left domain

P_2, and the upper right domain P_3. The blocks are shown apart and at an angle so that the global indices, discussed below, can be shown for the entire contact surface.

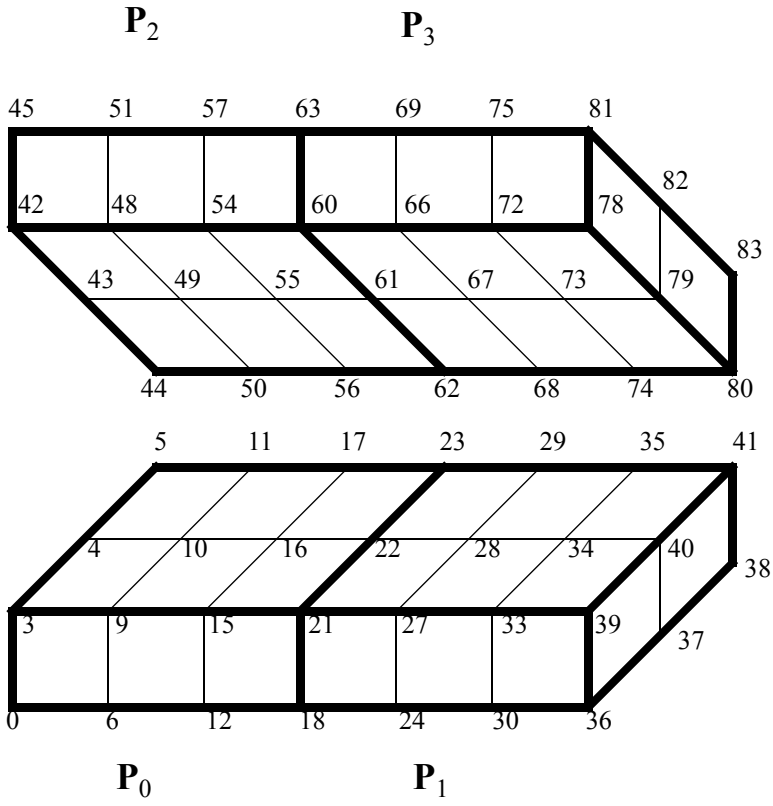


FIGURE A-5

This figure will be referred to in the following discussions of the various index spaces.

A.3.3 Global Index Space

When the grid is initially generated, a unique index is assigned to each node in the problem. If there are N nodes in the overall problem, then the global index space for the nodes is [0...N-1]. For large problems, the memories of even the largest sequential computers may be overwhelmed by the arrays in this index space. When this is the case, the problem may be generated in pieces, or may be initially decomposed into bulk domains, so that no

actual arrays in the global index space are allocated. Even in this case a mapping will exist that assigns to each node in the piece or in the bulk domain its global index.

For convenience, the tag ***Gidx*** is associated with the global index space, as in the Gidx index of a node. In Figure A-5 the Gidx indices of the global index space are shown.

A.3.4 Contact Surface Index Space

Each node on a given side of the contact surface has associated with it a contact surface index, or ***Sidx*** index. No two nodes on the same side of the surface will have the same Sidx index, but unrelated nodes on opposing sides may share the same index (but in a different index space).

Likewise each face on the surface is assigned an index, also referred to as its Sidx index although node and face Sidx index spaces are independent.

In order to uniquely identify a node based on its Sidx index, there exists a ***global map*** from Sidx index to corresponding Gidx index for each side of the surface. Figure A-6 shows the master and slave global maps for the example of Figure A-5.

Sidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Gidx	3	4	5	9	10	11	15	16	17	21	22	23	27	28	29	33	34	35	39	40	41

Master Sidx Global Map

Sidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Gidx	42	43	44	48	49	50	54	55	56	60	61	62	66	67	68	72	73	74	78	79	80

Slave Sidx Global Map

FIGURE A-6

Initially the user specifies which surfaces of the problem are master and slave contact surfaces during input to the grid generator. The grid generator, after creating the mesh and assigning an order to the nodes, determines which of these nodes lie on the specified surfaces, and outputs a list of the Global indices of just these nodes, sorted in the order of the global index. This is how the global map is produced.

The grid generator also builds the connectivity (Definition A.10) of each side, using the Sidx indices of each of the nodes on a given face to specify that face.

A.3.5 Bulk Domain Index Space

After the grid is generated by the grid generator, it must be partitioned in order to be solved in parallel (the partition phase can, of course, be part of the grid generator). The bulk decomposition creates N_{BD} separate bulk domains, by assigning each zone, or finite element (Definition A.2), of the problem to exactly one domain. If a zone is assigned to a particular bulk domain, then its constituent nodes and faces are also included in that domain. But whereas zones are unique to a domain, nodes and faces along the boundary between domains are shared by the adjoining domains, in the sense that each of the adjoining domains has its own copy. For example, in Figure A-5 nodes with Gidx 18-23 and the faces they bound are in both P_0 and P_1.

Each class of mesh entities (node, face, zone) in a bulk domain is ordered sequentially within its class in the domain, and assigned an index (its ***Bidx***, or bulk domain index) based on its location in the order.

A.3.5.1 Bulk Domain Global Node Map

Since each mesh entity in a bulk domain is also a mesh entity in the overall problem, it possesses a global index as well as a bulk domain index. The same node may exist in more than one bulk domain, with a different index in each. In order to determine when two Bidx indices refer to the same node (necessary, for instance, when computing total force on a node that lies on a bulk domain boundary) a bulk domain global map is created, similar to the contact surface global maps in Section A.3.4. Figure A-7 shows the bulk domain global node map for the bulk domain P_1 in the example of Figure A-5.

Gidx	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
Bidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Bulk Domain Global Node Map for P_1

FIGURE A-7

The bulk domain index space is the home space for most of the calculations in the overall dynamic code (the contact calculations are an exception). The various properties of the nodes and zones, such as coordinates, densities, volumes, etc., are stored in the bulk domain. Forces, accelerations, time advancement of non-contact nodes are all calculated in the bulk domain. For the most part, each bulk domain is considered to be an independent problem, with its connection to the overall problem treated by the domain as a sort of dynamic boundary condition.

A.3.5.2 Scalability of the Bulk Domain

One of the chief advantages of this domain-centric model is that the domain index space is much smaller than the global index space, and further, it is memory scalable (see

Section 2.2). The bulk domain index space, and thus the total memory requirements of a bulk domain, may be made as small as necessary simply by dividing the problem into more, smaller domains.

The drawback is that, since the same "names" (indices) are used over again by each domain, each domain must refer its local indices to a global index space to determine when two entities in different domains are actually the same entity.

Since most of the work is done in bulk domains, the bulk connectivity (Definition A.7, Definition A.9) is expressed in that index space. Each bulk domain includes only the connectivity for zones and external faces (see Section A.3.5.3 below) local to that domain. For each zone, the connectivity lists the constituent nodes, using the bulk domain index of the nodes in the list. For the example of Figure A-5, in bulk domain P_1 the zone formed by nodes with Gidx {18, 19, 21, 22, 24, 25, 27, 28} will be stored in the connectivity list as the list {0, 1, 4, 3, 6, 7, 10, 9}.

A.3.5.3 External Face List

An ***external face*** is a face in the bulk domain that only has a zone on one side. There are three reasons why a face may be external:

1. it lies on a true problem boundary
2. It lies on an interface between bulk domains
3. It lies on a contact surface

Examples of each of these types occur in Figure A-5 (the nodes will be written in Gidx for clarity. In the external face list they are actually stored in Bidx indices):

1. true boundary: {36, 37, 40, 39}
2. bulk domain interface: {18, 19, 22, 21}
3. contact surface {27, 33, 34, 28}

The external facelist is simply a collection of the nodelists of each external face (in Bidx indices).

A.3.6 Primary Domain Index Space

Each bulk domain has two daughter domains, a master and a slave ***primary domain***, which consist of the restriction of the bulk domain to the corresponding contact surface side. One or both of these derivative domains may, and often will, be null. For example, in Figure A-5 bulk domain P_1 has an associated master primary domain that consists of the (Gidx) nodes {21, 22, 23, 27, 28, 29, 33, 34, 35, 39, 40, 41}, and an associated slave primary domain that consists of $\varnothing$. The index of a node in the list of nodes in a primary domain, when that list is sorted by increasing Gidx, is the primary domain index, or ***Pidx*** index, of the node. For instance, in the above example the node in P_1 with a Gidx of 23 is assigned a Pidx of 2.

The primary domain index space for a given bulk domain and contact side is extended by adding to the end of the list of ***local*** primary domain nodes the set of ***proxy*** nodes (Definition A.20); i.e., neighbors of the local nodes that are not themselves local. The list of proxies is again sorted by increasing Gidx, but is kept separate from the local nodes, and follows them in the index space. Referring once more to Figure A-5, the nodes in the master primary domain of bulk domain P_1, augmented with proxy nodes and given in Gidx for convenience (assuming neighborhood radius 2), are: {21, 22, 23, 27, 28, 29, 33,

34, 35, 39, 40, 41, 9, 10, 11, 15, 16, 17}, which are assigned the corresponding Pidx indices 0 - 17.

No global domain map is required by the algorithm for primary domains. Instead there are two other maps defined: from primary domain index to bulk domain index and from primary domain index to contact surface index, labeled respectively the ***Pidx→Bidx*** map, and the ***Pidx→Sidx*** map. For the purposes of these maps, the proxy nodes are not included. Figure A-8 lists these maps for the case of the master primary domain of bulk domain P_1 from Figure A-5. For reference the corresponding Global index is given as

Gidx	21	22	23	27	28	29	33	34	35	39	40	41
Bidx	3	4	5	9	10	11	15	16	17	21	22	23
Pidx	0	1	2	3	4	5	6	7	8	9	10	11

Pidx->Bidx Map

Gidx	21	22	23	27	28	29	33	34	35	39	40	41
Sidx	9	10	11	12	13	14	15	16	17	18	19	20
Pidx	0	1	2	3	4	5	6	7	8	9	10	11

Pidx->Sidx Map

well.

FIGURE A-8

A.3.7 Contact Domain Index Space

The contact domains are in general unrelated to the primary or bulk domains. They exist to bring opposing patches of the contact surface together on the same processors in order to calculate their mutual effects on each other. In Figure A-5 a sample bulk domain decomposition into P_0 - P_3 is shown by the bold lines. Note that each bulk domain is only

involved in one side of the contact surface. For example, P_1 cannot calculate the force exerted by the slave side, unless domains containing the slave side send that information. Although that is one approach, a more general approach is taken by this algorithm, and all data is assumed to be redistributed across the machine for the contact decomposition.

Nodes in a contact domain belong to one of two categories; they are either assigned (Definition A.22) or ghost (Definition A.23). After all the nodes from various primary domains have been received, the contact domain orders each side by first sorting all the assigned nodes by increasing Sidx, then following that by a sorted list of all the ghost nodes. A node's index in this combined list of assigned and ghost nodes is its ***contact domain index***, or ***Cidx*** index.

To give a concrete illustration, suppose that for the problem of Figure A-5, the sets {33, 39, 40, 41} (master, Gidx index) and {72, 78, 79, 80} (slave, Gidx index) are assigned to contact domain C_3. Assuming a neighborhood radius of 2 (Definition A.13), C_3 will also receive the ghost master nodes {21, 22, 23, 27, 28, 29, 34, 35} (Gidx index), and ghost

slave nodes {60, 61, 62, 66, 67, 68, 73, 74} (Gidx index). Figure A-9 shows the resulting index mappings, including the Gidx values for reference.

Gidx	33	39	40	41	21	22	23	27	28	29	34	35
Sidx	15	18	19	20	9	10	11	12	13	14	16	17
Cidx	0	1	2	3	4	5	6	7	8	9	10	11

Cidx->Sidx Master Map

Gidx	72	78	79	80	60	61	62	66	67	68	73	74
Sidx	15	18	19	20	9	10	11	12	13	14	16	17
Cidx	0	1	2	3	4	5	6	7	8	9	10	11

Cidx->Sidx Slave Map

FIGURE A-9

In addition to the Cidx->Sidx maps, the contact domain also uses an inverse map, from Sidx to Cidx. This is shown in Figure A-10 for the preceding example, for the master side.

Cidx	-	-	-	-	-	-	-	-	-	4	5	6	7	8	9	0	10	11	1	2	3
Sidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sidx->Cidx Map

FIGURE A-10

Unlike all the preceding index spaces, the contact domain index space changes each cycle, and is reconstructed by the contact domain after it receives its input from all contributing primary domains. Like the bulk domain index space, the contact domain indices are "working" indices, used directly to access nodal data such as coordinates, etc., for calculations. This implies that the contact domain has its own copy of this data.

A.3.8 Relative Order of Index Spaces

The global index for each node is assigned contiguously during mesh generation. Typically, indices are assigned to the three-dimensional body(s) by sweeping through the nodes in one dimension first and incrementing the global index for each successive node in a row. Then successive rows in the second dimension are assigned indices in the same fashion, and finally successive planes of rows. Accommodations must be made for more complex geometries such as general unstructured meshes, but this gives an example of the simplest case. In any case, simple or complex, the order in which the nodes are assigned global indices can be considered arbitrary, since from that point on the mesh is treated as unstructured, and does not assume any implicit order.

The mesh generator is also responsible for identifying the contact surface, guided by user input, and for assigning the master and slave contact surface indices (Sidx indices). It does this by creating a list of the global indices of all the nodes on a given side of a surface, and then sorting this list by increasing Gidx index. The offset of a given contact node in this sorted list is, by definition, the contact surface index of the node. Thus the contact surface index space is in the same order as the global index space: the Sidx->Gidx map will always be a monotonically increasing function.

This fact is relied on and used repeatedly in the code. For instance, when it is known that two lists consist of monotonically increasing indices, it is much cheaper to find their intersection. It also makes it possible to perform binary searches on long lists.

Likewise, the bulk domain index space is ordered in the same order as the global index space, so that the Bidx->Gidx map is monotonically increasing. This makes it simpler for

adjacent bulk domains to determine their common nodes (again, due to cheaper intersection).

The primary domain index space is a little more complicated, since it consists of a set of local nodes, ordered by global index, followed by a set of proxy nodes, again ordered by global index. Since any or all of the proxy nodes may have a lower global index than some local nodes, the Pidx index space as a whole is no longer ordered by global index. In cases where this is a problem the two halves of the Pidx index space are treated separately, since each half is well-ordered.

Similarly, the contact domain index space consists of two parts, the assigned nodes and the ghost nodes. The assigned nodes are ordered by global index, followed by the ghost nodes, which can again lead to an order inversion across the two halves. See Figure A-9 for an example of this. The solution is the same: the two halves are each well-ordered and can be processed in sequence when necessary.

A.4 Primary Decomposition

The primary decomposition is a distribution of contact nodes and faces into some number of separate domains. It is completely dictated by the bulk decomposition, since each primary domain is simply a restriction of a parent bulk domain to nodes and faces on the master or slave side of the contact surface. From the perspective of the bulk decomposition, contact surfaces exist only as problem boundaries. In general there is no consideration given to optimizing the distribution of contact nodes and faces in partitioning the mesh.

Although a primary domain is a subset of a bulk domain, it possesses properties and undergoes operations that do not occur in the overall bulk domain, and so it is maintained as a separate, though associated, entity.

A.4.1 Structure of the Primary Domain

Much of the information related to nodes and faces of a primary domain is already stored in the parent bulk domain. For instance, since contact nodes are also bulk nodes, their current coordinates, velocities, etc., are kept in the bulk domain. Since a primary domain and its parent bulk domain always reside in the same processor (and the same local address space), it would be redundant to keep two copies of this data. Instead, the primary domain only needs to know the Bidx (Section A.3.5) of a given node to look up these properties. It is the purpose of the Pidx->Bidx map (Section A.3.6) to provide the Bidx of a given primary domain node.

On the other hand, there are a number of properties which are specific to contact nodes. It would be wasteful to keep this data in the bulk domain index space, since it would only have meaning for a small subset of indices. Instead these properties are saved in the primary domain index space, in ***nodeStuff*** blocks.

An array of these blocks is allocated for each primary domain. Each contact node, whether local (Definition A.19) to the primary domain or proxy (Definition A.20) to it, has its own nodeStuff block. The arrays of nodeStuff blocks are in the primary domain index space, so the nodeStuff block for a given node is at the nodes's Pidx index in the nodeStuff array.

There is no corresponding requirement for "faceStuff" blocks, since very little data is associated with the face, and what data there is (e.g. stresses in the underlying zones) exists already in the bulk domain, without need for replication in the primary domain.

Figure A-11 shows a typical nodeStuff array and an expansion of an arbitrary nodeStuff block. The following subsections provide more detailed information about the members of a nodeStuff block.

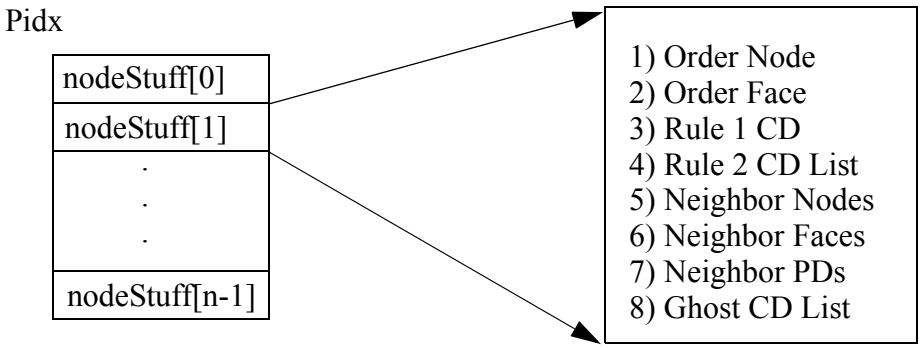


FIGURE A-11

A.4.1.1 Order Node

As defined in Section A.2.3, The order node is the closest node on the other side of the surface to the specified node. The Sidx index of the order node is recorded in the nodeStuff block.

A.4.1.2 Order Face

The order face is the face on the other side of the surface in contact with, or directly across from, the specified node.

A.4.1.3 Rule 1 CD

The Rule 1 CD of a node is simply the contact domain, or CD, to which the specified node is assigned. Each node is assigned to exactly one CD. For master nodes, this assignment is static. For slave nodes, the Rule 1 CD changes dynamically as the surface changes, and is defined to be the Rule 1 CD of the slave node's current order node (a master node). If a node has a specified Rule 1 CD, then it and all its node neighbors must be sent to that CD for the contact calculations. Chapter 3 discusses this requirement as well as the motivation for several of the following nodeStuff elements, in a general setting.

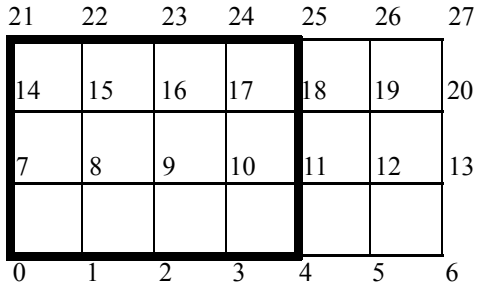
A.4.1.4 Rule 2 CD List

For a given node, the Rule 2 CD List is a list of the CDs that have one or more assigned nodes whose order node is the given node. Again, Chapter 3 motivates this requirement. If a particular CD is in the Rule 2 CD List of a given node, then that node and all its node neighbors must be sent to that CD for the contact calculations. How Rule 1 and Rule 2 information is applied will be discussed in Section A.9.1; for now it is just being pointed out that the nodeStuff block is where these items are kept.

A.4.1.5 Neighbor Nodes

For a given local or proxy node, this is a list of all the local nodes within the given node's neighborhood. In Definition A.13 the notation $[n_i]$ was introduced to represent the node neighborhood of n_i, without respect to any domain decomposition. The radius is implicit. This notation may be extended to $[n_i]_P$, to represent the intersection of $[n_i]$ with the set of nodes local to Primary Domain P: $[n_i]_P = [n_i] \cap N_P$. For a given local or proxy node n_i, the

set $[n_i]_P$ is kept in the node's nodeStuff block. Figure A-12 shows two examples of the neighbor node list; for a local node and for a proxy node. The Primary Domain P is defined by the bold outline.



$[n_{10}]'_P = \{1, 2, 3, 4, 8, 9, 10, 11, 15, 16, 17, 18, 22, 23, 24, 25\}$

$[n_{12}]'_P = \{3, 4, 10, 11, 17, 18, 24, 25\}$

FIGURE A-12

In the above example, for clarity the "primed" node neighborhoods have been listed by the Sidx indices of the nodes. In the nodeStuff block, the node neighborhoods are kept as lists

of corresponding Primary Domain indices. In Figure A-13 the conversion for the two listed neighborhoods is made from Sidx index space to Pidx index space.

Sidx	0	1	2	3	4	7	8	9	10	11	14	15	16	17	18	21	22	23	24	25
Pidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Local Node Pidx->Sidx Map

Sidx	5	6	12	13	19	20	26	27
Pidx	20	21	22	23	24	25	26	27

Proxy Node Pidx->Sidx Map

$[n_8]_P$ = {1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19}

$[n_{22}]_P$ = {3, 4, 8, 9, 13, 14, 18, 19}

FIGURE A-13

A.4.1.6 Neighbor Faces

Similarly to neighbor nodes in the previous section, neighbor faces is a list of the faces in the neighborhood $[n_i]$ of node n_i, that are also in the primary domain: i.e. in $[n_i]_P$. The faces are listed in the primary domain index space for faces of the given contact side.

A.4.1.7 Neighbor PD List

Each local node in a primary domain keeps a list of other primary domains that are within the neighborhood radius of the given node. In other words, if a given node n_i, local to primary domain P, is also a proxy node to primary domain Q, then Q is in the neighbor PD list in n_i's nodeStuff block. This information is sent to the contact domain and used to determine which PD's the contact domain will expect data from next cycle. This will be covered in greater detail in Section A.12.3.

A.4.1.8 Ghost CD List

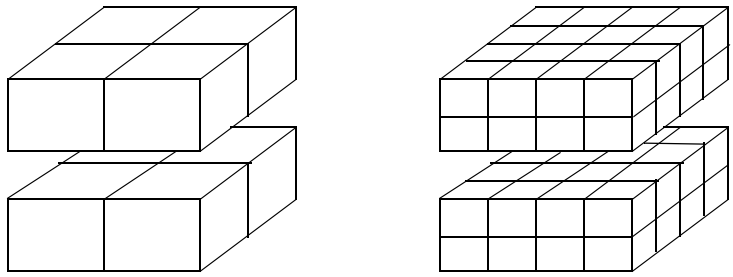
Each local node in a primary domain keeps a list of all the contact domains that the node is sent to as a ghost in the current cycle. This information is sent to the contact domain where the node is assigned, so that the contact domain can determine which other CD's it must communicate with to update ghost intermediate values during the contact calculations. This will be covered in greater detail in Section A.12.2.

A.4.2 Load Balance and Scalability

In general a contact surface will not be distributed evenly across the bulk decomposition, since it is a two-dimensional object embedded in a three-dimensional mesh. For instance, in Figure A-14 (a) there are eight bulk domains. Each bulk domain has about the same fraction of the contact surface (the surface lying between the lower and upper blocks). In Figure A-14 (b) the mesh has been refined by doubling the number of zones and the number of bulk domains in each dimension (the figure shows only bulk domains, not individual zones). Because both zones and bulk domains are doubled in each dimension, the number of zones per bulk domain, and thus the amount of work per bulk domain, remains constant.

The result is that the overall mesh size has been increased eight-fold. However, the number of contact faces has only increased four-fold. Also, the surface in Figure A-14 (b) is only distributed across half of the bulk domains. If each bulk domain is assigned to a pro-

cessor (the typical case) then load balance for the surface is very poor; half the processors will sit idle during operations in the primary domain.



(a) (b)

FIGURE A-14

As another example, if a problem is partitioned into 1000 bulk domains, one can expect O(100) non-null primary domains. Of course, in specific problems with actual contact surfaces, mileage will vary. But as a rule of thumb, for N processors or bulk domains and a single contact surface, only about $N^{2/3}$ will have non-null primary domains.

This is not as bad as it first appears. Consider what happens in going from Figure A-14 (a) to Figure A-14 (b). Assuming each bulk domain is mapped to a processor, so that eight processors are assigned in (a) while 64 are assigned in (b), then the same amount of work per processor is done in (a) as in (b). But furthermore, the same amount of work is done in each primary domain in (a) as in (b), for those primary domains that still have work. So even though load balance gets worse, this is compensated by the slower growth in size of the contact surface than the overall problem. Just as for the bulk domains, the work per primary domain executes (theoretically) in constant time as problem and machine are scaled proportionately, even though more and more processors are idled.

A.5 Contact Decomposition

The contact decomposition is a partitioning of the overall contact surface, such that each contact node is ***assigned*** to exactly one of M contact domains. A contact domain will include more than just its assigned nodes, but these additional requirements are derived from the assigned nodes.

In general the contact nodes could be assigned to contact domains in any arbitrary fashion. A node could for instance be assigned to the contact domain with the same order rank (ID) as the lowest ranked primary domain to which it is local. Alternatively all the master nodes could be assigned to different contact domains than all the slave nodes. In either of these cases the contact decomposition would be static with respect to the assigned nodes. (The "additional requirements" referred to above would still change dynamically).

However, the overall goal of the contact decomposition is to bring together opposing nodes and faces on the two sides of the surface into relatively autonomous contact domains, where the force balance and impenetrability calculations of contact enforcement can be performed for the assigned nodes of the domain. In doing so, two subsidiary goals are:

a) to create a load-balanced distribution of the contact work, and
b) to minimize communication between primary and contact domains

The above-mentioned schemes would do a poor job of satisfying these goals.

A.5.1 Locale of an Assigned Node

When a node is assigned to a contact domain, it is for the purpose of updating its state: that is, determining a modified acceleration that accounts for pressure from the other side, and

from this a corrected velocity and position for the node. To accomplish this it needs not only its own initial state but information about its local environment, or locale. For the contact enforcement scheme used in this model, the locale consists of:

1. nodes and faces in the assigned node's neighborhood,
2. the ordernode of the assigned node,
3. nodes and faces in the ordernode's neighborhood.

As an example, suppose that master node 6 in Figure A-15 has been assigned to contact domain CD_1, and that its ordernode this cycle is slave node 12. Further, let the neighborhood radius be one.

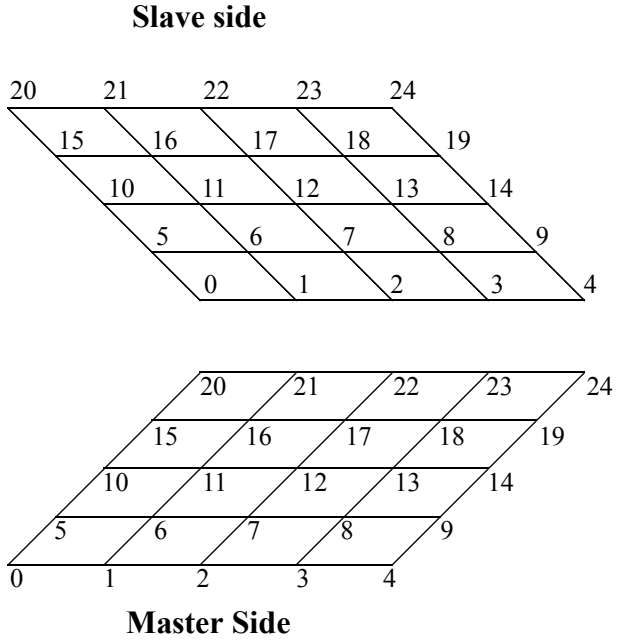


FIGURE A-15

Then in addition to master node 6 (m_6) the following nodes (and associated faces) must also be present in contact domain 1:

$[m_6] = \{m_0, m_1, m_2, m_5, m_6, m_7, m_{10}, m_{11}, m_{12}\}$

$[s_{12}] = \{s_6, s_7, s_8, s_{11}, s_{12}, s_{13}, s_{16}, s_{17}, s_{18}\}$

Any or all of these additional nodes may or may not be assigned to CD_1, but in any case their full state must be available on CD_1. Those nodes in the above list not assigned to CD_1 are referred to as ghost nodes: their state will be used, but not modified (by CD_1).

A.5.2 A "Good" Contact Decomposition

In the above example, a single assigned node generated the requirement for an additional 17 nodes (it would be much worse for a more practical neighborhood radius of two). Since they must be there regardless, it would be most efficient if as many as possible are assigned to CD_1. This has two implications.

First, a good decomposition will choose dense compact subsets of the surface to assign to a single contact domain, in terms of the metric imposed by the connectivity, in order to maximize the number of neighbors of an included node that are themselves included. This is in essence a boundary-to-surface ratio issue, where minimization of the ratio leads to the minimal ghost requirements.

Second, a good decomposition will attempt to assign node-ordernode pairs to the same contact domain. Again, this is because if the ordernode must be present, it is most efficient that this be the CD where it is assigned.

These two considerations are not independent: as illustrated in Figure A-16, a tight group of nodes on one contact side maps (via the order relations) to a tight group of ordernodes on the other side.

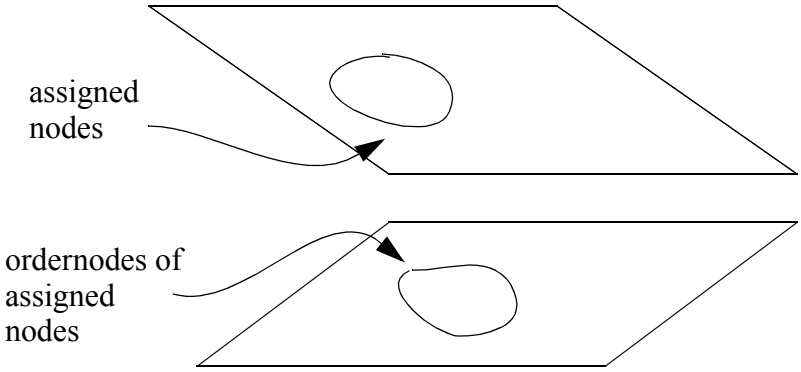


FIGURE A-16

A.5.3 Master Static / Slave Dynamic Contact Decomposition

The approach taken in this model is to statically assign the master nodes of the contact surface to the various contact domains, choosing the sets to minimize the boundary / area ratio. The static aspect of the contact decomposition is considered in greater detail in Section A.6.8.

Once the master side assignments have been determined statically, the slave assignments are determined dynamically each cycle, by using the current ordernode relations for each slave node to determine what its current master ordernode is, and then assigning the slave to the same contact domain as its master node.

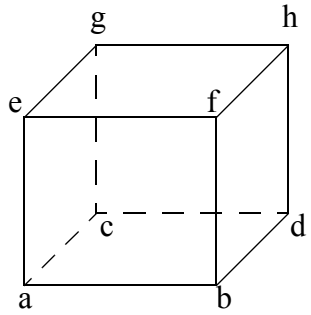
A.6 Initialization

A.6.1 Determining Local Nodes and Faces

As a result of the mesh generation and partitioning preprocessing steps, the algorithm is provided at startup with the following:

1.) the zonelist of all elements in each bulk domain.
2.) the facelist of all faces on the contact surface.
3.) the Bidx->Gidx and Sidx->Gidx maps.

The zonelist has one entry for each zone, or element, in the bulk domain. This entry, in turn, consists of an ordered list of the eight nodes at the corners of the hexahedral zone. The nodes are specified by their Bidx index. Figure A-17 shows a typical zone and its zonelist entry. It also shows how the faces of the zone can be extracted from the zonelist by six specific choices of four nodes at a time.



Faces

{a, b, d, c}
{e, f, h, g}
{a, c, g, e}
{b, d, h, f}
{a, b, f, e}
{c, d, h, g}

Zonelist entry: {a, b, d, c, e, f, h, g}

FIGURE A-17

Item 2, the facelist of contact surface faces, has one entry for each face on the surface. Each entry is in turn a list of the four nodes, represented by their Sidx indices, of the face. Unlike the zonelist, this list is not specific to a particular bulk domain; it includes the entire surface.

The maps of item 3 were discussed in Section A.3.5.1 and Section A.3.4, respectively.

The first step in determining the local nodes of a primary domain is to create a list of all faces in the bulk domain from the zonelist. This is done by looping through the zones of the zonelist and extracting the six faces of the zone. For each face, a list of the zones having that face is kept. A given face will have either one or two including zones. From the full facelist, the subset of faces that have only one including zone is extracted. This is the external facelist (Section A.3.5.3), and is a much smaller set than the full facelist, which can now be discarded.

Next, the Bidx->Gidx map is applied to the nodes of a copy of the external facelist, and the Sidx->Gidx map is applied to the nodes of a copy of the contact surface facelist, so that the two facelist copies are now both in Gidx index space. The two lists can now be searched for faces in common: i.e. for contact faces local to the primary domain.

In this search each positive and negative rotation of the indices of a face must be considered. For example, a face with Gidx indices {a, b, c, d} is the same face as {b, c, d, a}, {c, d, a, b}, {d, a, b, c}, {d, c, b, a}, {c, b, a, d}, {b, a, d, c}, and {a, d, c, b}. The fastest way to handle this is to sort each face by increasing Gidx index, and then just compare sorted faces. This seems to allow non-matches, such as {a, b, c, d} and {a, c, b, d}, but in practice at most one of these is a valid face and will occur in the facelists.

When a match is found, that face can be added to the list of local contact faces. This is done by saving the indices in both facelists where the match occurred. These indices can also be used to refer back to the original, unglobalized facelists, from which the Bidx and Sidx indices of the nodes on that face can be extracted and added to two lists of local con-

tact nodes, one in Bidx index space and the other in Sidx index space. When the lists are sorted, a given offset (index) in either list refers to the same node. This is true because both Bidx and Sidx index space are created in the same order as Gidx index space. That is, both the Bidx->Gidx and Sidx->Gidx maps are monotonically increasing functions. The offset into either of the sorted lists is the Pidx index of the node.

To illustrate this phase of the initialization, consider the following example. In Figure A-18 part of a problem is shown. The block is divided into two bulk domains (P_0 on the left, P_1 on the right), and the upper surface is designated the master side of a contact surface. The block is drawn four times so that significant nodes can be shown with their Gidx, Bidx, Sidx, and Pidx indices.

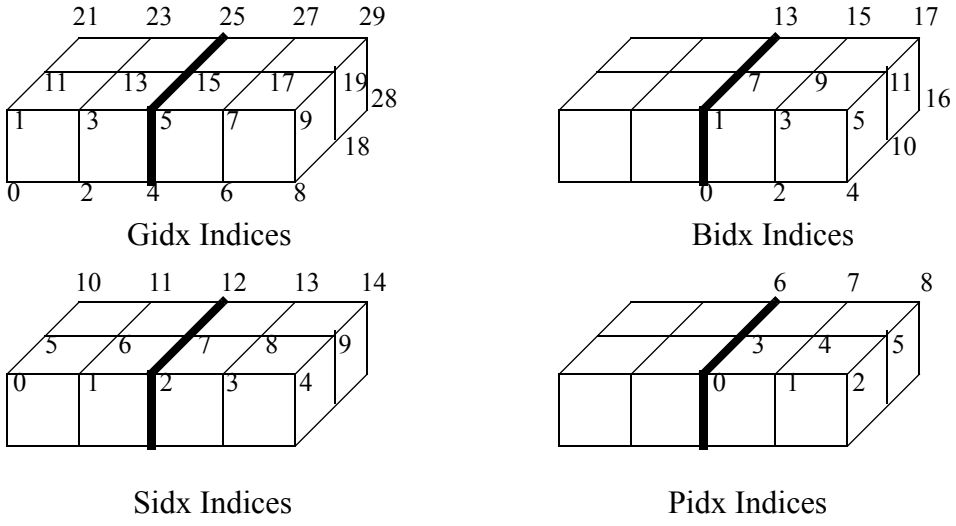


FIGURE A-18

In Figure A-19 the external facelist for P_1 and the master side contact facelist are shown, in Bidx and Sidx indices respectively.

Face idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Node 0	0	2	12	14	0	6	4	10	0	2	6	8	1	3	7	9
Node 1	2	4	14	16	6	12	10	16	2	4	8	10	3	5	9	11
Node 2	3	5	15	17	7	13	11	17	8	10	14	16	9	11	15	17
Node 3	1	3	13	15	1	7	5	11	6	8	12	14	7	9	13	15

External FaceList for P_1

Face idx	0	1	2	3	4	5	6	7
Node 0	0	1	2	3	5	6	7	8
Node 1	1	2	3	4	6	7	8	9
Node 2	6	7	8	9	11	12	13	14
Node 3	5	6	7	8	10	11	12	13

Master Contact Facelist

FIGURE A-19

In Figure A-20 these facelists have been globalized, and the set of nodes for each face sorted.

Face idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Node 0	4	6	24	26	4	14	8	18	4	6	14	16	5	7	15	17
Node 1	5	7	25	27	5	15	9	19	6	8	16	18	7	9	17	19
Node 2	6	8	26	28	14	24	18	28	14	16	24	26	15	17	25	27
Node 3	7	9	27	29	15	25	19	29	16	18	26	28	17	19	27	29

External FaceList for P_1

Face idx	0	1	2	3	4	5	6	7
Node 0	1	3	5	7	11	13	15	17
Node 1	3	5	7	9	13	15	17	19
Node 2	11	13	15	17	21	23	25	27
Node 3	13	15	17	19	23	25	27	29

Master Contact Facelist

FIGURE A-20

From the globalized facelists the following matches are found (first index is the external facelist index, second index is the master contact facelist index).

(12, 2), (13, 3), (14, 6), (15, 7)

These are the contact faces that are local to the primary domain. They are kept as two parallel lists: {12, 13, 14, 15} and {2, 3, 6, 7}. The index into either of these lists of a particular face is that face's Pidx index (separate namespace from the node Pidx index space).

Next, the nodes belonging to the faces in the two lists (which are the local nodes of the primary domain) are extracted into two separate node lists and sorted. In the first case the nodes are represented in Bidx indices, in the second case in Sidx indices. The resultant lists provide the Pidx->Bidx map and Pidx->Sidx map (Section A.3.6), as shown in Figure A-21 for the above example.

Bidx	1	3	5	7	9	11	13	15	17
Pidx	0	1	2	3	4	5	6	7	8

Pidx->Bidx Map

Sidx	2	3	4	7	8	9	12	13	14
Pidx	0	1	2	3	4	5	6	7	8

Pidx->Sidx Map

FIGURE A-21

A.6.2 NodeStuff I: Local Neighbor Nodelists, Facelists

For each primary domain P and each side of the surface an array of nodeStuff blocks, as defined in Section A.4.1, is allocated. The i^{th} nodeStuff block will hold data for the node with Pidx i. For each local node n_i in the primary domain, a list of its neighbor nodes $[n_i]$ is computed and stored in the node's nodeStuff structure, and similarly for the local neighbor faces. These lists are discussed more fully in Section A.4.1.5 and Section A.4.1.6. At this point the neighbor node list is not yet constrained to nodes local to the primary

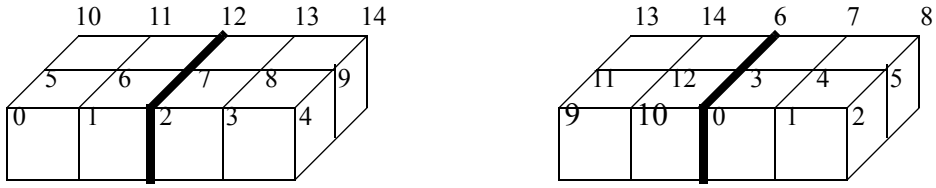
domain, since the full neighbor lists are needed to determine proxy nodes, described in the next section.

A.6.3 Determining Proxy Nodes

As defined in Definition A.20, a proxy node of a primary domain P is a contact node that is in the neighborhood $[n_i]$ of a local node n_i, but is not itself a local node. Proxy nodes would typically be called ghost nodes in the literature, but to avoid confusion between primary domains and contact domains the term ghost node is reserved for contact domains, while proxy nodes are associated exclusively with primary domains.

After a primary domain has determined its set of local nodes N as discussed in the previous section, it then uses its nodeStuff neighbor node lists (Section A.6.2) to find the node neighborhood $[n_i]$ (Definition A.13) of each node $n_i \in N$. Then it collects all nodes in all such $[n_i]$ into a single list (in Sidx index space) and removes from the list all the local nodes. What remains is the set of proxy nodes. This set is then sorted by increasing Sidx and assigned Pidx indices, where the Pidx index of the first proxy node is one greater than the highest local node Pidx index, and so forth. At the same time, in order to retain a pointer to the proxy node's true identity, the proxy node's Sidx is added to the Pidx->Sidx map at its Pidx index.

As an example, consider the problem introduced in the previous section. In this case the proxy nodes have been added to the Pidx index space, and the resulting Pidx->Sidx map is shown. This assumes a neighborhood radius of two.



Sidx Indices Pidx Indices

Sidx	2	3	4	7	8	9	12	13	14	0	1	5	6	10	11
Pidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Pidx->Sidx Map

FIGURE A-22

A.6.4 NodeStuff II: Proxy Node Neighbor Nodelists, Facelists

Once the proxy nodes for a primary domain have been calculated, the Pidx namespace and corresponding nodeStuff array is extended to include the proxy nodes, and for each proxy node, its local neighbor nodes and faces are stored in the proxy node's nodeStuff. Also, the neighbor lists $[n_i]$ of local nodes n_i are trimmed of any non-local (i.e. proxy) nodes. That is, $[n_i]$ is further reduced to $[n_i]_P$ in the neighbor nodelists of both local and proxy nodes.

A.6.5 Initializing Proxy Communication

A proxy node in one primary domain is a local node in one or more other primary domains, where its full state is kept current. The point of having proxy nodes is to have a convenient place to store a copy of a small subset of the state of those nodes that are close

enough to local nodes to influence them (in particular to influence which contact domains particular local nodes must be sent to).

In order to update the information stored by the proxies (a subset of the nodeStuff block discussed in Section A.4.1) the primary domain must determine who has the information (i.e. in which primary domain(s) the proxy is local). Conversely, any primary domain where the node is local must determine to which primary domain(s) the node is proxy.

Since primary domains are themselves static, the information about where a given node is local and where it is proxy is unchanging, and thus the communication channels can be determined during initialization.

To do this, once a primary domain P has determined its proxy nodes (Section A.6.3), it sends a list of them (in Sidx indices) to all other primary domains. Since the Sidx index is independent of primary domain, it can be used to compare nodes in different primary domains for identity.

Each other primary domain, when it receives the proxy list from P, compares it to its list of local nodes, converted to Sidx indices using its Pidx->Sidx map (Section A.3.6), and looks for matches. It builds a list of the set of such matches and sends the list back to P in Sidx indices, as well as retaining a version of the list converted to local Pidx indices. These lists of nodes to be sent from one domain to another will be referred to as ***invoices***. They appear not only in PD↔PD communication, but also in PD↔CD and CD↔CD communication, to be discussed in later sections.

When P receives the invoice from another primary domain it converts it from Sidx to its own Pidx indices. In this case the Pidx indices will be indices of proxy nodes, beyond the end of the local nodes in the Pidx index space.

At this point the PD↔PD communication pattern is established. Each primary domain has, in its own Pidx index space, an invoice list of what local nodes to send to surrounding primary domains, and what proxy nodes to receive from surrounding primary domains. Wherever the invoice is null, no further communication with that PD is required. This has a very important consequence for scalability of the overall algorithm: all subsequent PD↔PD communication is limited to nearest neighbors. As discussed in Section 2.2, this is necessary for scalability.

To help focus the preceding discussion, the following example is provided. In Figure A-23 one side of a slide surface is shown partitioned into nine primary domains.

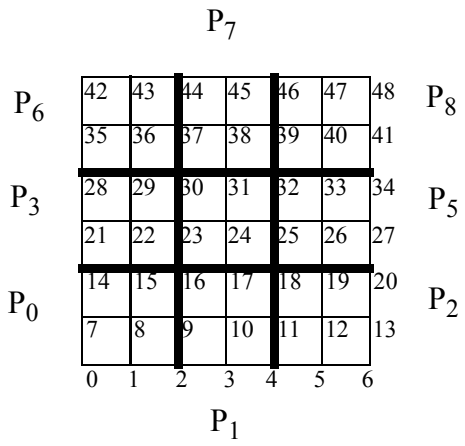


FIGURE A-23

Focusing on primary domain P_2, in Sidx P_2 has local nodes {4, 5, 6, 11, 12, 13, 18, 19, 20} and proxy nodes {3, 10, 17, 24, 25, 26, 27} (assuming a neighborhood radius of one). P_2 sends its proxy list to all other primary domains, and gets back the following invoices:

P_0 : ∅

P_1 : {3, 10, 17}

P_3 : ∅

P_4 : {17, 24, 25}

P_5 : {25, 26, 27}

P_6 - P_8 : ∅

There will be no further communication by P_2 with P_0, P_3, or P_6 - P_8. P_2 will convert the invoices from P_1, P_4, and P_5 into its own Pidx space:

P_1 : {9, 10, 11}

P_4 : {11, 12, 13}

P_5 : {13, 14, 15}

Meanwhile, P_2 also remembers what invoices it has sent out to other primary domains, in order to fill their proxy requirements:

P_1 : {5, 12, 19}

P_4 : {11, 12, 19}

P_5 : {11, 12, 13}

P_0, P_3, P_6 - P_8 : ∅

which it converts to its local Pidx space:

P_1 : {1, 4, 7}

P_4 : {3, 4, 7}

P_5 : {3, 4, 5}

Thus primary domain P_2 knows what to expect from each other PD and where to store it when it arrives, and what to send to each other PD and where in its local index space to get

it from. The invoices are constant for the rest of the run, and constitute all PD↔PD communication.

A.6.6 NodeStuff III: Neighbor PD lists

For reasons that are explained in Section A.12.3, it is necessary to build a list for each node of any other primary domains in that node's neighborhood, or in other words, a list of other PDs on which the local node is proxy. It does this by first initializing for each node an empty neighbor PD list. Then the PD<->PD invoices (Section A.6.5) for each neighbor PD are scanned. For each local node found in a particular PD invoice, that PD is added to the neighbor PD list for that node. Once calculated, the list is stored in the node's nodeStuff structure.

A.6.7 Global Ordering

A critical assumption of the overall contact algorithm is that a node on one side of the surface only moves a small distance with respect to the other side of the surface in a single time step. Specifically, if a node n_i has an ordernode (Section A.2.3) n_j, on a given cycle, then on the next cycle n_i's ordernode must be in the node neighborhood $[n_j]^1$ (Definition A.13) of the old ordernode n_j, and its new orderface will be a face connected to the new ordernode.

This assumption allows us to do a local search each cycle for the next ordernode, based on the most recent ordernode. If the search cannot be limited in some such way, then scalability is impossible. However, this method still needs a starting point: the initial ordernode must be determined by other means.

The simplest method is a brute-force global search. Each node on a given side of the contact surface calculates the distance to every node on the other side of the surface. The closest one found is that node's ordernode. Assuming each side has O(N) nodes, this procedure takes $O(N^2)$ time on a single processor.

It is possible (at least on present-day problems and machines) to replicate all nodal coordinates onto M processors of a parallel machine, and to assign 1/M of the distance calculations to each processor. This reduces overall time for global ordering to $O(N^2/M)$, which is typically "fast enough" for an initialization phase that only gets executed once at the start.

However, in some recent large runs of several hundred million zones, it was observed that the global ordering could occupy several minutes of time, and so a further refinement has been implemented. The longest coordinate dimension of the problem (x, y, or z) is determined, and then the contact nodes on each side are sorted, using their coordinate in the selected dimension as the sort key.

The sorted list of contact nodes on one side (the ***active*** side) is divided into M equally-sized contiguous sets, and each such set is assigned to one processor. That processor determines the coordinate range in the sorted dimension of its active set, and from this selects an overlapping range from the passive side's sorted list. For each active node, the processor now only searches the overlapping range of the passive side. Following this, active and passive sides are reversed, and the operation repeated.

Using this approach on M processors, the cost of the search, per processor, is reduced to $O(N^2/M^2)$, with the additional cost of the sorts, O(NlogN).

When the ordernode and orderface for each node have been determined, they are written into global arrays in Sidx index space, using global OR operations, where each processor just fills in the nodes it is responsible for. The result is that all processors momentarily have full order information for the contact surface. From this, each primary domain can extract the ordernode and orderface for just its own local nodes, storing this information into the nodeStuff blocks (Section A.4.1) of the local nodes. At this point the global contact information arrays are freed.

A.6.8 Master Side Static Decomposition

As discussed in Section A.5.3, the nodes on the master side of the contact surface are assigned statically to the various contact domains. Every cycle the master nodes will be sent to the same CDs. The contact proximity mapping X (Definition 5) is then used to determine where the slave nodes are assigned during any particular cycle.

Three different static decompositions have been implemented. These are discussed in the following subsections. In each case the decomposition is actually done in terms of contact faces. This is the more natural approach to dividing up regions, and is inherited from the bulk decomposition, in which zones (not nodes) are assigned uniquely to the various bulk domains. After contact faces are partitioned an arbitration pass is made to convert from the face decomposition to a nodal decomposition. This is discussed in Section A.6.8.4.

Actually, a fourth method of decomposition is also available, whereby the user supplies a file which specifies for each master node which CD it is assigned to. This method is useful for debugging the code, but is not practical for real applications.

A.6.8.1 Decomposition by Contact Surface Index

This is the most naive decomposition, and is simply based on the Sidx (Section A.3.4) ordering of the master side. The list of all faces is sorted by Sidx index, and then divided into N equally-sized contiguous subsets, where N is the number of contact domains.

While this method is simple to understand and implement, it has some drawbacks. Contact domains tend to come out as one-dimensional strings of nodes on the two dimensional surface, which leads to a 4-to-1 ratio of ghosts to assigned nodes (for neighborhood radius of 2). While this is an improvement over isolated nodes (with a 24-to-1 ratio), it can be greatly improved upon.

A.6.8.2 METIS RSB Decomposition

This method of decomposition uses the METIS [30] software library to produce a decomposition based on the Recursive Spectral Bisection method. It is beyond the scope of this work to discuss RSB decomposition in great detail. But, roughly speaking, RSB operates on a graph wherein the contact faces are the "nodes" of the graph, and the graph edges connect any two "nodes" (contact faces) which are in contact (i.e. share a contact node). The goal of the RSB algorithm is to produce a partition of the graph which has roughly the same number of graph "nodes" in each part, and which cuts through the minimum number of edges in doing so. Since each cut edge in effect implies the need for ghost data, the RSB

decomposition has the result of minimizing ghost data requirements, which in turn minimizes communication. Given the nature of the contact surface used to build the graph, the decomposition has the effect of minimizing the perimeter / area ratio. Where the decomposition in the previous section tends to produce 1-D strings, the METIS decomposition tends to produces nearly square rectangles, with as many "internal" nodes as possible. Thus this decomposition more nearly satisfies the requirements for a "good" decomposition set out in Section A.5.2.

A.6.8.3 Master-Stays-Home Decomposition

As the problem size N grows, the amount of contact surface work only grows as $N^{2/3}$. If the number of processors is increased proportionally to N, the size of the bulk domain on each processor remains constant, while the size of the contact domains, if distributed evenly across the machine, actually shrinks. This can lead to a situation in which the contact communication time, due to message latency time, swamps the contact enforcement time, and there is no gain in further reducing the domain size. In fact it can become more expensive to use all processors than to use an appropriate subset.

To compensate for this "shrinking workload" effect, a decomposition was developed to keep the amount of contact work per processor constant as problem and machine were simultaneously scaled up, but for a shrinking subset (proportional to $N^{2/3}$) of all processors. The other processors do no slide calculations, and unless they have slave nodes do not participate in any slide communication. This decomposition can be accomplished quite simply by assigning each master node to the same processor that it lives on in its primary domain.

A.6.8.4 Master Side Nodal Decomposition

The preceding decompositions are all face decompositions: they assign each face to precisely one Contact Domain. Nodes are shared by more than one face: thus a method is needed to determine a nodal decomposition based on the face decomposition.

The primary goal is to assign an equal number of nodes to each Contact Domain. It is also desirable to create a "smooth" interface between domains, in order to minimize the number of required ghost nodes. Figure A-24(a) gives an example of a jagged interface, while Figure A-24(b) shows a smoother alternative. The circled nodes are ghost nodes to the Contact Domain on the left (assuming a radius of 1). It can be seen that the jagged boundary requires more ghost nodes.

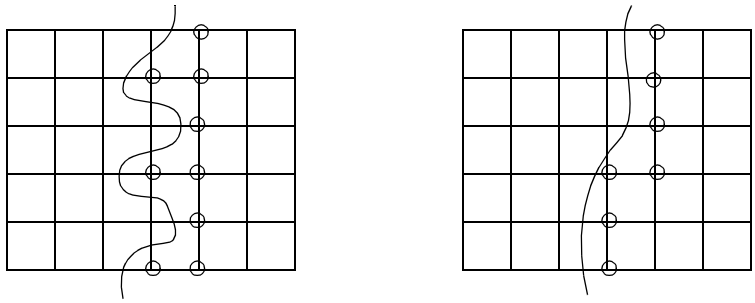


FIGURE A-24

To accomplish this end, the average number of nodes per CD is calculated. Then all nodes which fall into only one CD based on the face decomposition are assigned to the CD of their face. The remaining nodes, which are shared by more than one CD, are assigned to the lowest numbered sharing CD which does not yet have an average number of nodes. If all associated CDs have reached their average, then the nodes is assigned to the CD with the smallest current node count.

A.7 MAIN LOOP OVERVIEW

The work done each cycle during the contact phase can be divided into a number of subphases:

During PD↔PD communication each primary domain shares with its neighbor domains the updated state of its local nodes that are proxy nodes for those neighbors. Specifically, it shares updated Rule 1 and Rule 2 information (Section A.4.1.3, Section A.4.1.4) to allow the neighboring PDs to determine where to send local nodes that are in the neighborhoods of these proxies.

During the PD→CD communication subphase, each primary domain determines which contact domains need each of its local nodes, as either assigned or ghost nodes, and sends them there.

During the contact domain construction subphase, each contact domain receives all the nodes and faces sent to it (possibly redundantly) from various primary domains, and creates a new temporary index space (the Contact Domain Index Space, or Cidx) to hold its part of the problem.

During the Calculation subphase, the net force on each of the assigned nodes and its resulting new position are calculated. As part of this task the order relations between opposing sides are updated.

The CD↔CD communication subphase is actually embedded within the Calculation subphase, and allows intermediate results to be communicated between CDs, so that each can

work with updated values for ghost nodes, rather than redundantly computing those values.

Finally, during the CD→PD communication subphase the results are returned to the appropriate primary domains, and stored at their permanent (Pidx or Bidx) indices.

A.8 PD<-->PD Communication

The pattern for PD↔PD communication is static. During initialization (Section A.6.5), both sender and receiver invoices are created between each pair of neighbor primary domains. Each PD loops through the list of its neighbor domains. For each neighbor it accesses its sender invoice. For each local node in the invoice it packs into a send buffer the node's Rule 1 CD and Rule 2 CD list. The send buffer is then sent to the neighbor PD. The neighbor PD has a matching invoice, so there is no need to identify the nodes being sent, but simply the data in invoice order. The neighbor PD uses the matching invoice to copy the current data to the nodeStuff blocks of the corresponding proxy nodes.
Section A.6.5 goes into more detail about the construction of the static PD↔PD invoices.

A.9 PD<-->CD Communication

In order for a contact domain to do its calculations, it must receive the data from the primary domains. No data is retained in the contact domain from one cycle to the next (with one exception; Section A.12.3). Furthermore the contact domain is passive in the determination of its work each cycle. The primary domains determine what to send to each contact domain, and the contact domain simply works with whatever it gets.

For each local node the PD determines which CDs to send the node to, and whether it is sent as an assigned node or a ghost node, and then sends it. How this overall goal is accomplished is discussed in the following subsections.

A.9.1 Rule 1 and Rule 2 Generators

Each primary domain determines which contact domain(s) each of its local nodes must be sent to each cycle. It does this by determining a set of generators for each CD it sends to, and a secondary set of nodes generated by the set of generators.

A Rule 1 generator for a particular CD is a node that has been assigned to that CD. A given node may only be a Rule 1 generator for one CD. A Rule 2 generator is a node that is the order node of a Rule 1 generator. A node may be a Rule 2 generator for more than one CD, since nodes assigned to different CDs may share the same order node.

If a node local to a primary domain is a generator for a contact domain, then the PD sends the node to the CD. If a generator node is either local or proxy to a PD, then it generates the requirement that all its neighbor nodes that are local to the PD also be sent to that CD.

The Rule 1 generators of a CD are identically the assigned nodes of the CD. It is just a different perspective. Rule 2 generators and any generated nodes (unless they are also Rule 1) are ghost nodes of the CD.

A.9.2 Tag Lists

Tag lists are Pidx-based arrays that are associated with particular CDs, and are used to flag the nodes or faces in the primary domain that must be sent to that contact domain this

cycle. There are separate node and face tag lists. For instance, if a node with Pidx i is to be sent to CD j, then the node tag list associated with CD j will have a non-zero value in the i^{th} element.

At the start of the PD→CD subphase there are no tag lists, as the PD does not yet know which CDs are its neighbors this cycle. Each PD loops through all the nodes in that PD, including proxy nodes. For each node it examines the node's Rule 1 CD and Rule 2 CD list. For each CD that it finds there, if there are no tag lists associated with that CD yet, it allocates node and face tag lists for it and initializes them to zero. At this point it also adds the CD to the list of neighbor CDs to this PD. If the tag lists already exist, it just uses them.

For each node in the current node's neighbor node list (Section A.4.1.5) it decrements the value in the node tag list at the neighbor node's Pidx. Similarly, for each face in the current node's neighbor face list (Section A.4.1.6) it decrements the value in the face tag list at the neighbor face's Pidx. If the current node itself is a local (not proxy) node, and if the CD is the current node's Rule 1 CD (i.e. the node is assigned to that CD), then the value at its Pidx in the node tag list is overwritten with a large positive integer.

For example, consider the primary domain in Figure A-25, and assume the neighbor radius is 1.

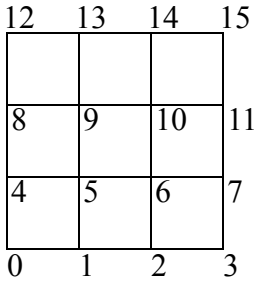


FIGURE A-25

Assume the PD->CD stage is just starting, so there are no tag lists. Suppose node 0 is assigned to CD 3 (i.e. its Rule 1 CD is 3). Then when node 0 is considered, a node and face tag list for CD 3 will be allocated and initialized:

CD 3 node Tag List

0	0	0	0	0	0	0	. . . . .	0

At this point CD 3 will be added to the list of CDs that this PD talks to. Node 0 has a node neighbor list of {0, 1, 4, 5}, so the tag list will be updated to:

CD 3 node Tag List

-1	-1	0	0	-1	-1	0	. . . . .	0

Finally, since node 0 is local and CD 3 is its Rule 1 CD, the tag list becomes:

CD 3 Node Tag List

100	-1	0	0	-1	-1	0	. . . . .	0

This tag list technique has two useful features:

1. It removes redundancy, as a single node will appear as ghost at most once
2. it distinguishes assigned nodes from ghosts: assigned are positive, ghosts negative

For example if node 1 is also assigned to CD 3, then it will decrement the tag list for Pidx's {0, 1, 2, 4, 5, 6}, and then overwrite the value at Pidx 1:

CD 3 Node Tag List

99	100	-1	0	-2	-2	-1	0	. . . . .	0

Note that Pidx 0 is still positive, while Pidx 4 and 5 are still negative.

The corresponding face lists use the decrement operator only, since faces are not distinguished between assigned and ghosts.

A.9.3 Invoices

Invoices are lists of nodes to be sent to another domain. In the case of PD→CD communication, the nodes are sent to a contact domain. As discussed in Section A.3, a given node may have several different "names", or indices, representing its order in different index spaces. The three relevant index spaces here are bulk domain or Bidx, primary domain or Pidx, and contact surface or Sidx. It is necessary to keep versions of an invoice in all three of these indexes, as explained below.

When the data is being marshalled into a buffer to be sent to a particular CD, some of the information about a particular node is stored in the bulk domain node (e.g. coordinates). Other data, which only exists for contact nodes, is stored in the nodeStuff struct at the Pidx index of the node. So both Bidx and Pidx must be known to gather all the nodal data. On the other hand, when the data is sent to the CD the Sidx version of the invoice is sent with

it, so that the contact domain knows which nodes it got. The Sidx index space is "global", in the sense that it is independent of any domain spaces. This allows the contact domain to eliminate any redundancy, when the same node is sent from multiple domains.

There are "assigned node invoices" and "ghost node invoices". The assigned node invoices contain the nodes actually assigned to the CD. The CD will update the state (position, etc.) of its assigned nodes and return the updated state to the sending primary domain(s). Ghost node invoices contain the nodes that are not assigned to that CD, but are in the neighborhood of an assigned node or its order node.

In addition to the node invoices there are also face invoices, in Bidx and Sidx versions (faces don't have a Pidx domain). These are in effect ghost invoices, since faces are not assigned to CDs.

The invoices are built directly from the corresponding tag lists. A tag list is scanned from start to finish. When a non-zero entry is found, the Pidx at which it is found is added to the Pidx version of either the assigned invoice (if the non-zero tag list entry is positive) or the ghost invoice (if the entry is negative). The Pidx is also applied to the primary domain's Pidx→Sidx and Pidx→Bidx maps to find the node's corresponding Sidx and Bidx indices, and these are added to the Sidx and Bidx versions of the invoice.

Since there are three versions of node invoices for both assigned and ghost nodes, two versions of face invoices, and two sides, master and slave, there are sixteen invoices in all associated with a particular (PD,CD) pair.

A.9.4 msgBases

A msgBase is a data structure used to control each endpoint of point-to-point communication between primary domain and contact domain. A given primary domain has a separate msgBase for each contact domain it communicates with. Likewise, each contact domain has a separate msgBase for each primary domain it communicates with. Since the set of CDs a primary domain will talk to varies from cycle to cycle, the corresponding msgBases are allocated dynamically each cycle, and then freed at the end of the cycle.

Since primary domains and contact domains include both master and slave sides, their msgBases contain fields for both sides.

In Figure A-26, it can be seen that a separate msgBase is used for each domain communicated with, with as many or few allocated as needed. For instance PD 3 has no contact surface nodes, thus it doesn't communicate with any CD, and needs no msgBase.

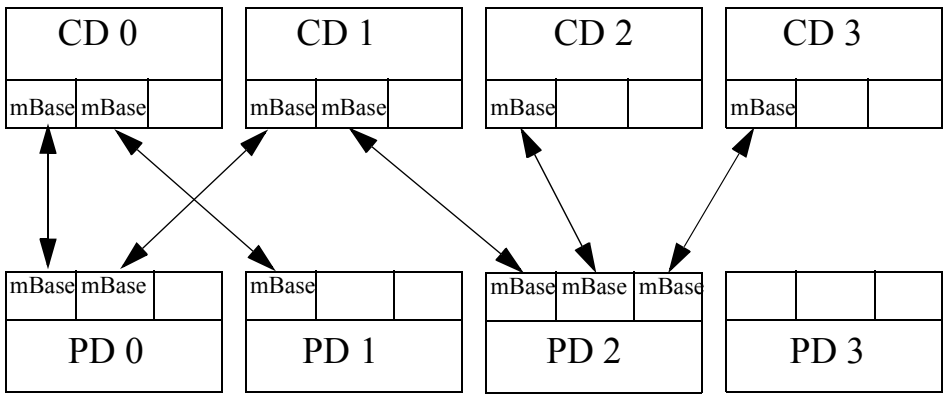


FIGURE A-26

Primary domain and contact domain msgBases vary slightly. The primary domain msgBase is discussed in the following section, followed by a section on the contact domain msgBase.

A.9.4.1 Primary Domain msgBases

If, for a given primary domain, there are any tag lists (Section A.9.2) created for a particular CD, then a msgBase is allocated to control information flow between the primary domain and that contact domain.

A primary purpose of the msgBase is to store the 16 invoices (8 master, 8 slave) but it is also used for buffer management. A high level view of the msgBase is shown in Figure A-27:

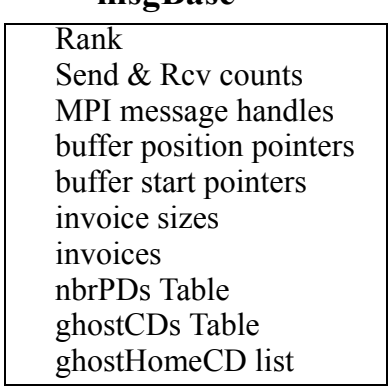


FIGURE A-27

Rank is the processor number, or MPI rank, of the contact domain associated with this msgBase.

Integer and floating-point buffers are sent separately, and later received back from the CD. The send & receive counts keep track of the sizes of all message buffers, int and float, incoming and outgoing.

The MPI message handles are used to check the status of asynchronous communication operations associated with this msgBase. For instance this processor (as a primary domain) will allocate receive buffers and issue receives of results from various contact domains associated with different msgBases, before the results are actually ready. At

some later point it will wait for the receives it issued earlier to complete, interrogating MPI via the message handles for those receives.

Buffer position pointers are used to keep track of the current locations in the various buffers being read and written.

Pointers to the buffers keep track of where the current send & receive int and float buffers are stored.

Invoice sizes are just the number of nodes in each invoice. There are only 6 unique sizes, since some of the invoices are the same length (i.e. the Bidx, Pidx and Sidx versions). These six are: master assigned nodes, master ghost nodes, slave assigned nodes, slave ghost nodes, master faces, and slave faces.

The invoices themselves are the 16 invoices described in the previous section.

Finally, the msgBase contains three tables for both master and slave sides, which are built during construction of the invoices:

The ***nbrPDs*** table has an entry for each node in the assigned or ghost invoice that has a non-empty Neighbor PD list (Section A.4.1.7). For each such node, the nbrPDs table contains the Sidx of the node, the length of its Neighbor PD list, and then the neighbor PD list itself.

The ***ghostCDs*** table has an entry for each node assigned to the CD that is also sent as a ghost node to any other CDs. For each such node, the ghostCDs table contains the Sidx of

the assigned node, the number of CDs on which the node is a ghost, and a list of those CDs.

The ***ghostHomeCD*** list has an entry for each node in the ghost invoice, and simply contains the CD where that node is an assigned node.

A.9.4.2 Contact Domain msgBases

Each contact domain needs to allocate a separate msgBase for each primary domain that it communicates with this cycle. But this brings up an issue: how does the contact domain know which primary domains will send it data?

The primary domains determine which contact domains to talk to as a result of constructing the tag lists (Section A.9.2). A simple approach to the inverse problem would be to make this information globally available. For instance, each PD could broadcast its "neighbor" CDs to all processors. However, one goal of this algorithm is to avoid all non-scalable communication, which rules out broadcasts. Instead, a more sophisticated method is used by the contact domains to determine, at the end of a cycle, which PDs will communicate with it next cycle. This ***nextTimePDs*** list is the only state preserved across cycles by the contact domain. The details of building nextTimePDs are discussed in Section A.12.3. Here the existence of the list is simply assumed.

Each contact domain loops through its nextTimePDs list and allocates a msgBase for each PD. This msgBase uses the same structure as the primary domain msgBase, but its use of the invoices is slightly different. The Sidx invoices in the CD msgBase are identical to the corresponding invoices of the PD msgBase. They are simply sent during the communica-

tion, and copied into the CD msgBase invoice. However, the contact domain doesn't make any use of Pidx or Bidx index spaces. In their place is a single index space local to the contact domain, the Cidx. (To be precise there are actually four Cidx index spaces, for master and slave, nodes and faces. But this is also true of the Pidx index space). A copy of the invoice in the Cidx space will be built during construction of the contact domain, discussed in Section A.10.3.

A.9.5 Sending and Receiving

As a preliminary step in the communication, the primary domain loops through its set of msgBases, and calculates the size of int and float buffers required for each msgBase. It does this by extracting the invoice lengths from the msgBase and multiplying by appropriate factors (number of ints per node, number of floats per face, etc.), then adding the lengths of any tables that must be sent.

These buffer sizes are used to allocate the send buffers for the msgBase, and they are also sent to the corresponding contact domain. The contact domain then uses the lengths to allocate the receive buffers for the connection, and issues asynchronous reads of the expected int and float buffers. By this time the contact domain msgBases have already been allocated, based on the nextTimePDs list, so the CD loops through its msgBases to determine which PDs to expect to receive lengths from. (Note: each processor at this point is functioning at least potentially as both a primary domain and a contact domain, so the contact domain cannot wait synchronously for data. It still has to send data as a primary domain. To wait synchronously at this point would likely result in deadlock).

The next stage is to fill the send buffers. The primary domain loops through its set of msg-Bases again. For each msgBase it fills an integer and floating point buffer. It starts with the master side, then the slave side. For each side it first puts the lengths of the various invoices into the integer buffer: (i.e assigned node count, ghost node count, and face count). Then it copies the Sidx versions of the corresponding invoices into the integer buffer. For each assigned node it copies the Sidx of the assigned node's order node and order face. Finally, it copies the nbrPDs table, the ghostCDs table, and the ghostHomeCD list (Section A.9.4.1) into the integer buffer.

For each node, assigned or ghost, the coordinates, velocities, and initial accelerations (computed as though nodes were free-surface) as well as the amount of mass apportioned to the node are packed into the floating-point send buffer. For each face, the current stress in the bulk zone underlying the face is packed into the float buffer.

When all the data for both buffers have been packed into the buffers, the buffers for this msgBase are sent to the destination CD, using a pair of asynchronous sends.

The destination CD has already allocated the receive buffers and issued asynchronous receives, as discussed earlier in this section. This allows the data transfer to occur, without further interaction from either primary domain or contact domain. The primary domain can now move on to the next msgBase in its list and repeat the whole process.

When all sends for all primary domain msgBases have been completed, the processor switches roles and, as a contact domain, loops through all its contact domain msgBases. For each msgBase, it issues a Wait for completion of the asynchronous receive that it

issued earlier. This Wait will not return until the data has all been transferred to the int and float message buffers of that msgBase.

After all the contact domain msgBases have received their buffers, the processor switches back to primary domain mode, waits for all its asynchronous sends to complete, and then deallocates the storage used for the send buffers.

To see why the last Waits for send completions are necessary, consider Figure A-28:

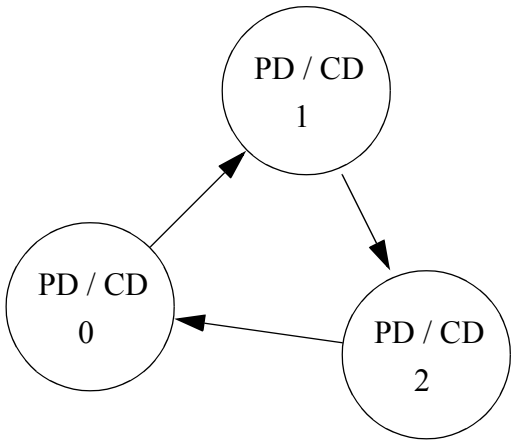


FIGURE A-28

Each primary domain shares a processor with a contact domain. In this simple example each of the primary domains and each of the contact domains have one msgBase. PD 0 sends to CD 1, PD 1 sends to CD 2 and PD 2 sends to CD 0. Suppose that processor 1 is way behind processors 0 and 2, due to some load imbalance or OS interrupt. So it issues no sends or receives until some later time. Then PD 0 issues an asynchronous send to CD 1, but the send doesn't happen because CD 1 hasn't issued a receive yet. Following that, CD 0 issues an asynchronous receive from PD 2, and then waits for completion of the receive. If PD 2 is keeping up and performs the send, this wait will complete. But PD 0 cannot then release its send buffers to CD 1, since the transfer hasn't happened yet. Thus

the Wait for primary domain send completions is necessary, before send buffers can be released.

A.10 Building the Contact Domain

After the PD→CD communication pass is complete, the contact domains are in a raw form. Each contact domain has a number of msgBases, each associated with a source primary domain, and each msgBase has an int and a floating point buffer containing the contributions from that primary domain. The next task is to turn this raw data into a cohesive and integrated form from which the contact calculations can be made.

A.10.1 Initializing the Sidx Invoices

The contact domain loops through its msgBases. For each msgBase, it firsts extracts the lengths of the various invoices from the int buffer and copies them to the appropriate fields of the msgBase. Then, in lieu of physically copying the Sidx invoices from the int buffer into newly allocated storage, it simply sets the invoice pointers in the msgBase to point into the int buffer at the appropriate invoice location. The indices in the Sidx invoices are independent of any domain, or in other words, of the parallel decomposition, and so serve as global IDs of the nodes and faces involved.

A.10.2 Building the Sidx<–>Cidx Maps

Each node and face in the contact domain, whether assigned or ghost, must be assigned a local or Cidx index so that the code knows how to access it. Normally the index represents the offset from the beginning of an array. For instance, if X is the array containing the X

coordinates of nodes, and if a node is assigned a Cidx index i, then the x coordinate for this node will be stored at X[i].

The same node may be sent from multiple primary domains. This will occur when a node is on the boundary between primary domains. All such domains will consider the node to be local to it, and will send copies to any contact domains that require the node. Each primary domain will have its own Bidx and Lidx indices for the node, but they will all have the same Sidx. Thus the redundancy can be removed upon arrival in the CD by allowing only one Cidx for each Sidx. This mapping, between an Sidx and its corresponding Cidx, is called the Sidx→Cidx map. The inverse mapping is, unsurprisingly, called the Cidx→Sidx map.

To build these maps, the CD first loops through its msgBases, extracts all the Sidx invoices of a given category (e.g. master assigned nodes), and concatenates them. Then it sorts this concatenated list of Sidx's and removes any redundant entries. The result is a list of the nodes (or faces) on the CD in that category. The following example illustrates this.

In this example there are two CD msgBases. For brevity, only the master side invoices are shown.

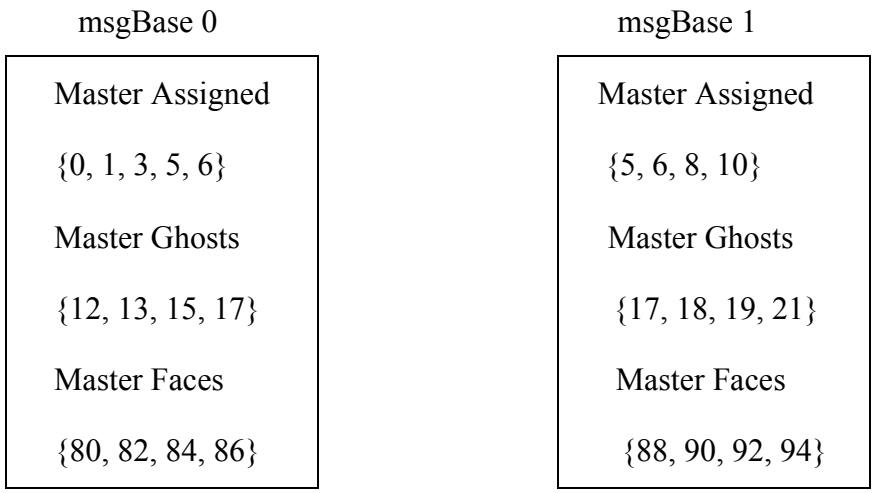


FIGURE A-29

The master assigned nodes in this example will be concatenated into the list:

{0, 1, 3, 5, 6, 5, 6, 8, 10}

then sorted to produce the list:

{0, 1, 3, 5, 5, 6, 6, 8, 10}

Then uniq'd to produce the list:

{0, 1, 3, 5, 6, 8, 10}.

Similarly the master ghosts are combined into the list:

{12, 13, 15, 17, 18, 19, 21}

and the master faces:

{80, 82, 84, 86, 88, 90, 92, 94}

Note that there are no redundant faces. A contact face always belongs to one and only one domain.

The next step is to concatenate the master assigned nodes and master ghosts, and separately the slave assigned nodes and slave ghosts. For the example above this leads to the list of master nodes:

{0, 1, 3, 5, 6, 8, 10, 12, 13, 15, 17, 18, 19, 21}.

This list is in effect the master Cidx -> Sidx map. This is more explicit in the following figure:

Cidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Sidx	0	1	3	5	6	8	10	12	13	15	17	18	19	21

FIGURE A-30

The slave map is produced in a similar fashion.

The inverse Sidx→Cidx map can be easily generated from the Cidx→Sidx map. For the above example, an array as long as the Sidx index space is allocated and initialized to -1. Then the Cidx→Sidx map is scanned. For entry i (i.e. Cidx i) its corresponding Sidx j is read. Then at the jth location of the Sidx→Cidx map, the value i is stored.

For the above example, assuming the Sidx index space is of length 24, the resulting Sidx→Cidx map is:

Sidx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Cidx	0	1	-1	2	-1	3	4	-1	5	-1	6	-1	7	8	-1	9	-1	10	11	12	-1	13	-1	-1

FIGURE A-31

A.10.3 Cidx Invoices

The Sidx -> Cidx maps can now be used to create Cidx versions of the various invoices in each msgBase. The Cidx invoices will be used when the nodal data is read in from the int and float buffers, to indicate at which (Cidx) index in the various CD data arrays to store

the data for a given node. The Cidx invoices are also used later to gather the data to be returned to the primary domain corresponding to each msgBase, after contact processing is completed.

To create a Cidx invoice, first an int array is allocated of the same length as the corresponding Sidx invoice. The the Sidx invoice is scanned. At a given offset i in the Sidx invoice is an Sidx index j. j is applied to the Sidx→Cidx map to find the Cidx index k corresponding to Sidx index j. k is then written into the ith location of the Cidx invoice under construction.

Returning to the previous example, msgBase 1 had a master assigned node Sidx invoice {5, 6, 8, 10}. To create the corresponding Cidx invoice, an array of 4 ints is first allocated. Then the first index in the Sidx invoice, 5, is applied to the Sidx -> Cidx map of Figure A-31 to get the corresponding Cidx, 3. Processing the whole Sidx invoice produces

the Cidx invoice {3, 4, 5, 6} for msgBase 1 master assigned nodes. This same process is applied to all other invoices in both msgBases to produce Cidx versions of all invoices:

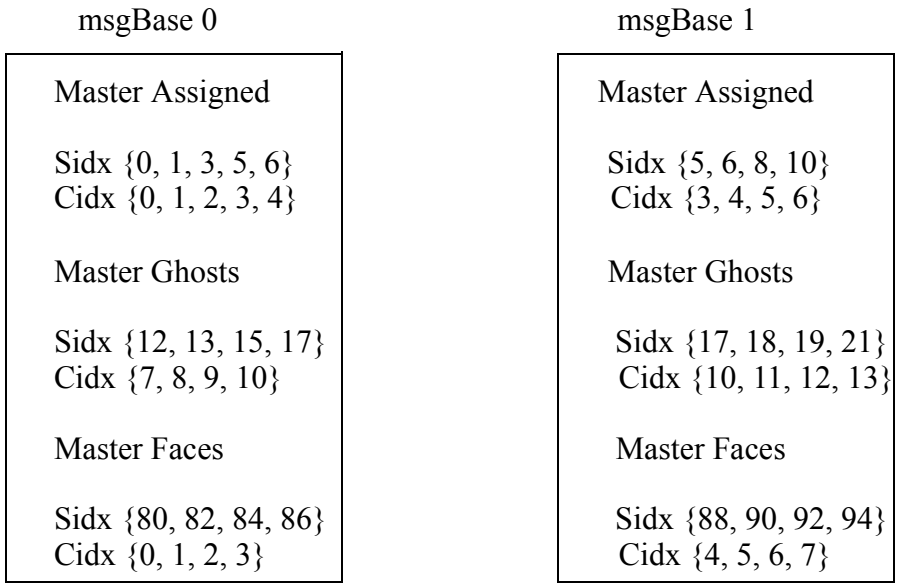


FIGURE A-32

A.10.4 Cidx Connectivity

The Connectivity and the Dual Connectivity were defined in Section A.2.2. In the current version of the algorithm, the full connectivity information for each side exists on all processors. However, it is expressed in Sidx indices. Part of the initialization of the contact domain each cycle involves creating a more limited set of connectivity information, just involving faces and nodes on the contact Domain, and expressed in the more immediately usable Cidx index spaces.

The Sidx index of a face provides the offset into the full connectivity table of the list of nodes that constitute that face. These nodes are themselves expressed in Sidx indices. To create a local version of connectivity, the Cidx→Sidx face map (master or slave) is looped through. For each Cidx, the corresponding Sidx is used to get the four nodes of the face.

The Sidx→Cidx node map is used to convert these nodes from Sidx to their local Cidx indices, and the list of nodes is then copied into the local connectivity at the Cidx index of that face.

After this local version of the connectivity list is built for each side, the dual connectivity is constructed, which supplies for each Cidx node a list of Cidx faces that touch the node. To accomplish this, the set of faces in the connectivity list is looped through. For each face, its Cidx is added to the dual list of each of the four nodes in the face.

A.10.5 Reading Nodal and Face Data

The actual data (nodal coordinates, facial stresses, etc.) can now be read in, primarily from the float buffers associated with the msgBases.

First, all the local storage to hold this data is allocated. Then the contact domain loops through its set of msgBases. For each msgBase it loops through the invoice set in predetermined order (i.e. master assigned nodes in invoice order first, then master ghosts, etc.). The data was stored into the buffers by the corresponding primary domain in the same predetermined order. As the data for a particular node or face is arrived at in the buffers, the Cidx version of the invoice just described (Section A.10.3) supplies the Cidx index into the various arrays at which to store the incoming data.

The nbrPDs, ghostCDs, and ghostHomeCD tables (Section A.9.4.1) are read into the msgBase and processed from there to assign each node a list of PDs in its proximity (as a primary domain node), each assigned node a list of CDs where the node resides as a ghost,

and each ghost node the CD where the ghost is unique. Pointers to each of these lists are stored in Cidx-based arrays.

A.10.6 CD <-> CD Invoices

During the contact calculations, intermediate values computed for assigned nodes are sent to neighboring CDs where the nodes are ghost. This greatly reduces the number of ghost nodes required on any given CD and the amount of redundant calculations made involving those ghost nodes. As in PD↔PD communication, the CD↔CD communication is "local". A neighbor CD is simply one that shares (as a ghost) one or more assigned nodes. The number of neighbors does not grow with problem size or processor count.

In addition to the primary use of CD↔CD communication, a secondary use supports the mechanism for determining which PDs will communicate with the CD next cycle. This was touched on in Section A.9.4.2, when the nextTimePDs list was introduced, and will be expanded in Section A.12.3. It is part of the mechanism for allowing the overall communication pattern to evolve without making any use of global or nonscalable communication.

To build the CD↔CD invoices, the CD loops through all its assigned nodes for each side independently. For each node it examines the node’s ghostCDs list. For each CD in the node’s ghostCDs list, it adds the node’s Sidx index to an invoice specific to that CD and contact side.

After this process completes, each CD has a set of invoices, master and slave, for each neighbor CD that it sends to, and the invoices contain a list of the Sidx indices of assigned nodes it will send updated values for, to the destination CD.

Unlike PD↔PD communication, CD↔CD communication is not symmetrical. For instance, CD 1 may have an assigned node which is also a ghost on CD 2, and yet CD 2 may have no assigned nodes that are also ghost on CD 1. Also unlike PD↔PD communication, the communication pattern is dynamic, not just in which nodes are sent but even in which CDs communicate from one cycle to the next. So a second task of the CD↔CD setup is for each CD to determine which CDs will send to it. To do this, the CD loops through its ghost nodes, master and slave, and builds a list of the home CDs of these nodes, removing redundant entries.

A.11 Contact Calculation

In this section the algorithm used to actually enforce the contact between the two sides is described. This discussion will be focused on the work done by a single contact domain. Each contact domain is constructed to be largely self-contained for the purpose of contact calculations. An exception to this is when intermediate values for ghost nodes are updated, from the domain where they are assigned.

The goal of the contact calculation is to determine the new positions, velocities and order nodes of each assigned node in the contact domain. These tasks are broken down into a number of subtasks described in the following subsections.

A.11.1 Parametric representation of faces

Each contact face, determined by its four corner nodes, is represented as a bilinear surface, so that the entire contact side is a continuous, piecewise differentiable surface. The equation of the surface for a given face is determined by the coordinates of its corner nodes. Since the face is a 2-D surface in 3-D space, it is most simply represented by using two parameters, ***s*** and ***t***.

Let $\overline{X}(s,t)$ represent the surface, so that for a particular s and t, $\overline{X}(s,t)$ is the vector of cartesian coordinates at that s and t. Another way to express this function is:

$$\begin{aligned} x &= x(s,t) \\ y &= y(s,t) \\ z &= z(s,t) \end{aligned} \qquad \text{(EQ 1)}$$

Consider the face in Figure A-33. The coordinates at the four corners, $\overline{X}_1$, $\overline{X}_2$, $\overline{X}_3$, and $\overline{X}_4$ (where $\overline{X}_1 = (x_1, y_1, z_1)$, etc.), are known. Note that these corner nodes are not in general coplanar.Define the parametric coordinates s and t so that at $\overline{X}_1$, $s = -1$ and $t = -1$, and so on, as shown in the figure.

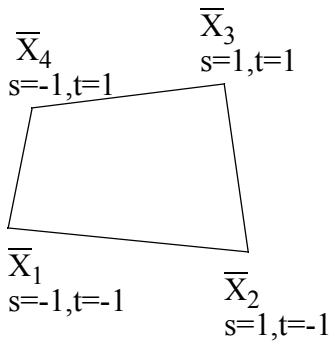


FIGURE A-33

From these known values at the corners it is quite simple to write the parametric form of the surface:

$$\bar{X}(s,t) = \frac{1}{4}[\bar{X}_1(1-s)(1-t) + \bar{X}_2(1+s)(1-t) + \bar{X}_3(1+s)(1+t) + \bar{X}_4(1-s)(1+t)]$$

To see that this is true, plug in the (s, t) at any of the corners. Three of the terms become zero, and the remaining term evaluates to $\overline{X}_i$, the Cartesian coordinates of the corner.

Further, $\overline{X}(s,t)$ is clearly bilinear. If either s or t is held constant, $\overline{X}(s,t)$ becomes linear in the remaining parameter.

Another convenient form for calculation is:

$$\bar{X}(s,t) = \bar{A} + \bar{B}s + \bar{C}t + \bar{D}st \qquad \text{(EQ 2)}$$

where:

$$\begin{aligned}
\bar{A} &= \frac{1}{4}[\bar{X}_1 + \bar{X}_2 + \bar{X}_3 + \bar{X}_4] \\
\bar{B} &= \frac{1}{4}[-\bar{X}_1 + \bar{X}_2 + \bar{X}_3 - \bar{X}_4] \\
\bar{C} &= \frac{1}{4}[-\bar{X}_1 - \bar{X}_2 + \bar{X}_3 + \bar{X}_4] \\
\bar{D} &= \frac{1}{4}[\bar{X}_1 - \bar{X}_2 + \bar{X}_3 - \bar{X}_4]
\end{aligned} \qquad \text{(EQ 3)}$$

The set {A, B, C, D} for a given face is referred to as the parametric coefficients for that face. The parametric coefficients of each face in the contact domain are calculated and stored. Note that they are vector quantities.

A.11.2 Basis Functions

In this section the discussion of the previous section is cast in a more general form. Consider again the equation:

$$\bar{X}(s,t) = \frac{1}{4}[\bar{X}_1(1-s)(1-t) + \bar{X}_2(1+s)(1-t) + \bar{X}_3(1+s)(1+t) + \bar{X}_4(1-s)(1+t)]$$

If we define:

$$\begin{aligned} \Phi_1 &= \frac{1}{4}(1-s)(1-t) \\ \Phi_2 &= \frac{1}{4}(1+s)(1-t) \\ \Phi_3 &= \frac{1}{4}(1+s)(1+t) \\ \Phi_4 &= \frac{1}{4}(1-s)(1+t) \end{aligned} \qquad \text{(EQ 4)}$$

then we can write the equation for the surface of the face as

$$\bar{X}(s,t) = \bar{X}_1\Phi_1 + \bar{X}_2\Phi_2 + \bar{X}_3\Phi_3 + \bar{X}_4\Phi_4 = \sum \bar{X}_i\Phi_i(s,t) \qquad \text{(EQ 5)}$$

The Φ_i are referred to as the ***Basis Functions*** and are a fundamental building block of the Finite Element Method. (Actually, in Finite Elements the Φ_i as defined here would be considered restrictions of the basis functions to the single contact face under consideration).

A defining characteristic of the basis functions are that they each take a value of 1 at one node and 0 at all other nodes.

The basis functions are useful for interpolating the values of either vector or scalar-valued functions anywhere on the surface of the contact face. Given any function F defined as F_i at the four corners i of a face, at any intermediate (s,t) F can be bilinearly interpolated as

$$F(s,t) = \sum F_i\Phi_i(s,t) \qquad \text{(EQ 6)}$$

In fact, this is exactly how the surface itself is defined: $X(s,\,t)\;=\;\sum X_i\Phi_i(s,\,t)$, where the $\overline{X}_i$ are coordinates.

In succeeding sections, the basis functions will be used to interpolate a number of different functions, such as normal to the surface, and normal force to the surface at a given (s,t). The method is to first know or calculate the values of the function at the nodes, F_i, then to determine (s,t), then calculate $\Phi_i(s,\,t)$ for each i at the given (s,t), and finally to compute $F(s,\,t)\;=\;\sum F_i\Phi_i(s,\,t)$

A.11.3 Face-Centered Areas

Next the area of each face in the contact domain is calculated. This can be expressed as:

$$A\;=\;\int_{-1}^{1}\int_{-1}^{1}\left|\frac{\partial}{\partial s}\bar{X}(s,\,t)\times\frac{\partial}{\partial t}\bar{X}(s,\,t)\right|ds\,dt \qquad \text{(EQ 7)}$$

One-point quadrature is used to evaluate this integral, with the integrand being sampled at $s\;=\;t\;=\;0$. At this point:

$$\begin{aligned}\frac{\partial}{\partial s}\bar{X}(s,\,t)\Big|_{s,\,t\,=\,0}&=\;[\bar{B}+\bar{D}t]\big|_{s,\,t\,=\,0}\;=\;\bar{B}\\ \frac{\partial}{\partial t}\bar{X}(s,\,t)\Big|_{s,\,t\,=\,0}&=\;[\bar{C}+\bar{D}s]\big|_{s,\,t\,=\,0}\;=\;\bar{C}\end{aligned} \qquad \text{(EQ 8)}$$

so that:

$$\left|\frac{\partial}{\partial s}\bar{X}(s,\,t)\times\frac{\partial}{\partial t}\bar{X}(s,\,t)\right|\Big|_{s,\,t\,=\,0}\;=\;\left|\bar{B}\times\bar{C}\right| \qquad \text{(EQ 9)}$$

With some algebra, it can be shown that:

$$\left|\bar{B} \times \bar{C}\right| = \sqrt{(\bar{B} \cdot \bar{B})(\bar{C} \cdot \bar{C}) - (\bar{B} \cdot \bar{C})(\bar{B} \cdot \bar{C})} \quad \text{(EQ 10)}$$

So to find the area of a given contact face, the parametric coefficients for the face are retrieved and used to calculate

$$A = 4\sqrt{(\bar{B} \cdot \bar{B})(\bar{C} \cdot \bar{C}) - (\bar{B} \cdot \bar{C})(\bar{B} \cdot \bar{C})} \quad \text{(EQ 11)}$$

A.11.4 Node-Centered Areas

For purposes to be described later (Section A.11.9) it is useful to apportion the contact surface area among the nodes. The approach used is to first initialize each nodal area to zero. Then, in a loop over all faces, the area of each face (computed above) is divided evenly among each of its four corner nodes and added to the accumulating area for those nodes.

A.11.5 Node-Centered Normals

At a later point in the algorithm (Section A.11.7) it will be necessary to determine the normal to the surface at arbitrary points on each contact side. The normal based directly on the piecewise-bilinear representation is not sufficient, because it changes discontinuously at face edges. Consider Figure A-34. $\hat{n}_1$ and $\hat{n}_2$ are well-defined, but $\hat{n}_3$ is not.

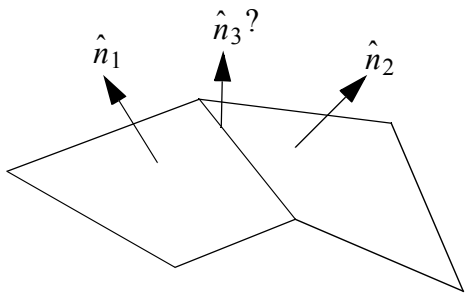


FIGURE A-34

In order to provide a representation of the surface normal that is everywhere continuous, the normals are first calculated at each node. These values are then used as a basis for bilinear interpolation at all other points.

To calculate the normal at a node, the node is considered the juncture of all segments that it touches.

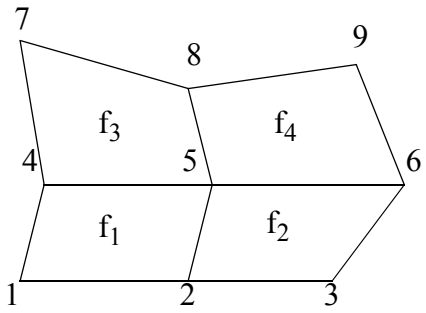


FIGURE A-35

In Figure A-35, node 5 is at the juncture of the four faces f_1, f_2, f_3, and f_4. Let $R_{i,j}$ be the directed line segment from node i to node j. Then the normal at node 5 is *defined* to be the normalized sum of contributions from f_1 through f_4. The contribution from f_1 is $R_{5,4} \times R_{5,2}$. This vector is perpendicular to the triangle formed by nodes 2, 4, and 5, and is proportional to its area. Similarly, f_2's contribution is $R_{5,2} \times R_{5,6}$, and so forth.

After the contributions from all adjoining faces are summed (vectorially), the resultant vector is normalized to unit magnitude.

A.11.6 Node-Centered Normal Force

A stress tensor is defined on each contact face, which actually originates in the stress tensor for the zone underlying that face in the Bulk Domain. Consider Figure A-36, which shows the same faces as Figure A-35, but also shows the underlying zones.

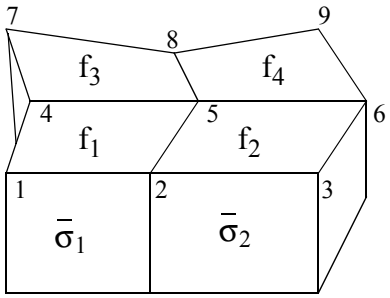


FIGURE A-36

In this figure $\bar{\sigma}_i$ is the second-order stress tensor in the zone underlying contact face f_i. For each face a contribution $\bar{F}_i$ to the normal force at the center node 5 is first calculated. Consider the face f_1. Represent the stress tensor in cartesian index form: $\sigma_{i,j}$. Suppose n_i, again in index form, is the normal vector for the face calculated by $R_{5,4} \times R_{5,2}$ as discussed in the preceding section. Then the stress vector on the face is the tensor product $\sigma_{i,j} n_j$. This vector is not necessarily normal to the face. In order to calculate the normal component it must again be "dotted" with the normal: i.e. $\sigma_{i,j} n_j n_i$.

This is not yet quite correct, because it provides force/area, while what is needed is actual force, since the mass will be treated as concentrated at the node. To accommodate this, the cross-product itself, $R_{5,4} \times R_{5,2}$, is used as n_i, which is the normal scaled by the area. Since this appears twice in the product $\sigma_{i,j} n_j n_i$, the normal component of the force is wrong by a factor of the area of the face. This is corrected when the contributions from all

adjoining faces are summed, by dividing the result by the sum of areas of contributing faces, thus providing an area-weighting of the contributions of all adjoining faces.

A.11.7 Parametric Coordinates

In order to calculate the force exerted on a contact node n_i, referred to as the ***active node***, by the contact surface side opposite the node, it is necessary to first pinpoint the location on the surface that is directly opposite the active node. The order face (Section A.4.1.2) is already known, so the problem reduces to finding the parametric coordinates (s,t) for that point on the surface of the order face at which a normal to the order face passes through n_i. Figure A-37 depicts the situation:

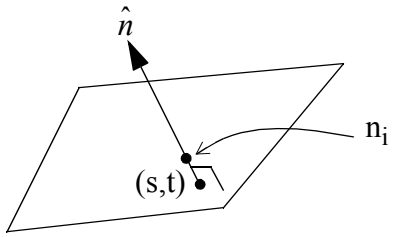


FIGURE A-37

In Figure A-37, the active node n_i is shown together with its order face. At the selected (s,t), the normal to the order face passes through the node n_i. The point on the surface at this (s,t) is referred to as the ***image node*** (although it is not really a node), and (s,t) are the parametric coordinates of the image node.

The normal at a given (s,t) of the contact face, as discussed in Section A.11.2, is based on interpolation of the nodal normals defined at the corners of the order face. Suppose the

normals at the four corners of the order face, calculated by the method described in Section A.11.5, are $\hat{n}_1$, $\hat{n}_2$, $\hat{n}_3$, and $\hat{n}_4$. Then at (s,t), the normal is given by

$$\hat{n}(s,t) = \sum \hat{n}_i \Phi_i(s,t) \qquad \text{(EQ 12)}$$

(EQ 13)

The parametric coordinates (s,t) of the image node are determined iteratively. A first approximation is made by setting (s^0, t^0) to one of the corners (i.e. $(\pm 1, \pm 1)$) if the node is close enough to that corner; otherwise $(s^0, t^0) = (0, 0)$, the middle of the order face.

At a given (s^i, t^i) the spatial coordinates are given by Eq 6:

$$\bar{X}^i = \sum_k X_k \Phi_k(s^i, t^i) \qquad \text{(EQ 14)}$$

where X_k are the spatial coordinate vectors of the four corners.

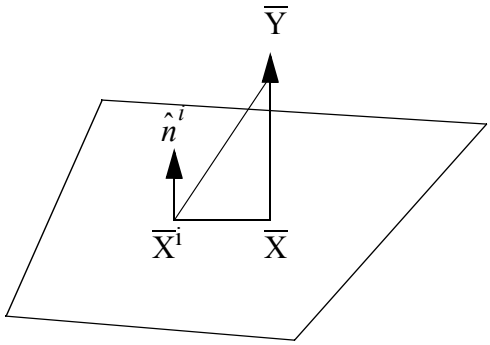


FIGURE A-38

In Figure A-38, $\overline{Y}$ is the current active node, $\overline{X}$ is its associated (unknown) image node, $\overline{X}^i$ is the i^{th} estimate of $\overline{X}$, and $\hat{n}^i$ is the normal to the face at $\overline{X}^i$. In the i^{th} step of the iteration, the component of $\overline{Y}$ - $\overline{X}^i$ that is tangential to the order face at $\overline{X}^i$ is added to $\overline{X}^i$ to approximate $\overline{X}$.

$$\bar{X}^{i+1} = \bar{X}^{i} + (\bar{Y} - \bar{X}^{i}) - [(\bar{Y} - \bar{X}^{i}) \cdot \hat{n}^{i}]\hat{n}^{i} \qquad \text{(EQ 15)}$$

If the order face is planar, a single step will yield $\overline{X}$. If not, successive steps will converge to $\overline{X}$. In any case, what is really wanted is the value of (s,t) at $\overline{X}$. To determine (s^{i+1}, t^{i+1}), a Taylor Series expansion about $\overline{X}^{i}$ is performed:

$$\bar{X}^{i+1} = \bar{X}^{i} + \frac{\partial \bar{X}^{i}}{\partial s}(s^{i+1} - s^{i}) + \frac{\partial \bar{X}^{i}}{\partial t}(t^{i+1} - t^{i}) \qquad \text{(EQ 16)}$$

where $\frac{\partial \bar{X}^{i}}{\partial s}$ is the partial derivative of $\frac{\partial}{\partial s}\bar{X}(s, t)$ evaluated at (s^{i}, t^{i}), and similarly for $\frac{\partial \bar{X}^{i}}{\partial t}$.

Recalling the previous expression for $\overline{X}$ using the parametric coefficients (Section A.11.1):

$$\bar{X} = \bar{A} + \bar{B}s + \bar{C}t + \bar{D}st \qquad \text{(EQ 17)}$$

Then:

$$\begin{aligned} \frac{\partial \bar{X}^{i}}{\partial s} &= \bar{B} + \bar{D}t^{i} \\ \frac{\partial \bar{X}^{i}}{\partial t} &= \bar{C} + \bar{D}s^{i} \end{aligned} \qquad \text{(EQ 18)}$$

Equating the two expressions for $\overline{X}^{i+1}$ in Eq 15 and Eq 16:

$$\bar{Y} - \bar{X}^{i} - [(\bar{Y} - X^{i}) \cdot \hat{n}^{i}]\hat{n}^{i} = \frac{\partial \bar{X}^{i}}{\partial s}(s^{i+1} - s^{i}) + \frac{\partial \bar{X}^{i}}{\partial t}(t^{i+1} - t^{i}) \qquad \text{(EQ 19)}$$

Since this is a vector equality it is equivalent to three scalar equations in two unknowns, s^{i+1} and t^{i+1}. Out of the three equations, the pair with the largest Jacobian determinant is chosen, and solved for the next approximation, $\bar{X}^{i+1} = \bar{X}(s^{i+1}, t^{i+1})$.

In the current implementation, this algorithm is always performed a fixed number of iterations.

Once the parametric coordinates of the image node have been determined to sufficient accuracy, they can be used to interpolate the values at the image node of all the physical quantities associated with nodes, such as nodal mass, velocity, acceleration, normal stress, etc., using the formula of Eq 6:

$$F = \sum F_i \Phi_i(s, t) \qquad \text{(EQ 20)}$$

where, again, the F_i are the values at the corner nodes, and the Φ_i are the basis functions.

Calculating the parametric coordinates is the single most expensive operation performed by the contact domain. In some cases, specifically when the contact surface is very flat, a simpler algorithm can be employed. In the simpler approach, a tangent plane at the midpoint (i.e. (s, t) = (0, 0)) of the order face is taken, the normal to the plane passing through the active node is calculated, and the (s, t) at which this normal intersects the order face are taken as the parametric coordinates of the associated image node. A closed form algebraic solution exists, and is much quicker to calculate than the iterative solution. However, the iterative method is usually necessary, and is the default.

A.11.8 Interpolations at the Image Node

Once the parametric coordinates of the image node are known, the basis functions $\Phi_i(s, t)$ (Section A.11.2) are evaluated at those coordinates. The Φ_i can then be used to interpolate various nodal quantities at the image node, such as nodal mass and node-based normal force. For example the normal force at the image node is

$$F = \sum F_i \Phi_i \quad \text{(EQ 21)}$$

where F_i are the values of the node-centered normal forces (Section A.11.6) at the four corners of the order face, which contains the image node.

A.11.9 Pressure Adjustment to Acceleration

The next step is to account for the pressure, or normal force per unit area, exerted by the area surrounding the image node on the area surrounding the active node. This is the completion of the process by which the bulk node accelerations were calculated, prior to Contact calculations. At that time, acceleration of the contact nodes was computed from a weighted sum of the stresses in each of the elements that shared the node. That is, the nodes were treated as nodes on a free surface. In reality, however, elements on the other side of the contact surface contribute stress (in this case pressure, since the surface is assumed frictionless) to the net stress at the node. This step corrects the acceleration to account for this pressure.

Theoretically this step could be skipped, since normal stress is continuous across a contact surface, and thus will cancel out during center-of-mass acceleration calculation (Section A.11.10). However, the approach taken here has a smoothing effect on instabilities, since it results in accelerations entering the center-of-mass calculations that are more nearly equal.

To adjust acceleration for pressure effects, each side will in turn be taken as the active side. First, the average pressure for each active face is calculated. For each of the corner nodes of the face, the average of the normal force at that node and the normal force at the

image node is taken. Next, the average of these averages over the four nodes of the face is calculated. This is the average pressure P for that face.

Then the pressure is converted from a scalar areal density to a vector force by multiplying it by one quarter of the area of the face, and assigning it a direction normal to the face. This is done simply by multiplying it by $\overline{B}$ x $\overline{C}$. As discussed in Section A.11.3:

$$\bar{B} \times \bar{C} = \frac{\partial}{\partial s}\bar{X}(s, t) \times \frac{\partial}{\partial t}\bar{X}(s, t) \qquad \text{(EQ 22)}$$

evaluated at (s, t) = (0, 0). $\frac{\partial}{\partial s}\bar{X}(s, t)$ is a vector tangent to the face surface in the direction of increasing s, while $\frac{\partial}{\partial t}\bar{X}(s, t)$ is tangent to the same surface in the direction of increasing t. Thus together they form the tangent plane at (s, t) = (0, 0), and their cross product is normal to the face at that point.

Furthermore, it was shown in Section A.11.3 that $\overline{B}$ x $\overline{C}$ = $\frac{1}{4}$ area of the face. So $-P\bar{B} \times \bar{C}$ is one quarter of the force exerted by pressure on the face, which is just the amount apportioned to each of the four corner nodes (the negative sign is due to the sign convention for pressure, which is opposite that of stress).

Next, in a loop over all active nodes, each node loops through each of the faces it is directly connected to, and accumulates that face's $-P\bar{B} \times \bar{C}$ into the net force $\bar{F}_P$ on that node due to pressure.

Finally, the force F_P on the node is converted to an acceleration increment $\bar{a}_P$ by Newton's Law:

$$\bar{a}_P = \bar{F}_P / m \qquad \text{(EQ 23)}$$

where m is the mass assigned to the node.

$\bar{a}_P$ is added to the acceleration of the node calculated during the bulk calculations and sent to the contact domain. This adjusted acceleration is the input to the center-of-mass acceleration computation described in the next section.

A.11.10 Center of Mass Adjustment to Acceleration

A final correction is made to the acceleration of each assigned node, prior to integrating its new position. This correction treats the active node and its image node as a system, and calculates the component of acceleration of the center of mass of this system, in the direction orthogonal to the contact surface. This component replaces the orthogonal acceleration of the active node, while its tangential acceleration is left unchanged.

This step ensures continuity of momentum orthogonal to the surface, which must hold physically, while allowing discontinuity of momentum tangentially as the two sides slide relative to each other.

Let the subscript "a" refer to the active node, and the subscript "i" refer to the image node. The acceleration of the image node, and the mass apportioned to it, must be interpolated from the image node's parametric coordinates and the corresponding values at the corner nodes of the order face:

$$m_i = \sum m_k \Phi_k(s,t)$$
$$\bar{a}_i = \sum \bar{a}_k \Phi_k(s,t) \quad \text{(EQ 24)}$$

Then the center-of-mass acceleration is:

$$\bar{a}_{com} = \frac{m_a a_a + m_i a_i}{m_a + m_i} \quad \text{(EQ 25)}$$

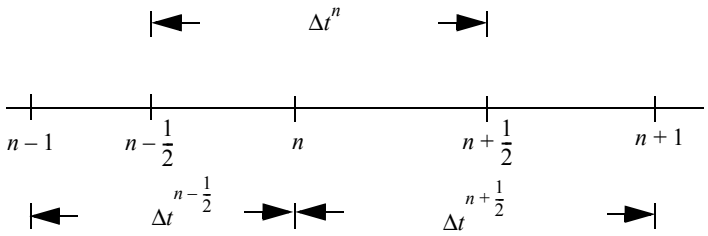
and the component of $\bar{a}_{com}$ normal to the surface is $\bar{a}_{com} \cdot \hat{n}_a$, where $\hat{n}_a$ is the node-centered normal of the active node (Section A.11.5). The normal component of the active node's acceleration is replaced with this center-of-mass normal component, while the tangential component is left unchanged:

$$\tilde{a}_a = \bar{a}_a + [(\bar{a}_{com} - \bar{a}_a) \cdot \hat{n}_a]\hat{n}_a \quad \text{(EQ 26)}$$

This correction is applied to all assigned nodes, master and slave, symmetrically on the contact domain.

A.11.11 Calculating New Node Velocities and Positions

Once the corrected values for nodal acceleration are available, it is a simple matter to calculate the velocities and positions. The bulk calculation uses a staggered time grid, where the velocities are calculated at the "half-steps", and the accelerations and positions at the "whole-steps":



Note that $\Delta t^{n+\frac{1}{2}} = \frac{\Delta t^n + \Delta t^{n+1}}{2}$. Central differences are used to build the integration scheme:

$$\bar{a}^n = \frac{\bar{v}^{n+\frac{1}{2}} - \bar{v}^{n-\frac{1}{2}}}{\Delta t^n}$$
$$\bar{v}^{n+\frac{1}{2}} = \frac{\bar{x}^{n+1} - \bar{x}^n}{\Delta t^{n+\frac{1}{2}}} \quad \text{(EQ 27)}$$

$\bar{a}^n$ has just been calculated (Section A.11.10). Δt^n and $\Delta t^{n+\frac{1}{2}}$ are input to the contact algorithm, as are $\bar{x}^n$ and $\bar{v}^{n-\frac{1}{2}}$. So the new velocity and position are calculated as:

$$\bar{v}^{n+\frac{1}{2}} = \bar{v}^{n-\frac{1}{2}} + \bar{a}^n \Delta t^n$$
$$\bar{x}^{n+1} = \bar{x}^n + v^{n+\frac{1}{2}} \Delta t^{n+\frac{1}{2}} \quad \text{(EQ 28)}$$

A.11.12 Order Node and Order Face Update

After the coordinates of the assigned nodes have been updated, the new order node and order face must be determined for each assigned node. The search is limited by the following constraints:

1. The new order node must share a face with the old order node
2. The new order face must include the new order node as a corner

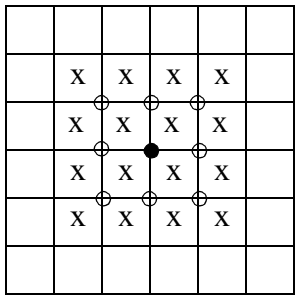


FIGURE A-39

The above figure shows part of the contact side opposite a given assigned node (called the ***active node***). The active node itself is not shown. Assume that the old order node, from before the coordinate update, is marked by the filled-in circle. Then the candidates for the new ordernode are the old ordernode, and any of the nodes marked by open circles. The new order face can therefore be any of the faces marked by "x". However, not all these faces needs to be checked. The new order node is first determined, then only the faces that include the new order node are candidates for order face.

The new order node is simply the candidate with the least euclidean distance to the active node.

Once the order node is determined, all faces that include the order node, as determined by the dual connectivity (Section A.2.2), are tested to determine which is the order face.

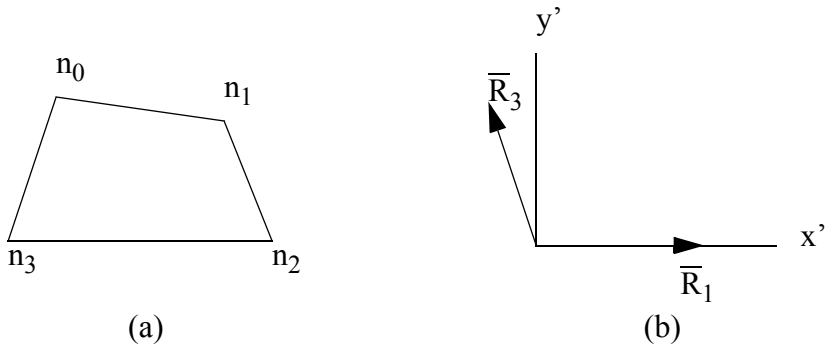


FIGURE A-40

In Figure A-40 (a), the face is one of the candidate faces. n_0 is the new order node. A vertex is formed by the three nodes n_1, n_0, n_3, which also define the plane passing through the three points. To simplify the discussion, consider the local reference frame x'-y' of Figure A-40 (b), in which $\overline{R}_1$ represents the vector from n_0 to n_1, and $\overline{R}_3$ represents the vector from n_0 to n_3. The reference frame is chosen so that $\overline{R}_1$ lies along the positive x' axis, and $\overline{R}_3$ lies in the upper-half plane.

Now consider the projection of the active node onto this plane. Figure A-41 shows this situation for two different cases. $\overline{R}_i$ is the projection of the active node onto the plane formed by $\overline{R}_1$ and $\overline{R}_2$.

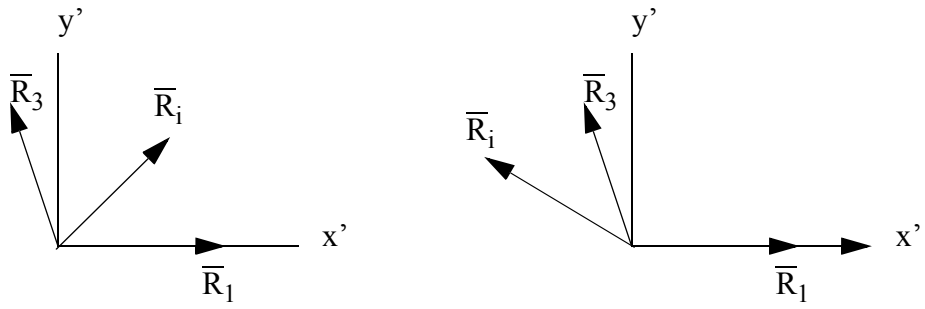


FIGURE A-41

In (a) the face containing the vertex would be accepted as the new order face, while in case (b) it would be rejected. The criterion is whether or not the vector $\overline{R}_i$ lies within the smaller angle formed by $\overline{R}_1$ and $\overline{R}_3$. This can be expressed algebraically by the following two equations:

$$(\overline{R}_1 \times \overline{R}_i) \cdot (\overline{R}_1 \times \overline{R}_3) > 0$$
$$(\overline{R}_i \times \overline{R}_3) \cdot (\overline{R}_1 \times \overline{R}_3) > 0 \qquad \text{(EQ 29)}$$

A.11.13 The Put-Point-On Correction

It was found in practice that as a simulation proceeded, the two sides of a contact surface would gradually separate or interpenetrate. Unlike Lagrange or Penalty methods of contact enforcement, the method described here has no inherent mechanism to compensate for the gradual round-off error, no feedback system based on distance between the sides that will act to keep the sides together. There is only the (imperfect) symmetry of operations. In most physical simulations such small errors are an accepted and largely ignored feature of discretization, but in contact calculations each side provides a fine caliper to measure

the numerical imperfections of the other side: i.e. the sides manifestly do not stay in contact. To remedy this a constraint is explicitly applied, by projecting the slave side onto the master side. For most applications it is sufficient to project the slave node velocities onto the master side: i.e. to match velocities with the image node on the master side. Less frequently it is necessary to project the coordinates themselves onto the master side image nodes. The rationale here is that, at this point the motion of either side is accurate to within the inherent limits of the method, so the master side is arbitrarily chosen as "truth".

A.12 CD<–>CD Communication

A.12.1 Rationale

During the Contact calculations, there are two points where updated nodal quantities are required for ghost nodes. The first point occurs during pressure calculation.

To calculate the force on an assigned node due to pressure, the pressure for all surrounding faces on the same side is needed. This in turn requires the net normal force at each corner of the surrounding faces, which is the average of the normal force at the corner node and the interpolated normal force at the corresponding image node. But to calculate normal force at the image node requires nodal normal forces at each corner of the face containing the image node. And in order to calculate the nodal normal force at any corner node requires the stress on all faces surrounding the node, and the coordinates of all nodes at the corners of those faces.

Other than the initial assigned node, all other nodes in the above description may be ghost. One alternative is to calculate the nodal normal force for ghost nodes out to a radius of 2

from any assigned node, which would require an additional layer of ghosts beyond that, as well as a lot of redundant calculation. A second alternative, the one chosen here, is to implement CD↔CD communication to get the nodal normal forces for ghost nodes from the CDs where they are assigned nodes.

A second CD↔CD communication point occurs after the final accelerations have been calculated. Updated velocities and coordinates of ghost nodes are required in order to accurately determine the new order node (which may be a ghost) and order face, as well as to correctly project the velocity of coordinates of slave nodes onto the moved master surface during the Put-Point-On operation. When the accelerations of the ghost nodes are known, it is simple to integrate (as described in Section A.11.11) to get the correct velocities and positions of ghosts as well as assigned nodes.

In addition to the computational requirements, CD↔CD communication is also used to build the nextTimePDs lists, determining which primary domains the CD will talk to next cycle.

A.12.2 Updating the Ghosts

In Section A.10.6 the CD↔CD invoices were described. Also included in the discussion were a list of the neighbor CDs that have ghost copies of this CD's assigned nodes (the CDSendList), and a list of CDs that have assigned nodes that this CD has ghost copies of (the CDRcvList).

First, the contact domain loops through its neighbor CDs in the CDRcvList, allocates buffers for each, adequate to receive the expected data, and then issues an asynchronous read from the neighbor.

Next it loops through the neighbor CDs in the CDSendList. For each contact domain in the list it allocates a buffer, copies the invoices (master and slave) for that neighbor into the buffer, and then copies the appropriate data for the particular CD↔CD communication pass for each node in the invoice into the buffer. It then sends the buffer to the neighbor CD.

After all the neighbor CDs in the CDSendList have been sent the update data for their ghost nodes, the contact domain waits for the asynchronous reads issued earlier to complete. When they are complete, the contact domain processes each input buffer. It first extracts the sent invoices (in Sidx indices), and converts them to Cidx indices using its Sidx→Cidx map, then extracts the update data in invoice order and uses it to update the appropriate values in the corresponding ghost nodes.

A.12.3 The nextTimePDs List

Another pass of CD↔CD communication is required to help determine which primary domains will send this contact domain data next cycle. This determination is made at the end of the contact calculation, after the new order node and order face have been determined.

Recall from Section A.9.1 that the nodes required on a particular contact domain are determined by the set of generators for that contact domain. To recap, a node is a

generator for the CD if it is assigned to the CD (Rule 1), or if it is the order node of a node assigned to the CD (Rule 2). Given the set of generators for a CD, then the union of the neighborhoods of the generators is the set of nodes required on the contact domain.

To convert this information to the knowledge of which primary domains have these nodes, and hence communicate with the CD, the neighbor PD lists (Section A.4.1.7) of the generators are used. The neighbor PD list of a node is simply the union of the primary domains where the node or any nodes in its neighborhood are local. This list is static for a node, and is sent to the contact domain as part of the nodal data.

Once the set of generators for the contact domain next cycle has been determined, the union of the neighbor PD lists of these generators produces the list of primary domains that the contact domain will talk to next cycle.

The next issue then, is to determine the set of generators. There are three sources for this set:

1. The master nodes assigned statically to the contact domain
2. The set of slave nodes that are order nodes of the assigned master nodes.
3. The set of slave nodes whose order nodes are assigned master nodes

The first group is trivial: it does not change from one cycle to the next, so those nodes (and their neighbor PD lists) are always available. The second group is also simple, because the new order node of an assigned master node must be available on the CD as either an assigned or ghost slave node (this is one basis for the choice of a neighborhood radius). the neighbor PD lists of ghosts as well as assigned nodes are available on the contact domain, so the CD loops through its assigned master nodes, finds their new slave order nodes, and

from there the new order node's neighbor PD list, which it ORs into the nextTimePDs list under construction.

The third group provides the most difficulty. An assigned slave node's order node for the current cycle is guaranteed to also be assigned to the same CD, by the nature of the decomposition; this is how the slave node's CD is assigned. However, after the nodal positions are advanced and the ordering recalculated, the assigned slave node's new order node may be a ghost. Or shifting the perspective to the CD of the new order node, ghost slaves may order on assigned masters, and thus be generators for the CD next cycle. But the CD doesn't know the new order nodes of ghost slaves; that is computed where the slave is an assigned node. Thus the need for CD↔CD communication.

To process the third group of generators, each CD loops through its list of assigned slave nodes. For each, it finds its order node. If the order node is an assigned master node, it ORs the order node's neighbor PD entries into the nextTimePDs list under construction. If, however, the order node is a ghost, it finds the CD where the ghost is assigned, and ORs the order node's neighbor PD entries into an accumulating list for that particular CD.

When all the assigned slave nodes have been processed in this fashion, the accumulated list of neighbor PDs for each neighbor CD is sent to the CD, where it is OR'ed into that CDs nextTimePDs list.

A.13 CD -->PD Communication

At the finish of the contact calculation the updated data is returned to the primary domains. Again, as in PD→CD communication, the msgBases and their associated invoices are used to guide the transfer.

A.13.1 Packing the Results

Recall that when a primary domain sends nodes to a contact domain, the primary domain allocates a msgBase for that particular transaction, and creates invoices of the nodes sent. The Sidx version of the invoice is sent along with the data. On the receiving end the Sidx invoices are converted to Cidx invoices, which are stored in the receiving msgBase for that primary domain.

In returning the data, these Cidx invoices are used to determine what to send back to the primary domain. As a Cidx invoice is traversed the local index is used to retrieve the various results, specifically new ordernode, orderface, position and velocity, which are then copied into the outgoing int and float buffers for that primary domain. Only the assigned node invoices, master and slave, are traversed since updated results are only calculated and returned for the assigned nodes.

There is one exception to this. If a slave node, assigned or ghost, is the ordernode of one or more master nodes assigned to this contact domain, that slave node is flagged. A slave node thus flagged is a Rule 2 generator (Section A.9.1) for this contact domain next cycle. The status, flagged or unflagged, for each assigned and ghost slave node is returned in the int buffer, in invoice order.

It should be noted that, for nodes along the boundaries of PDs, which are shared by multiple PDs, each such primary domain redundantly sent the node to the contact domain. The redundancy was removed there, and the updated nodal values calculated once. But the node still appears in the invoice of each corresponding CD msgBase. Thus when the results are packed into the outgoing buffers for return to the primary domains, the results for these shared nodes will be copied into each corresponding buffer, and all contributing primary domains will receive the updated nodal data for the shared node

A.13.2 Sending and Receiving

After the int and float buffers for each PD that communicates with this CD this cycle are complete, the buffers are sent back to the originating PDs. This step is similar to the PD→CD communication (Section A.9.5), except that no preliminary sends of buffer lengths is required, since the originating PDs know how much data to expect back.

A.13.3 Unpacking the Results

When the buffers from the various contact domains that communicate with this primary domain are all received, the buffers are unpacked into primary domain and bulk domain structures. The updated coordinates and velocities of all nodes are copied over the corresponding pre-contact data in the bulk domain list of nodal coordinates and velocities. When the invoices were originally constructed (Section A.9.3) a Bidx version of each invoice was built. The data returned from the CD is in invoice order, so the Bidx invoice can now be used to store the returned data at the correct index in the bulk domain.

From the integer buffer the primary domain obtains the updated ordernode and orderface for each node it sent to the CD. These are copied into the corresponding fields of the nodeStuff structure (Section A.4.1) using the Pidx invoice to select the correct nodeStuff structure. If a node is a slave, then the statically assigned CD of its new master ordernode is used to determine which CD the slave will be assigned to next cycle (its rule 1 CD). Also if a slave node has been tagged as being the ordernode of one or more assigned master nodes of the CD which returned this buffer, then the CD is added to the Rule 2 CD list in the node's nodeStuff struct, which will make this node a rule 2 generator for the CD next cycle (Section A.9.1)

With this, the contact work for this cycle is complete.

APPENDIX B ALE3D EXTENSIONS

The development described in the previous chapters is a simplification of the software actually incorporated in ALE3D. This was done to avoid hiding the essential issues in a deluge of details. In this appendix some of the real-world complications of the production implementation will be discussed, though at a higher level than the previous discussion.

B.1 Multiple domains per Processor

ALE3D is designed to support ***domain overloading***, or the ability to support multiple bulk domains on each processor. This has several advantages. First, it decouples the problem generation, specifically the domain decomposition, from the number of processors actually used. Thus a problem decomposed into 1000 domains may subsequently be run on 50, 100, 1000, or any other available number of processors (not necessarily a divisor of 1000). This provides flexibility to adapt to a dynamic machine load without regeneration.

Second, it enables a large-grained style of load-balance. If there are many more domains than processors, then if the computational requirements for the various domains change and the problem becomes unbalanced, there is the possibility of regrouping the domains

assigned to each processor to restore the load balance. This potential is not currently exercised in ALE3D.

Third, domain sizes can be chosen small enough so that the significant data for a domain can fit into the processor's memory cache, leading to improved single-processor performance.

Finally, the domain level provides a large-grained level of parallelism which can conveniently be taken advantage of in the prevalent "network of SMP's" model of supercomputer. A single node of such a machine consists of multiple processors. Multiple domains are assigned to a node, and the code is then multi-threaded at the domain level: Each processor of the node takes a different iteration in any loop over domains.

Balanced against these advantages is the drawback of greater memory requirements. This is due to to the need for each domain to have available not only its local data, but a copy of "ghost" data for nodes and zones in the neighborhood of its borders. This leads to data redundancy that increases as the number of domains per node increases.

Domain overloading add complexity to the contact implementation primarily in the communication handling. To minimize the number of required messages, all communication is processor-to-processor, rather than domain-to-domain. Thus the software must package data from the different local domains together, and must keep track of the processor-domain hierarchy. For example, in PD↔PD communication, as described in Section A.8, invoices are kept for what to send to each neighbor processor, defined as a processor one or more of whose domains is a neighbor to one or more domains of this processor.

The invoice structure actually contains a list of substructs, one for each local domain. Each substruct contains a number of invoice lists, one for each domain in the neighbor processor.

Similarly, in PD↔CD communication, the msgBase structures described in Section A.9.4 actually are defined at the processor level, and contain substructs which in turn contain the invoices of nodes and faces from some particular domain of the processor.

B.2 Multiple Slide Interfaces

The description in Chapters 3 and 4 assumed a single slide interface. In ALE3D an arbitrary number of slide interfaces may be defined. Each slide is treated independently of the others (except for overlapping or intersecting slides, described in Section B.4 and Section B.5). However, the slide decomposition may assign portions of multiple slides to a single processor, so the data structures and code in the contact domain must support this multiplicity. In particular the msgBase struct hierarchy is again deepened so that there are separate substructs for each slide surface involved. For each of these there are separate substructs for each domain, as described in Section B.1. Also, the processor handling multiple slides builds a separate contact domain, as described in Section A.10, for each of them.

B.3 Void Opening and Closing

ALE3D supports the ability for slide interfaces to initially be separated and to close at some dynamically determined time, conserving total momentum in the process. Conversely, if a tensile force develops between the two sides of a slide interface, ALE3D

allows the two sides to separate. This makes events such as projectile-target impact possible to simulate.

Even when the two sides are separated, the contact logic continues to calculate the closest node to a given node on the other side each cycle. In fact without this knowledge it would be impossible to determine when the void has closed. But beyond this, it also allows the local search algorithm used in updating the order node (Section A.11.12) to suffice as closure is approached, rather than requiring a global search for the nearest node. As the two sides approach, the order node of a given node will in general change, but it will change in an orderly fashion, moving from a previous order node to one of its neighbors in terms of connectivity, due to continuity of motion. If this is not true then the timestep is too large, just as can occur when the surfaces are in contact.

B.4 Overlapping Slides

In some cases a node can belong to more than one slide surface. For instance, consider Figure B-1:

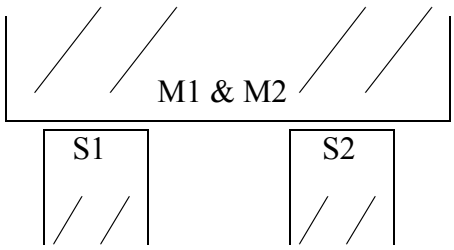


FIGURE B-1

Nodes along the top surface are defined to be on the master sides of both surface 1 and surface 2, since at different times any node on the top surface may be in contact with either of the slave sides S1 or S2. This is handled in ALE3D by redundantly calculating the

effect on such a node due to each surface independently, in different contact domains. At the end of the contact calculations the results are returned to the home bulk domain of the node. If a node belongs to an overlapped surface, then rather than updating the coordinates and velocities immediately, the results are stored in a temporary holding area until they are available from each overlapped slide. Then a decision is made independently for each overlap node. If a node is clearly associated with one surface or the other, that result is used. This is dictated by the ***extension flag***. If a node is "off the edge" of its opposing surface by more than a fraction of the size of its order face, then it is "on the extension" and its extension flag is set. This is done in the contact domain, and the extension flag is returned to the bulk domain. For an overlap node if its extension flag is set for one surface and not the other, then the results from the surface where it is not set are used.

If, however, neither extension flag is set, then the choice is arbitrated by a user-specified preference, which is statically set for each pair of overlap surfaces.

B.5 Intersecting Surfaces

In ALE3D it is possible for two or more slide surfaces to intersect. There are two possible geometries of intersection. The first, called a corner intersection, occurs when the surfaces meet at an angle as in Figure B-2.

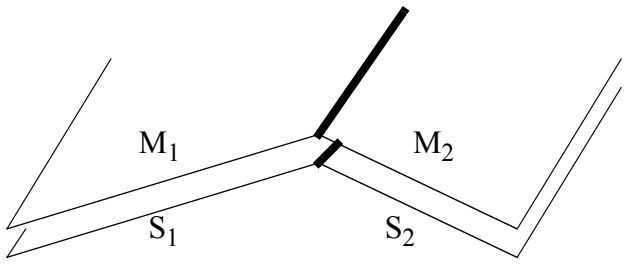


FIGURE B-2

Nodes that lie along the bold lines are intersection nodes. They are on surface 1 and 2 simultaneously. Unlike overlap nodes, where the force on the node is calculated independently for the two surfaces, and then only one result is used, for intersection nodes the force from the two opposing surfaces are summed. Furthermore the put-point-on operation (Section A.11.13) is performed sequentially on the two surfaces, and unless the surfaces are at right angles, this introduces an order dependency.

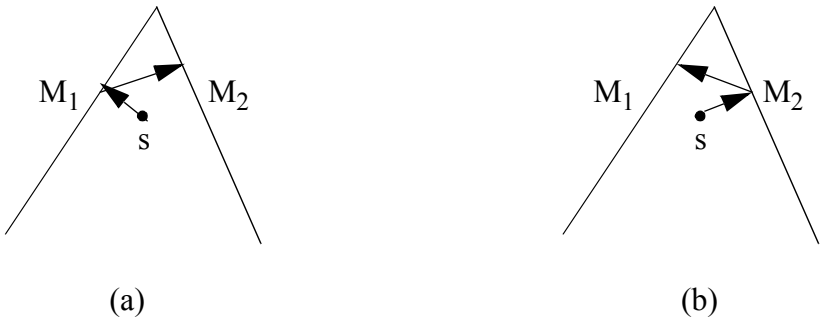


FIGURE B-3

In Figure B-3 (a), the slave node ***s*** is put-point-on onto master surface M_1, then onto M_2. Figure B-3 the order is reversed. Note that the slave node ***s*** does not end up in the same place under both operations.

A node may also lie on the intersection of 3 surfaces, as for instance the corner or a box.

The second type of intersection is called a "T" intersection. Consider Figure :

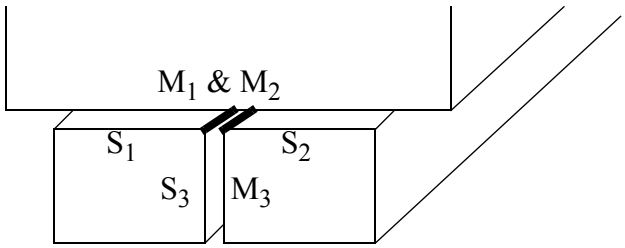


FIGURE B-4

Nodes along the bold lines are the intersection nodes. As with the corner intersection, the intersection nodes of a "T" intersection experience force from both slide surfaces, M_1 and M_3 or M_2 and S_3. Note that T intersections also involve overlapping surfaces (M_1 and M_2 in this case).

Intersecting surfaces add considerable complexity to the code. Because the forces from each surface are summed, and because the put-point-on is done serially, the contact calculation for the node for each surface it belongs to must be done on the same contact processor. But an intersection node may be a master node on one surface and a slave node on a different surface. As a master node it is assigned by the contact decomposition statically to one processor, but as a slave node its domain assignment may move around as its order node changes. Resolving this conflict is a delicate matter best understood through perusal of the source code.

B.6 Advection and Thermal Phases

In addition to the principal contact phase which is part of the Lagrange calculation, there are two other places (at present) in ALE3D where the contact decomposition is used. In both cases, the same machinery described in Chapter 5 is used to create the contact domains, but the actual contact calculation, described in Section A.11, is replaced by calculations appropriate to that phase.

During the advection, or remap, phase the zonal positions for advecting materials may be changed. For zones bordering a contact surface, this is implemented in such a way that the contact nodes tend to be pulled away from the opposing surface. At present, at most one

side of a contact surface may advect. To restore the shape of the surface, a put-point-on operation (Section A.11.13) is performed: i.e. the advecting surface is mapped back onto the non-advecting surface. This operation can only be done in the contact domain, since it is only there that corresponding patches of the two sides are available on the same processor.

Similarly, during the thermal transfer across contact surfaces, corresponding information from both sides of the surface must be available locally, and the contact domain mechanism is invoked to create this environment.

B.7 ALE (Coalescing) Slides

During the evolution of a slide interface, the surface may become so convoluted that it no longer makes sense to represent it as a surface; it is more accurate to represent it as a thin region of mixed cells transitioning from one side to the other. At this point the slide surface, or some subset of it, may be "deleted" in ALE3D. This requires that a least one side of the surface be an advecting surface, and that initially and during the remap each cycle, the nodes of the advecting side are made to match exactly the coordinates of corresponding nodes on the opposing side.

Given these conditions, when it is time to delete all or part of the surface, corresponding nodes and faces on the two sides are merged into one, and the surface, as two separate sides, ceases to exist. After this time, advection across what was previously the slide interface is possible, and the problem of surface convolution is "smoothed" by the mechanism of mixed zones.

BIBLIOGRAPHY

[1] David J. Benson, "Computational Methods in Lagrangian and Eulerian Hydrocodes", *Computer Methods in Applied Mechanics and Engineering 99 (1992)*

[2] Charles E. Anderson, Jr., "An Overview of the Theory of Hydrocodes", *International Journal of Impact Engineering, vol 5 (1987)*

[3] G. Maenchen and S. Sack, "The TENSOR Code", *Methods of Computational Physics, Vol 3 (1964)*

[4] Mark L. Wilkins, "Calculation of Elastic-Plastic Flow", *Lawrence Livermore National Laboratory UCRL-7322, rev 1 (1969)*

[5] N. E. Hoskins "Solution by Characteristics of the Equations of One-Dimensional Unsteady Flow", *Methods of Computational Physics, Vol. 3 (1964)*

[6] J. von Neumann and R. D. Richtmyer, "A Method for the Numerical Calculation of Hydrodynamical Shocks", *Journal of Applied Physics, vol 21 (1950)*

[7] W. E. Johnson, "OIL, A Continuous Two-Dimensional Eulerian Hydrodynamic Code", *GAMD-5580, General Atomic, San Diego, Ca (1965)*

[8] W. E. Johnson and C. E. Anderson, Jr, "History an Application of Hydrocodes in Hypervelocity Impact", *International Journal of Impact Engineering, vol 5 (1987)*

[9] R. Chiapetta, T. Belytschko, and J. Rouse, "A Computer Code for Dynamic Stress Analysis of Media-Structure Problem with Non-Linearities (SAMSON)", *Report AFWL-TR-72-104, Air Force Weapons Laboratory, Kirkland AFB, NM (1973)*

[10] T. Belytschko, "Computer Methods in Shock and Wave Propagation Analysis", *Computing in Applied Mathematics, AMD, Vol. 18, American Society of Mechanical Engineers (1976)*

[11] S. W. Key, "A Finite Element Procedure for Large Deformation Dynamic Response of Axisymmetric Solids", *Computer Methods of Applied Mechanics and Engineering, Vol. 4 (1974)*

[12] J. O. Hallquist, "A Procedure for the Solution of Finte-Deformation Contact-Impact Problems by the Finite Element Method", *Report UCRL-62066, Lawrence Livermore National Laboratory, Livermore, Ca (1976)*

[13] G. R. Johnson, "Analysis of Elastic-Plastic Impact Involving Severe Distortions", *Journal of Applied Mechanics, Vol. 43, No. 3 (1976)*

[14] Ronald L. Panton, "Incompressible Flow", *John Wiley and Sons, Inc. (1996)*

[15] W. F. Noh, "CEL: A Time-Dependent Two-Space-Dimensional, Coupled Eulerian-Lagrange Code", *Methods in Computational Physics, Vol 3 (1964)*

[16] C. H. Hirsch, "Numerical Computation of Internal and External Flows: Vol 1", *John Wiley and Sons, Inc. (1988)*

[17] Gilbert Strang, "Introduction to Applied Mathematics", *Wellesley-Cambridge Press (1986)*

[18] W. Pohl et. al., "The PISCES Software for Defense: A survey of PISCES Code Applications for Defense-Oriented Computational Problems", *PISCES International, Rev 2 (1984)*

[19] S. Hancock, "PISCES 2DELK Theoretical Manual", *Physics International (1985)*

[20] D. A. Matuska and J. Osborn, "HULL Documentation: Vol 2, User's Manual", *Orlando Technologies, Inc Technical Report (1986)*

[21] P. Yarrington, "The CTH-EPIC Link - A Coupled Eulerian/Lagrangian Computational Method", *Proceedings of the 12th U.S. National Congress on Computational Mechanics (1994)*

[22] David J. Benson, "An Efficient, Accurate, Simple ALE Method for Nonlinear Finite Element Programs", *Computer Methods in Applied Mechanics and Engineering 72 (1989)*

[23] R. Tipton, "CALE User's Manual", *Lawrence Livermore National Laboratory (1992)*

[24] F. L. Addesio et. al., "CAVEAT: A Computer Code for Fluid Dynamics Problems with Large Distortion and Internal Slip", *Los Alamos National Laboratory LA-10613-MS-REVISED (1990)*

[25] J. S. Peery, K. G. Budge, M. K. Wong, and T. G. Trucano, "RHALE: a 3-D MMALE Code for Unstructured Grids", *ASME Winter Annual Meeting (1993)*

[26] E. Dube et. al., "ALE3D User's Manual", *Lawrence Livermore National Laboratory (1999)*

[27] N. Kikuchi, J. T. Oden, "Contact Problems in Elasticity: A Study of Variational Inequalities and Finite Element Methods for a Class of Contact Problems in Elasticity", *SIAM Studies Vol. 8 (1986)*

[28] H. Hertz , "Gesammelte Werke", *Vol. 1, Leipzig (1895)*

[29] A. Signorini "Soptr Akune Questioni di Elastostatica", *Atti della Societa Italiana per il Progresso delle Scienze (1933)*

[30] George Karypis and Vipin Kumar, "METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 1.0", *University of Minnesota, Department of Computer Science (1995)*

[31] B. Hendrickson and R. Leland, "The CHACO User's Guide: Version 2.0", *Sandia National Labs, Albuquerque, NM (1995)*

[32] James G. Malone and Nancy L. Johnson, "A Parallel Finite Element Contact/ Impact Algorithm for Non-Linear Explicit Transient Analysis: Part 1 - The Search algorithm and Contact Mechanics", *International Journal for Numerical Methods in Engineering, Vol 37 (1994)*

[33] James G. Malone and Nancy L. Johnson, "A Parallel Finite Element Contact/ Impact Algorithm for Non-Linear Explicit Transient Analysis: Part 2 - Parallel Implementation", *International Journal for Numerical Methods in Engineering, Vol 37 (1994)*

[34] J. O. Hallquist, G. L. Goudreau, and D. J. Benson, "Sliding Interfaces with Contact-Impact in Large-Scale Lagrangian Computations", *Computer Methods in Applied Mechanics and Engineering 51 (1985)*

[35] Davis J. Benson and John O. Hallquist, "A Single Surface Contact Algorithm for the Post-Buckling Analysis of Shell Structures", *Computer Methods in Applied Mechanics and Engineering 78 (1990)*

[36] C. G. Hoover, A. J. DeGroot, J. D. Maltby and R. D. Procassini, "ParaDyn: DYNA3D for Massively Parallel Computers", *Presentation at Tri-Laboratory Engineering Conference on Computational Modelling (Oct 1995)*

[37] L. M. Taylor and D. P. Flanagan, "PRONTO3D - A Three Dimensional Transient Solid Dynamics Program", *Sandia National Laboratories Report SAND87-1912 (1989)*

[38] S. W. Attaway et. al., "A Parallel Contact Detection Algorithm for Transient Solid Dynamics Simulations Using PRONTO3D", *Computational Mechanics 22 (1998)*

[39] Kevin Brown, Steve Attaway, Steve Plimpton and Bruce Hendrickson, "Parallel Strategies for Crash and Impact Simulations", *Computer Methods in Applied Mechanics and Engineering 184 (2000)*

[40] R. Couch and D. Faux, "Simulation of Underwater Explosion Benchmark Experiments with ALE3D", *Lawrence Livermore National Laboratory Report UCRL-CR-123819 (1996)*

[41] R. Courant, K. O. Friedrichs, and H. Lewy, "Uber die partiellen differenz-gleichungender mathematischen Physik", *Mathematische Annalen 100 (1928).* English translation in *IBM Journal (1967)*

[42] L. Stanberry, Personal communication, *Lawrence Livermore National Laboratory, March 2002*