To be presented at "Symposium on Development, Validation, and Application of Inelastic Methods for Structural Analysis and Design", 1996 ASME International Mechanical Engineering Congress an Exposition (IMECE96), Atlanta, GA, November 17-22, 1996.

PARALLEL CONTACT DETECTION ALGORITHM FOR TRANSIENT SOLID DYNAMICS SIMULATIONS USING PRONTO3D

Stephen W. Attaway, Bruce A. Hendrickson, Steven J. Plimpton, David R. Gardner, Courtenay T. Vaughan, Martin W. Heinstein, James S. Peery

Sandia National Laboratories
Albuquerque, New Mexico

ABSTRACT

An efficient, scalable, parallel algorithm for treating material surface contacts in solid mechanics finite element programs has been implemented in a modular way for MIMD parallel computers. The serial contact detection algorithm that was developed previously for the transient dynamics finite element code PRONTO3D has been extended for use in parallel computation by devising a dynamic (adaptive) processor load balancing scheme. [1]

INTRODUCTION

This paper presents a review of inelastic analyses at Sandia National Laboratories with a focus on the development, testing and implementation of a parallel contact detection algorithm for solid mechanics program PRONTO3D. We have accomplished the important step of devising a dynamic load balancing scheme for contact detection, which is necessary for efficient parallel computation by the finite element method with material contacts.

Sandia's solid dynamics code PRONTO3D uses Lagrangian finite elements to model a wide variety of problems, such as the calculation of impact damage to shipping containers for nuclear waste and the analysis of vehicular crashes. Using parallel computers for these simulations has been hindered by the difficulty of searching efficiently for material surface contacts in parallel. A new parallel algorithm for calculation of arbitrary material contacts in finite element simulations has been developed and implemented in the PRONTO3D transient solid dynamics code. This paper presents some of the issues involved in developing, testing, and applying a parallel finite element code.

In order to implement a finite element calculation efficiently, the mesh is partitioned among the processors so as to minimize the need for inter-processor communications and evenly distributes the computational load. Optimal mesh partitioning for the finite element portion of the calculation is not difficult to achieve since each processor must communicate only with the few connected neighboring processors that share boundaries with the decomposed mesh. However, contacts can occur between surfaces that may be owned by any two arbitrary processors. Hence, a global search across all processors is required at every time step to search for these contacts.

Here, we discuss the details of a new parallel contact detection algorithm. We also present timing and scalability results for some large simulations that illustrate how the new contact-detection algorithm has enabled efficient parallel implementation of PRONTO3D. Example simulations involving up to two million elements are presented to demonstrate that the new algorithm is scalable in practice to over 1000 processors of the Intel Paragon.

The structure of this paper is as follows: We review the capabilities of PRONTO3D, then discuss the motivation for implementing it for parallel computations. In preparation for describing the parallel contact algorithm, we review the current serial contact algorithm and some background material needed to understand aspects of the parallel algorithm. A description of the parallel contact algorithm is followed by some examples that illustrate the performance of the parallel algorithm.

TRANSIENT DYNAMICS CAPABILITES OF PRONTO3D

Sandia's PRONTO3D code (Taylor and Flanagan, 1989), (Attaway, 1990), (Bergmann, 1991) is a Lagrangian finite element program for the analysis of the three-dimensional response of sol-

1. This work performed at Sandia National Laboratories supported by the U. S. Department of Energy under contract DE-AC0476DP00789.

id bodies subjected to transient dynamic loading. The program includes nonlinear constitutive models and accurately analyzes large deformations that may lead to geometric nonlinearities. PRONTO is a powerful tool for analyzing a wide variety of problems, including classes of problems in:

- impact dynamics,
- rock blasting, and
- accident analyses.

PRONTO3D is a direct descendant of the PRONTO2D code (Taylor and Flanagan, 1988). Experienced users will recognize the similarity in the structure between PRONTO2D and PRONTO3D, since the theory and algorithms are the same in both codes.

The development of PRONTO was motivated by the need for a code that could serve as a test bed for research into numerical algorithms and new constitutive models for nonlinear materials. Towards this goal, the code contains a well-documented and easy-to-use interface for implementing new constitutive models. In addition, a flexible, problem-oriented language has been developed for the input to PRONTO that allows the user to define a complex mechanics problems with a few concise commands. Internally, the code is well documented and consistent between subroutines. The algorithms within PRONTO3D were designed to be accurate, dependable, and to execute rapidly.

PRONTO contains no mesh generation or post-processing capabilities. Instead, it relies on external mesh generators and external post-processors. There are few references in the user input to finite element node or element numbers. The philosophy has been to define the problem geometry through the GENESIS mesh definition data base (Taylor, Flanagan and Mills-Curran, 1986). The output from PRONTO is accessed through the EXODUS data base. (Schoof and Yarberry, 1994). PRONTO is part of the SEACAS family of codes. An overview of some of the tools available in the SEACAS system is found in Sjaardema (1993).

The following is a summary of the technology within PRONTO3D:

- Explicit mid-point time integration
- Adaptive time step control
- Eight node brick elements and four node shell elements
- One point element integration
- Flanagan-Belytschko Hourglass control for hex elements
- Assumed Strain hourglass control for hexahedral elements
- Objective material coordinate system
- Global contact (self contact) with erosion of surfaces.

WHY PARALLEL?

A complicated process such as a collision or explosion involving numerous complex objects requires a large number of mesh elements to model accurately. In addition, the underlying physics of the stress-strain relations for a variety of interacting materials must also be included in the model. Running such a simulation for thousands or millions of time steps can be very computationally intensive.

In the past, increases in speed were obtained by building faster processors. However, the clock speed of a single processor computer is controlled by the size of the objects on the chip. Current manufacturing techniques limit the size of the chip. Because of this size limit, we can no longer expect to see large increases in speed by simply building processors with faster clock cycles.

High computational speed has been obtained through vectorization on supercomputers such as the Cray Y-MP. Multitasking on the Y-MP allows additional speed by breaking the vectorized loops into tasks that can be worked on by more than one processor. While this approach works very well for small numbers of processors, it does not scale well to large numbers of processors.

Massively parallel machines, such as the Intel Paragon, which have very large numbers of inexpensive processors, offer an alternative to the vectorized, multitasking machines. This massively parallel computer architecture uses multiple processors that pass messages between each other to achieve faster performances. Current plans for large scale parallel machines include processors with speeds of 200 MIPS and over 4000 processors.

The problem with the message-passing parallel architecture is the complexity of the code required to obtain a computationally balanced algorithm. Since finite element calculations require synchronization at each time step, the slowest processor will control the total problem solution time. Thus, each processor must be given the same amount of work.

Measures of Parallel Computer Performance

Here, we will review some important measures of performance. The grind time, fixed-size speedup, scaled speedup and parallel efficiency will be outlined.

The grind time is an often used measure of performance for a finite element program. The grind time, T_{grind}, is defined as

$$T_{grind} = \frac{T_{execution}}{N_{elements} N_{cycles}} \tag{1}$$

where $T_{execution}$ is the execution time of a mesh with $N_{elements}$ for N_{cycles}.

Two measures of parallel performance are the fixed-size speedup and the scaled speedup. The fixed-size speedup, S_f, is defined as:

$$S_f = \frac{T_{m,1}}{T_{m,p}} \tag{2}$$

where $T_{m,1}$ is the execution time for a problem with m elements executed on one processor and $T_{m,p}$ is the execution time for the same problem size m run on p processors.

The scaled speed-up is used to describe problems where the problem size increases in proportion to the number of processors and is defined as:

$$S_s = \frac{T_{m,1} p}{T_{m \times p,\, p}} \tag{3}$$

where $T_{m \times p,\, p}$ is the execution time for a problem of size $(m \times p)$ run on p processors.

The parallel efficiency is defined as

$$E = \frac{S}{p} \tag{4}$$

The scaled parallel efficiency indicates the efficiency of the parallel algorithm when the problem size increases in proportion to the number of processors.

Typically, the fixed-size parallel efficiency, $E_f = S_f/p$, will decrease for fixed-size problems when the number of processors increases. As the problem is divided between more and more processors, the amount of computation each processor has to do relative to the amount of communications becomes small. Eventually, the communications will dominate the calculation time. Thus, for small problems, there is a limit to how fast one can make parallel calculations run.

If the size of the problem increases as the number of processors increases, and if the problem scales well (e.g. $0.95 < E < 1$), then the ratio of communications to computation will stay constant. In order for a code to scale, each processor must be given an equal amount of work to do. In addition, the number and size of the interprocessor communications must not grow excessively as the problem size grows. Thus, the big question is: will a calculation scale?

SERIAL CONTACT ALGORITHM

In this section, we will outline the serial contact algorithm used in PRONTO3D. For a detailed account on the general formulation and numerical treatment of finite deformation contact problems in finite elements see Laursen and Simo (1991).

In explicit time integration algorithms, contacts are usually processed with a predictor/corrector method. Within a time step, the positions of nodes in the mesh are predicted assuming that no contacts occur. A check for overlap and penetration is made, and these overlaps are corrected by applying a contact force between the contact surfaces.

An outline of the computations performed every time step in the explicit time step algorithm in PRONTO3D is shown in Figure 1.The contact algorithm is composed of steps (1.7)-(1.9). We define a set of nodes on the exterior of the mesh called *contact nodes* and a set of surface patches on the exterior element faces called *contact surfaces*. For efficiency, the contact constraint is enforced only at the contact nodes. Thus, each contact node is checked to see if it penetrated a contact surface.

It is convenient to separate contact algorithms into a location phase and a restoration phase. The location phase consists of a neighborhood identification, step (1.7), followed by a detailed contact check. The neighborhood identification associates a contact node to a set of contact surfaces that it potentially could contact.

The neighborhood identification step requires a global search of the simulation domain which can require 30-50% of the overall run time when PRONTO3D runs on a vector machine like the Cray Y-MP. In principle, any two surface elements anywhere in the simulation domain can come in contact with each other during a given time step. This is true even for surface elements on the same object, as when a car fender is crumpled in a collision.

(1.1) Define/redefine contact surface.

(1.2) Integrate the equations of motion.

$$v^{n+1/2} = v^{n-1/2} + \Delta t \frac{f^n}{m} \tag{5}$$

$$x^n = x^{n-1} + \Delta t v^{n+1/2} \tag{6}$$

(1.3) Compute the strain rate based on the motion of the nodes.

$$\dot{\varepsilon} = f(v) \tag{7}$$

(1.4) Compute the stress rate based on the strain rate and the material models.

$$\dot{\sigma} = g(\sigma, \dot{\varepsilon}) \tag{8}$$

(1.5) Compute internal forces acting on nodes:

$$f_{int} = \nabla \bullet \sigma \tag{9}$$

(1.6) Predict the locations of the nodes assuming no contacts

$$\hat{a} = \frac{f_{int}}{m} \tag{10}$$

$$\hat{v} = v + \Delta t \hat{a} \tag{11}$$

$$\hat{x} = x + \Delta t \hat{v} \tag{12}$$

(1.7) Search for potential contacts between nodes and surfaces

(1.8) Perform detailed contact check to determine "best" contact surfaces.

(1.9) Enforce contacts by computing contact force required to remove overlap

$$f = f_{int} + f_{contact} \tag{13}$$

FIGURE 1.Outline of explicit time step algorithm.

The detailed contact check in step (1.8) determines which, if any, of the candidate contact surfaces is in contact with a contact node. It also determines the point of contact, the amount of penetration, and the direction that the contact node must be moved in order to remove the penetration. The detailed contact check is accomplished by monitoring the displacements of the contact nodes throughout a time step for possible penetration of a contact surface. This stage of the contact algorithm is logically complex but computationally inexpensive. The complexity arises from having to consider multiple contact surfaces as potential candidates for contact (Heinstein, et al.,1993).

The contacts are enforced in step (1.9). This step defines a contact constraint so that the contact node can be "pushed back" to remain on the contact surface. This constraint is enforced in the

following time step, or possibly over several time steps.

Outlined below are the current characteristics of the serial contact algorithm that we feel the parallel contact algorithm should share. The algorithm will:

i) use a fast, memory-efficient global search to decide which contact nodes are in proximity to an element contact surface;
ii) perform a detailed contact check using projected movements of both the contact surface and contact node to determine the location, magnitude, and direction of contact node penetration of the contact surface;
iii) automatically define all surfaces given the mesh connectivity;
iv) use an accurate method for enforcing the contact constraint.
v) model self-contacting surfaces;
vi) model tearing and eroding surfaces; and
vii) have simple user input.

In the next section, we will describe in more detail the methods used in the serial contact algorithm.

Automatic Identification of Contact Surfaces

The surface definition phase of the contact algorithm automatically determines which surfaces are interior and which are exterior. The exterior surfaces can be included in the contact search on a material-by-material basis. This flexibility allows the user to select a subset of the problem to be included in the contact search.

There are several algorithms available for determining the exterior of a surface (Whirley and Engelman, 1994) (Belytschko and Neal, 1989).The algorithm described here follows Heinstein, et al. (1993).

The surface definition algorithm uses a data structure that generates the initial surface definition and allows an incremental update of surface when necessary. In the algorithm, shell elements are considered as a subset of hexahedral elements. For clarity, the following discussion is limited to a mesh composed of hexahedral elements. The algorithm stores a 6 x n_e list of faces ID's for a mesh composed of n_e 8-node hexahedral elements and involves two simple steps. The first step is to build a list of face ID's for every element face:

$$faceid = n_{diag} + n_{min} \times n_{nod} \tag{14}$$

where n_{nod} is the total number of nodes in the problem, n_{min} is the smallest global node number defining the element face, and n_{diag} is the global node number that is diagonal to the smallest global node n_{min}.

In the second step, the element faces on exterior surfaces are found by locating all non-repeated face ID's. The contact nodes can then be determined by looping through the contact surface list and flagging the nodal points defined by the contact surface connectivity.

By storing a list of opposing faces for each element, an incremental update of the contact surfaces can be performed when an element is deleted. Once the surface map is complete, all the contact surfaces and contact nodes can be collected into a heap for sorting and processing. The idea of collecting contact surfaces and contact nodes into a heap allows modeling of contacts between a variety of finite element types. For example, the nodal points of elements, such as beams and trusses, can be added to the contact node list. Also, the potential contacts from coupling the finite element method with other methods can be modeled. For example, particle methods such as Smooth Particle Hydrodynamics (SPH) (Attaway, Heinstein and Swegle, 1994) can be easily coupled by adding the particles to the contact node list.

Location Phase

The location phase is the most time consuming part of the contact detection algorithm. Here, we will briefly review the algorithm used in the location phase. A more detailed description of efficient schemes for spatially sorting and searching for contacts can be found in the report by Heinstein, et al., (1993).

The most robust approach for finding potential contacts would be to check every contact node against every contact surface each time step. This exhaustive global searching approach requires nodal distance calculations on the order n_c^2, where n_c is the number of contact nodes.

The serial contact detection algorithm speeds up the location phase by using a global search algorithm based on a bounding box. This process involves bounding the space occupied by a contact surface (element face) at its known location and at its predicted location. Figure 2 shows a bounding box for a moving contact surface over one time step. The location of a contact surface at time *n* and *n+1* is shown in the figure. To insure that all potential contact nodes are within the bounding box, the dimensions of the box are expanded by the maximum amount that a contact node can move during a time step, $d_{max} = V_{max} dt$. Any contact nodes inside the capture box should be considered for potential contact with the contact surface.

To efficiently determine the contact nodes that fall within this capture box, the serial contact algorithm uses the *point-in-box search* algorithm developed by Swegle and coworkers (Swegle, et al., 1993). The algorithm consists of three individual one-dimensional sorts of the nodal locations, using each coordinate value as the sort key. Binary searches of each sorted dimension are used to determine the contact nodes that are closest to the maximum and minimum dimensions of the bounding box. Three lists of nodes whose positions fall within the bounds of the box for each coordinate direction are constructed using the results of the binary searches. The final list of points within the box is the intersection of the three one dimensional lists.

The great advantage of using the *point-in-box* global search algorithm is that it requires only $7n_c$ memory locations. In addition, the algorithm's execution time is proportional to $n_c \log n_c$ and is independent of any problem geometry.

While it may not be necessary to perform the global search every time step, we do so in the examples presented in this report. Thus, the algorithms presented here could be made to run much faster if memory were sacrificed to store the nearest neighbors between time steps.

The *point-in-box search* is also used in the parallel algorithm for searching for contacts on a given processor.

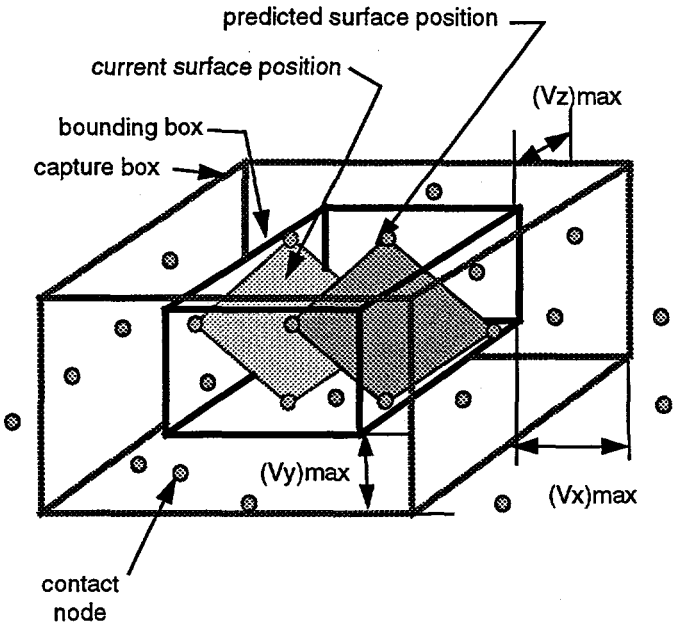

FIGURE 2.Contact search bounding box.

<u>Detailed contact check</u>

After gathering a list of potential interactions a detailed contact check is done for each contact node - contact surface pair. The detailed contact check determines: (1) which of the candidate contact surfaces, if any, is in contact with the contact node, (2) the point of contact, (3) the amount of penetration, and (4) the direction the contact node should be moved to enforce the contact constraint.

The difficulties associated with detailed contact determination included inaccurate push back direction and multiply defined potential contacts. In order to minimize ambiguities, the detailed contact check distinguishes between contact nodes that are not in contact and those that are already in contact.

A velocity based contact check is used for the contact nodes not already in contact. For contact nodes just coming into contact, the velocity based contact check identifies the point of contact (or impact) and the contact time. The velocity based contact check makes use of the current position of the surfaces and the estimated velocities in the following time step. The time that a moving contact node will intersect with a moving surface is solved as shown in Figure 3. For the contact to be valid, the point of contact must fall within the bounds of the contact surface during a time step. For some cases, there may be more than one contact surface that a node can contact within a time step. The surface that has the minimum time to contact will be selected as the contact surface. In certain cases there may be multiple contact surfaces where valid contact is possible. In these cases, a strength of contact check is used to determine the most opposed contact surface. A complete derivation of the velocity based contact check can be found in Heinstein, et al. (1993).

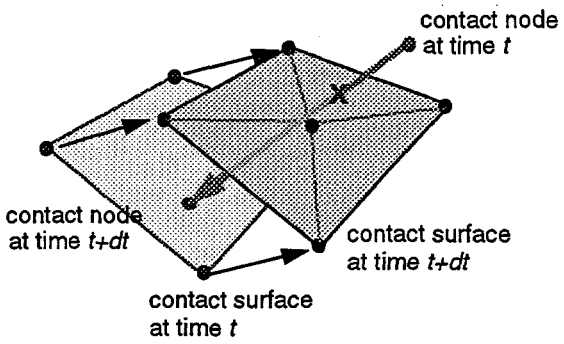

FIGURE 3.Velocity based contact check. The detailed contact check solves for the time until contact occurs(dt) and the coordinates of the contact point (x).

The calculations for a static contact check are based on the predicted configuration of the surface if the contact forces were removed. This predicted configuration would obviously have nodes penetrating contact surfaces. For contact nodes already in contact with a surface, ambiguities can arise because the surface normal is not continuous. This can result in not finding any contact when there should be one or finding multiple solutions to a single contact. The static based detailed contact checks resolve these ambiguities by distinguishing between convex and concave surfaces.

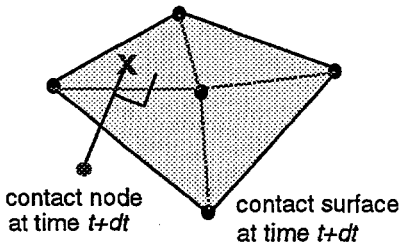

FIGURE 4.Static based contact check. The coordinates of the nearest point to a surface (x) is found using the surface normal of the last surface in contact.

The push back direction for a concave surface is determined simply by the minimum distance to the contact surface. For a convex surface, the push back direction is always along the normal of the contact surface that the contact node was previously in contact with. This avoids adding artificial velocity due to a change in push back direction, such as could occur near corners. Just as in the case of a velocity based contact check, there may be some instances where contact with multiple contact surfaces is possible according to the static contact check. Again a strength of contact check is used to determine the contact surface where the normal of the contact surface projected onto the normal of the contact node is minimized.

To summarize, the detailed contact check considers the po-

sition and velocity of both the contact node and contact surface in determining initial contact. A distinction between a concave and convex surface is made for nodes already in contact with a contact surface. This approach results in a more accurate determination of the point of contact, amount of penetration, and the direction of push back.

Enforcement of Contact Constraints

Several different approaches are available for contact enforcement. (Belytschko and Neal, 1991), (Malone and Johnson, 1994), (Laursen and Oancea, 1994) (Heinstein, Mello and Laursen, 1995). Here, we will only consider the partitioned kinematic approach used in PRONTO3D (Taylor and Flanagan, 1989), (Heinstein, et. al, 1993).

For the contact enforcement, a predictor-corrector method is used. First, the location of contact surfaces and contact nodes, assuming no contacts is predicted,

$$\hat{a} = \frac{f}{m} \tag{15}$$

$$\hat{v} = v + \Delta t\, \hat{a} \tag{16}$$

$$\hat{x} = x + \Delta t\, \hat{v} \tag{17}$$

where $\hat{a}, \hat{v}$, and $\hat{x}$ are the predicted acceleration, velocity and position respectively. The detailed contact check results in a calculated depth of penetration for each contact node into the contact surface

$$\delta = max(\hat{n} \cdot (\bar{x} - \hat{x}), 0)\,. \tag{18}$$

The contact constraint is satisfied by simultaneously applying a contact force to the contact node and to the contact surface so that the penetration is removed during the next time step. The application of this penalty force will result in both surfaces moving. Therefore, the force must be determined with an iterative method.

First we compute a penalty force assuming the contact surface cannot move. The acceleration (or force) needed to cancel the contact node penetration assuming it is contacting a rigid surface is given as:

$$a_n = \frac{\delta}{\Delta t^2} \tag{19}$$

or

$$f_s = \frac{\delta\, m}{\Delta t^2}\,. \tag{20}$$

Next we compute the movement of the contact surface that results from the application of the contact forces. More than one node may be contacting a surface; therefore, we must compute the resulting acceleration of the contact surface taking into account all contact node forces. The resulting forces acting on the surface can be determined by transforming the force to an equivalent set of forces acting at the corners of the contact surface:

$$F_i = \sum_s N_{is}(\xi, \zeta) f_s \tag{21}$$

where $N_{is}(\xi, \zeta)$ is a linear interpolation function across the contact surface and where ξ and ζ correspond to the point of contact of the node with the surface.

The total force acting on the contact surface must be assembled. The accelerations of the nodes on the contact surfaces are computed as:

$$a_i = \frac{\sum F_{is}}{m_I} \tag{22}$$

Once the accelerations of the contact surface have been computed, the initial guess for penalty force can be corrected. The acceleration of the contact point on the contact surface, due to the acceleration of contact surface, is given by:

$$a_{ps} = \sum N_{si}(\xi, \zeta) a_i \tag{23}$$

A corrected penalty force can be computed as:

$$f_s = \frac{\delta_s\, m_s}{\Delta t^2} - a_{ps} m_s\,. \tag{24}$$

To re-compute the acceleration of the contact surface using this new penalty force

$$m_i a_i = \sum_s (f_{Is} - a_{ps} m_s) \quad . \tag{25}$$

To solve for the 'best' penalty force, one would have to continue iterating. Fortunately, one pass is usually all that is required for an accurate solution. Any errors that are left after the one pass solution in the contact enforcement will be corrected in the next time step.

After assembling and solving for the motion of the contact surface, the contact node acceleration can be corrected to account for the relative motion between the contact node and the contact surface.

$$a_{ns} = a_{ps} - \frac{f_p}{m_s} \tag{26}$$

Finally, the predicted accelerations for all nodes can now be corrected by

$$a = \hat{a} + a_n \tag{27}$$

In the algorithm above, the contacts were assumed to be one-sided. That is, the contact nodes were penetrating the contact surfaces. In reality, the nodes on the contact surfaces are also contact nodes that may be penetrating other surfaces. The accuracy of the penalty force can be improved by using a symmetric (or partitioned) contact which allows both surfaces to be considered for contact simultaneously.

PARALLEL FINITE ELEMENT CALCULATION

Converting the finite-element (FE) volume integration to run on a parallel machine is a relatively straightforward task. In an explicit time stepping scheme, each mesh element interacts only with the neighboring elements that it is connected to in the FE mesh topology. If each processor is assigned a cluster of ele-

ments (submesh), then there will be a small number of point-to-point interprocessor communications along the boundary of the element clusters. These communications will be efficient since they are between adjacent processors. A variety of algorithms and tools have been developed that optimize the decomposition of the mesh into submeshes. For the calculations here, we used a software package called EDDT (Exodus Domain Decomposition Tool), which is based on CHACO (Hendrickson and Leland, 1995). The decomposition tool partitions the FE mesh, giving each processor an equal number of elements and minimizing the interprocessor communication. In practice, the resulting FE computations are highly load balanced and scale efficiently ($E_s > 0.95$) when large meshes are mapped to thousands of processors. The chief reason for the scalability is that the communication required by the FE computation is local in nature.

During each time step, the information (force, mass, etc.) on these boundary nodes must be swapped and summed between the processors. The nodes along the shared boundary of the mesh are known as communication nodes (sometimes called ghost nodes or duplicate nodes). In addition to decomposing the mesh into submeshes, the decomposition tool also defines the communication node sets necessary for the interprocessor communication.To swap information, the contributions from local elements are first summed on each processor. Then, the resulting sum is swapped and added to the results from other processors. At the end of this swap and add, each communication node should have the same result regardless of what processor it is on. Thus, the total force on the boundary nodes will be the same on all the processors. Each processor will have a "carbon" copy of the boundary nodes that can be independently integrated through time.

In order to perform the swap-and-add operation, a set of communication nodes is read from the submesh data file. Each communication set consists of nodes that live on the current processor which needs to be shared with other processors. The order of the nodes in this list is important. All processors will have a different local numbering for the communication set; however, the order will be the same for each processor. (i.e. global node number M will be the n^{th} node in a give list for all processors). First we send the information on this processor, then receive information sent by all the other processors. The ordering in the list allows the data received to be paired with the correct node for summing. Once received, the data is accumulated to form a sum.

Because the mesh connectivity does not change during the simulation (with a few minor exceptions), a static decomposition of the elements is sufficient to insure good performance. To achieve the best possible decomposition, we partitioned the FE mesh as a pre-processing step before the transient dynamics simulation was run. Similar FE parallelization strategies have been used in other transient dynamics codes (Hoover, et al. 1995) (Longsdale, et. al., 1994), (Longsdale, et al. 1995), (Malone and Johnson, 1994a 1994b).

Load Balance Issues

Efficient parallel FE computations with a contact algorithm requires a dynamic (adaptive) load balancing scheme for the contact portion of the calculations, as the following considerations indicate. Several things can cause a calculation to become unbalanced. For example, if the element assigned to one processor has a very complex material law, while other processors have very simple material laws, then the partitioning of the mesh must take into account the time it takes to evaluate each element. If the amount of time it takes to evaluate the material model varies as the problem runs, then the calculations can also become unbalanced.

Contact surfaces can also cause the calculation to become unbalanced. In simple problems, the locations of contacts can be guessed. However, in complex crash simulations, the location of contacts cannot be predicted.

On a parallel machine, contact detection is even more problematic. First, in contrast to the FE portion of the computation, some form of global analysis and communication is now required because the FE regions in contact can be owned by any two processors. Second, load balance is a serious problem. Formally, the task is to find all the geometric penetrations of a set of contact surfaces (faces of elements) by a set of contact nodes (corner points of elements). These contact surfaces and nodes come from elements that lie on the surface of the meshed object volumes and, thus, comprise only a subset of the overall FE mesh. Since the FE decomposition described above load balances the entire FE mesh, it will not (in general) assign an equal number of contact surfaces and nodes to each processor. Finally, finding the one (or more) surfaces that a node penetrates requires that the processor that owns the node acquire information about all surfaces that are geometrically nearby. Even if we devise a global communication scheme or a new decomposition technique that provides this information, the decomposition technique must be *dynamic* or adaptive instead of static because the set of nearby surfaces changes as the simulation progresses.

PARALLEL CONTACT ALGORITHM

In the sections below, we will outline the changes required to convert the serial contact algorithm to run on a parallel machine.

Identification of Contact Surface

Determining the exterior surface for a parallel calculation follows the same logic as for a serial calculation. Each sub-mesh is processed independently, and the communication side sets are used to remove the element faces along the communication boundary. Since the communication side sets are computed when the mesh is decomposed, no interprocessor communications are required to determine the external surfaces.

If we allow elements to be deleted as the analysis progresses, then the surface identification phase becomes more complicated with multiple processors. As elements are deleted from a submesh, the opposing face list must be used to update a faces on other processors. A communication step is required to update the face on the opposing processor.

Parallel Contact Detection

The most commonly used approach (Longsdale, et al.,

1994), (Longsdale, et al. 1995), (Malone and Johnson, 1994a 1994b) has been to use a single, static decomposition of the mesh to perform both FE computation and contact detection. At each time step, the FE region owned by a processor is bounded with a box. Global communication is performed to exchange the bounding box's extent with all processors. Then, each processor sends contact surface and node information to all processors with overlapping bounding boxes. This communication gives each processor the information needed to perform the contact detection. Though simple in concept, this approach will not efficiently load balance the contact detection for general problem.

A better load balance has been obtained by using separate decompositions for the contact detection and the finite element analysis. (Hoover, et al., 1995). The contact surfaces and nodes are first sorted into buckets (bins) by overlaying a regular, coarse 3-D grid on the entire simulation domain. The 3-D grid is used to generate a set of buckets, with the nodes and surfaces being distributed to the buckets based on their location within the 3-D grid (Whirley and Engelman, 1994), (Benson and Halquist, 1990),(Belytschko and Neal,1989). To perform the contact check, the nodes from a given bucket are combined with the nodes from neighboring buckets to form a node pool from which nodes and surfaces are checked for contact.

For the parallel implementation of the bucket sort, Hoover, et al. (1995) divided the coarse grid along one dimension into slices. One processor is responsible for contact detection within a slice. The set of buckets along the processor boundary must be communicated with the adjacent processors. While this approach is likely to perform better than a static decomposition, the implementation described in Hoover, et al. (1995) suffered from load imbalance and did not scale to large numbers of processors.

The bucket sorting algorithm described above can also suffer if the simulated body geometry expands with time. As the body expands, more buckets (memory) will be required. In addition, the size of the buckets are limited by the size of the largest element. Thus, if one element is very large, the number of points within a bucket can be quite large. In addition, if the problem geometry is very not space filling, then there will be many empty buckets, which can be an inefficient use of memory.

An important aspect of our approach is the use of a different decomposition for contact detection than we used for the finite element calculation. This allowed us to optimize each portion of the code independently. The key difference is that for contact detection, we used a dynamic technique known as recursive coordinate bisection (RCB) to generate the decomposition anew at each time step. We have found several advantages to this approach. First, and foremost, RCB assigns processor nearly the same number of contact nodes and surfaces, thereby achieving nearly perfect load balance in the contact detection calculation that occurs on each processor. Second, the cost of performing an RCB decomposition is minimal if it begins from a nearly balanced starting point. We use the RCB result from the previous time step, which will always be close to the correct decomposition for the current time step. Third, the local and global communication patterns in our algorithm are straightforward to implement and do not require any complicated analysis of the simulation geometry.

The price for these advantages is that we must communicate information between the FE and contact decompositions at every time step. Our results indicate that the advantage of achieving load balance greatly outweighs the extra communication cost of maintaining two decompositions.

In the next section we will provide some background material that will help explain in more detail our algorithm in the following section. Performance from simulations using PRONTO3D are then presented.

Background Material

Unstructured Communications. The parallel contact algorithm involves a number of unstructured communication steps. In these operations, each processor has some information it wants to share with a handful of other processors. Although a given processor knows how much information it wants to send and to whom, it doesn't know how much it will receive and from whom. Before the communication can be performed efficiently, each processor needs to know the number and sizes of the messages it will receive.

We accomplish unstructured communications with the approach sketched in Figure 5.

(5.1)	Form vector of 0/1 denoting who I send to
(5.2)	Fold vector over all *P* processors
(5.3)	Gather number of receives for my processor, *nrecvs* = vector*(q)*
(5.4)	For each processor I have data for, send message containing size of the data
(5.5)	Receive *nrecvs* messages with sizes coming to me
(5.6)	Allocate space & post asynchronous receives
(5.7)	Synchronize
(5.8)	Send all my data
(5.9)	Wait until I receive my data

FIGURE 5.Unstructured communications.

In steps (5.1)- (5.3) each processor learns how many other processors want to send it data. In step (5.1) each of the *P* processors initializes a *P*-length vector with zeroes and stores a 1 in each location corresponding to a processor it needs to send data to. The fold operation (Fox, et al. 1988) in step (5.2) communicates this vector in an optimal way; processor *q* ends up with the sum across all processors of only location *q*, which is the total number of messages it will receive.

In step (5.4) each processor sends a short message to the processors it has data for, indicating how much data they should expect. These short messages are received in step (5.5). With this information, a processor can now allocate the appropriate amount of space for all the incoming data, and post receive calls which tell the operating system where to put the data once it arrives. After a synchronization in step (5.7), each processor can now send its data. The processor can proceed once it has re-

ceived all its data.

Recursive Coordinate Bisectioning. The recursive coordinate bisectioning (RCB) algorithm we used was first proposed as a static technique for partitioning unstructured meshes (Berger and Bokhari, 1987). Although the RCB algorithm has been eclipsed by better approaches for static partitioning, RCB has a number of attractive properties when used as a dynamic partitioning scheme (Jones and Plassmann, 1994). The subdomains produced by RCB are geometrically compact and well-shaped. The algorithm can also be parallelized in a fairly inexpensive manner. Also, RCB is attractive because small changes in the geometry induce only small changes in the partitions. Most partitioning algorithms do not exhibit this behavior.

Figure 6 shows a graphical picture of how a RCB decomposition progresses. The goal of the RCB algorithm is to divide equally among P processors the combined set of N contact surfaces and nodes. For this operation we treated each surface as a single point. Initially, each processor owns the subset of the points based on the finite element decomposition. This decomposition may scatter the points anywhere in the domain. Some processors may have many points, while others have none.

The first step in the RCB is to choose one of the coordinate directions, x, y, or z, for bisectioning. For this first cut, we chose the direction that results in the sub-domains being as cubic as possible. The next task is to position the cut, shown as the dotted line in the figure, at a location which puts half the points on one side of the cut, and half on the other. This step is equivalent to finding the median of a distributed set of values in parallel.

To find the median, we use an iterative algorithm. First, we select the geometric midpoint of the box. Each processor counts the number of points it owns that are on one side of the cut. Summing this result across processors determines which direction the cut should be moved to improve the median guess. In practice, within a few iterations we find a suitable cut that partitions the points exactly. Then, we divide the processors into two groups, one group on each side of the cut. Each processor sends its points that fall on the far side of the cut to a partner processor in the other group. Likewise, each processor receives a set of points that lie on its side of the cut. These steps are outlined in Figure 7.

After the first pass through steps (7.1)-(7.4), we have reduced the partitioning problem to two smaller problems, each of which is to partition $N/2$ points on $P/2$ processors within a new bounding box.

Thus, we can recurse on these steps until we have assigned N/P points to each processor, as shown in Figure 6 for an 4-processor example. The final geometric sub-domain owned by each processor is a regular parallelepiped. Note that it is simple to generalize the RCB procedure for any N and non-power-of-two P by adjusting our desired "median" criterion at each stage to insure the correct number of points end up on each side of the cut.

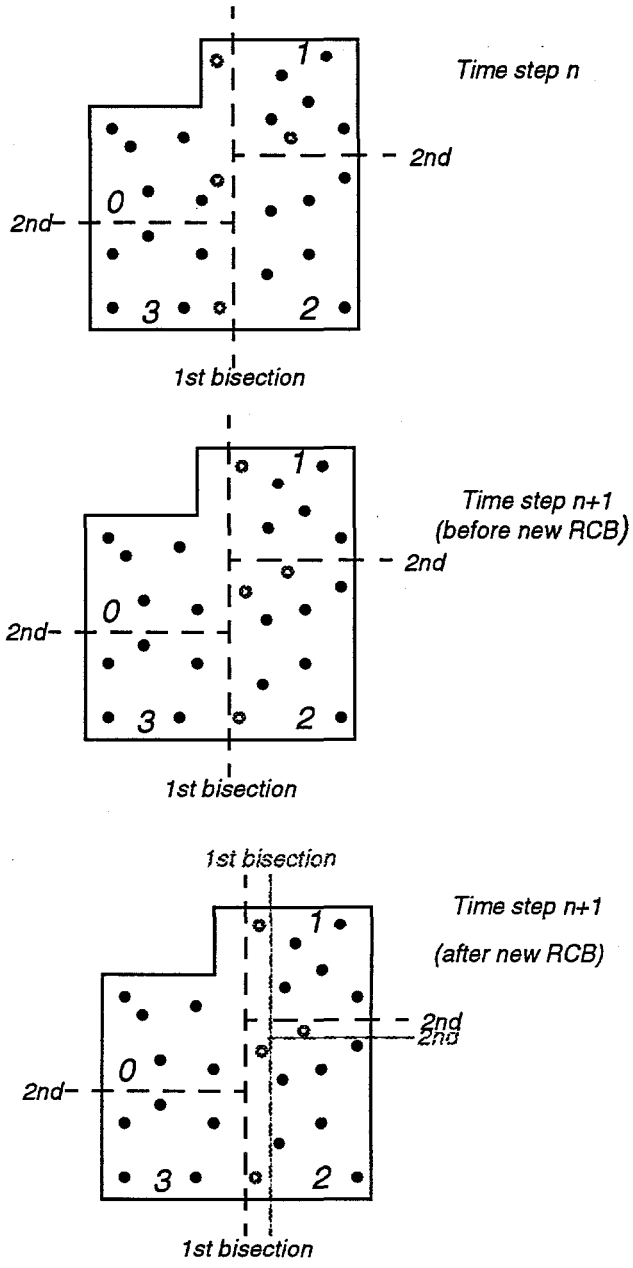

FIGURE 6. Recursive Coordinate Bisection. The global domain is divided in half by all the processors based on the surface coordinates. Then each half is divided in half by half of the processors, recursively, until the entire domain is divided. At each time step, the algorithm starts with a nearly optimal approximation.

(7.1) Choose a coordinate axis *(xyz)*
(7.2) Position cut so as to partition points equally
(7.3) Send points that lie on far side of cut
(7.4) Receive points that lie on my side of cut
(7.5) Recurse

FIGURE 7.Steps used in RCB algorithm.

Dynamic Decomposition for Contacts

Our parallel algorithm for contact detection is outlined in Figure 8. In step (8.1), the current position of each contact surface and node is communicated from the processor who owns it in the FE decomposition to the processor who owned it in the RCB decomposition during the previous time step. (On the first time step, this step is simply skipped.) The above communication step involves unstructured communication as detailed in the section titled Unstructured Communications. This step gives the RCB decomposition a starting point close to the correctly balanced answer, since the finite elements do not move far in any one time step. In step (8.2) we perform the RCB decomposition as described in the previous section to rebalance the contact surfaces and nodes based on their current positions.

(8.1) Send contact data from FE decomposition to old RCB decomposition
(8.2) Perform parallel RCB to rebalance
(8.3) Share RCB cut info with all processors
(8.4) For all my surfaces
  If surface capture box extends beyond my RCB box:
  Determine what other processors need it
(8.5) Send overlapping surfaces to nearby processors
(8.6) Find contacts within my RCB box
(8.7) Send contact results to FE owners

FIGURE 8.Outline of dynamic decomposition for parallel contacts.

The entire RCB decomposition can be represented as a set of *P-1* cuts, one of which is stored by each processor as the RCB decomposition is carried out. In step (8.3) we communicate this cut information so that every processor has a copy of the entire set of cuts. This communication is done via an *expand* operation (Fox, et al. 1988).

Before contact detection is performed, each processor must know about all contact surfaces that are near any of its contact points. Because we represented a surface as a single point during the RCB decomposition, some of these nearby surfaces will actually be owned by surrounding processors. So in step (8.4), each processor determines which of its contact surfaces extends beyond its RCB sub-domain. For those that do, a list of processors who need to know about that surface is created. This list is built using the RCB vector of cuts created in step (8.3). The information in this vector enables a processor to know the bounds of the RCB sub-domain owned by every other processor. In step (8.5), the data for overlapping contact surfaces is communicated to the appropriate processors.

Detailed Contact Check. In step (8.6) each processor can now find all the contacts that occur in its geometric RCB sub-domain. A nice feature of our algorithm is that this detection problem is identical conceptually to the global detection problem we originally formulated, namely to find all the contacts between a set of surfaces and nodes bounded by a box. In fact, in our contact algorithm, each processor calls the original serial PRONTO3D contact detection routine to accomplish step (8.6). Calling the serial routines at this point enables the code to take advantage of the *point-in-box* sorting and searching features the serial routine used to efficiently find contacts. It also means we did not have to recode the complex geometry equations that compute intersections between moving 3-d surfaces and points! Finally, in step (8.7), information about contacting surfaces and nodes is communicated back to the processors who own them in the FE decomposition. Those processors can then perform the appropriate force calculations and contact push-back.

In summary, steps (8.1), (8.5), and (8.7) all involve unstructured communication of the form outlined in Figure 5. Steps (8.2) and (8.3) also consist primarily of communication. Steps (8.4) and (8.6) are solely on-processor computation.

Parallel Contact Enforcement

The contact enforcement is performed in the FE decomposition. One could argue that a separate decomposition should be used to better insure load balance during this step. However, for most problems, only a small portion of the problem will be in contact (there may be fewer contacts than there are processors). In addition, the contact enforcement is computationally inexpensive.

Sense the contact enforcement was done in the FE decomposition, the existing serial algorithm was modified by adding communication steps at selected locations in the algorithm. For the parallel algorithm, each processor will own a set of contact nodes that can be in contact with a surface that may live on other processors. The contact search will return to each processor in the FE decomposition three things: a list of nodes in contact; the surfaces these nodes contact; and the processor on which the contacted surface lives. Because the enforcement calculations are being done in the FE decomposition, the processor that the surface lives on may not be adjacent to the current processor. However, the location of the processor is known which allows for direct point-to-point communication.

The initial penalty force in Eq. (19) can be computed without communications because the detailed contact check will return the penetration, surface id and surface processor id for each contact node that is in contact.

For the calculation in Eq. (22) communications are required. The penalty forces from each contact node in contact with a sur-

face must be summed to the corners of the contact surface. The initial penalty force must be communicated to any surfaces that do not live on the current processor. Thus, each processor must receive penalty forces to be accumulated in the surface sum in Eq. (21). This communication step is done using the unstructured communication scheme outlined in the Unstructured Communications section.

The assembly of the total force acting on the contact surface, Eq. (22), requires the nodal sums from each surface to be combined into an equivalent global force, a process that requires swapping and adding the forces along processor communication boundaries.

Once the acceleration of the contact surface has been computed, the contact surface accelerations must be communicated to the processors that own the contact node so that the acceleration of the contact point, Eq. (23), can be used to correct the penalty force, Eq. (24). One more communication is required to compute the acceleration of the contact surface using this new penalty force, Eq. (25).

In summary, the contact enforcement algorithm requires numerous small communication steps to trade information related to contact surfaces back and forth between processors. The algorithm follows the same logic as the serial algorithm, with the addition of the communication steps. In practice, the number and size of communications is small.

RESULTS

Bending Beam

As a first example, we will consider the parallel performance of a simple problem with no contacts. A simple vibrating beam with a uniform pressure, symmetry plane, and a pinned support is used for this purpose. This beam example problem is based on the example presented in Flanagan and Belytschko's (1982) classic paper on orthogonal hourglass control. The example has a pressure load along the top of the beam and a pinned boundary condition on each side as shown in Figure 9. A simple elastic material model was used. A plane strain assumption is created by prescribing no displacement boundary conditions along the front and back sides of the beam.

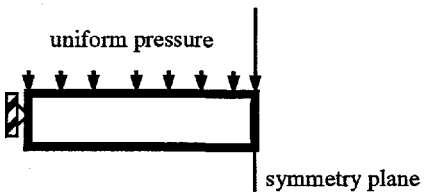

FIGURE 9.Beam bending example problem.

Fixed-Size Speedup. The performance for the vibrating beam problem with 19500 elements and 22,506 nodes is shown in Figure 10. The parallel calculation remained efficient for over 512 processor until there were only approximately 40 elements per processor.

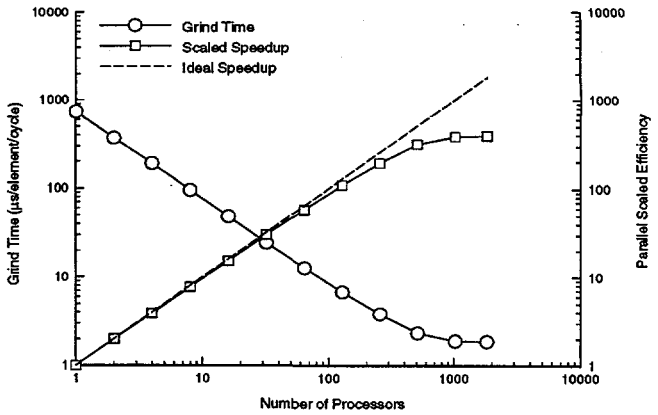

FIGURE 10.Performance for the fixed-size beam problem.

In the above example, the problem size was held fixed while the number of processors increased. Since the communications routines have a fixed overhead, the work to be done on a given processor must be greater than the communications overhead. The algorithm will continue to become more efficient as processors are added to the problem, as long as there is enough computational work for each processor. The low number of 40 elements per processor means that in addition to running very large problems, relatively small problems can be run very fast by dividing the work over many processors. For example, a problem with 40,000 elements should run optimally up to 1000 processors, while a problem with 4000 elements should run optimally up to 100 processors.

If the problem size is increased as the number of processors is increased, then the scaled performance can be measured. Figure 11 shows the scaled performance for the beam problem.

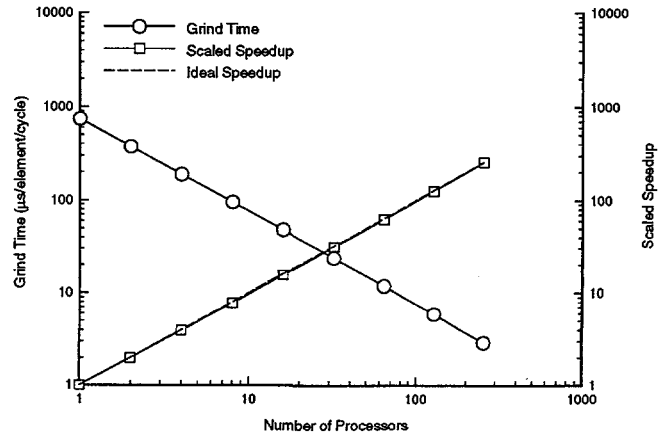

FIGURE 11.Scaled performance for the vibrating beam problem.

Here, the number of elements in the problem was proportional to the number of processors. For example, a one-processor problem would use 6480 elements. A two-processor problem would have 12960 elements. The maximum number of elements that

would fit on the smallest processor was 6480.

The performance scales linearly to 256 processors with 1,658,880 elements. Since each processor had the maximum number of elements that would fit on a processor, the computational work for each processor is quite large compared to the communication overhead. Limitations in the pre- and post-processing tools limited the calculation to 256 processors. This problem should continue to scale for the maximum number of processors on Sandia's Intel Paragon, 1824. If the maximum number of elements per processor were used, this would lead to a maximum problem size of over 11,000,000 elements.

Brick Wall

This example considers a wall of bricks being hit by an elastic-plastic rod. The initial geometry is shown in Figure 12. The impact causes the bricks to bounce off each other in an unpredictable manner. One of the added capabilities of the contact material algorithm is the efficient modeling of multi-body impact without *a priori* definition of contact surfaces. This example considers an elastic-plastic bar impacting a stack of 17 elastic bricks. A stationary elastic-plastic wall is also resting against the stack of bricks. All contact nodes and contact surfaces on the bodies were automatically defined using the contact material algorithm.

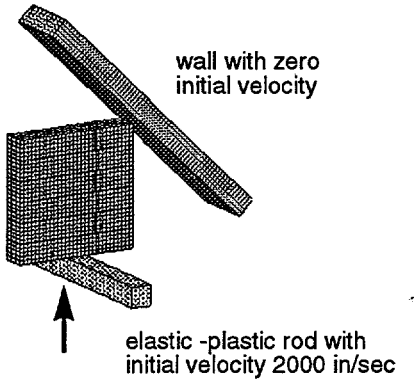

FIGURE 12.Brick wall example. A stack of bricks with a zero initial velocity are impacted by an elastic plastic rod.

The resulting deformed shape of the brick wall problem is shown in Figure 13.The impact of the bricks by the rod scatters the bricks geometrically. During the early stages if this problem, there are a large number of contacts that must be enforced. At late times, the bricks have spread out and very few contacts occur in each time step. Thus, for this problem, most of the contact algorithms time is spent searching for contacts.

The performance of the fixed-size brick wall problem is shown in Figure 14. For this calculation, 8400 elements were used (54 elements were used to model each full brick). The number of elements was held fixed while the number of processors increased. The calculation efficiently fell off when the number of processor exceeded 64. At this point each processor had only 130 elements.

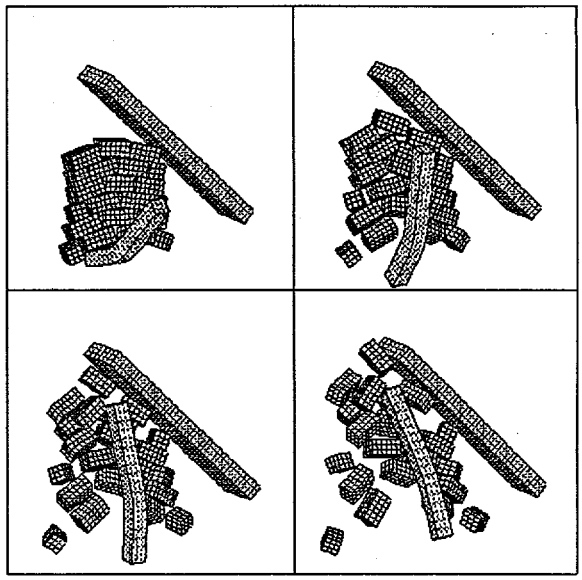

FIGURE 13.Deformed shape of bricks after impact.

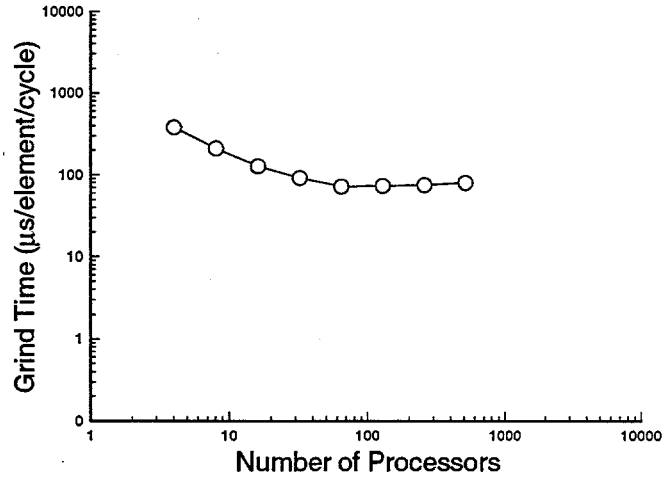

FIGURE 14.Performance of the fixed-size brick wall problem.

Figure 15 shows the performance for the scaled brick wall problem. Here, the number of elements per brick was increased as the number of processors increased. Each processor had 1890 elements/processor. The computational efficiency continued to increase as the number of processors increased.

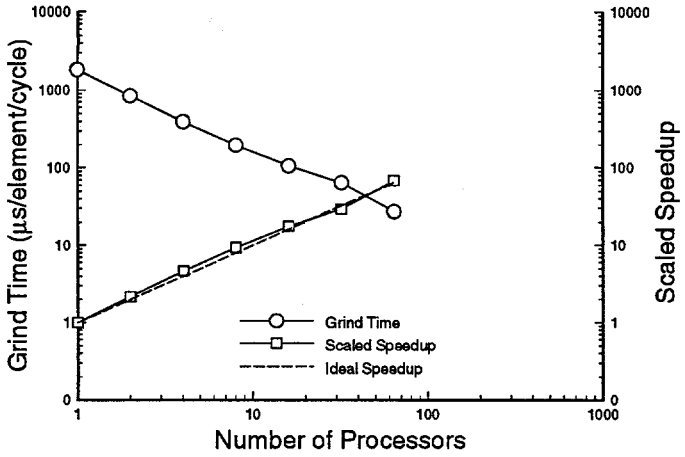

FIGURE 15.Performance on the scaled brick wall problem for 1890 elements/processor.

Can Crush

Figure 16 shows a schematic for simulation of a steel shipping container being crushed due to an impact with a flat inclined plate. In the finite element simulation, a symmetry plane was used so that only one half of the container was simulated. As the container crumples, numerous contacts occur between layers of elements on the folding surface. We have used this problem to test and benchmark our serial and parallel contact algorithm (Heinstein, et al.,1993).

The can is 0.25 inches thick, has an inside radius of 5 inches, and is 15 inches long. The bottom of the can is constrained in all directions. The 22x11 in. plate is 2.5 inches thick and is initially tilted at a 10 degree angle as it impacts the can at 5000 in/s.

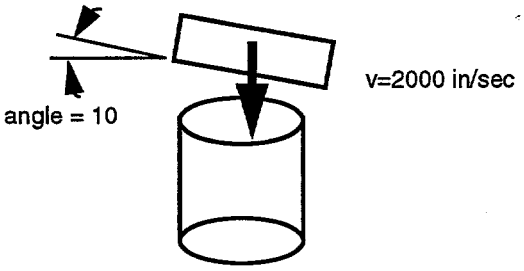

FIGURE 16.Schematic of the can crush model

Fixed Problem Size. Here, we present timing results for a fixed-size problem geometry containing 7152 finite elements. For the fixed-size scaling problem, both the container and wall were meshed 3 elements thick, so roughly 2/3 of the elements are on a surface.

Since the can will deform plastically, the number of integration points through the thickness will affect the accuracy. With only three integration points, the detection of plastic strain will be limited. More integration points would provide a more accurate integration of the plastic strain. Here, we are simply using three points through the thickness as a numerical example. We will refine the mesh in the scaled test problem in the next section.

Since each surface element contributes both a surface and node, there were about 9500 contact surfaces and nodes in the problem. Whether in serial or parallel, PRONTO3D spends virtually all of its time in two portions of the time-step calculation - FE computation and contact detection. For the serial code, the contact detection takes approximately the same amount of time as the FE computation. For the parallel code, the contact detection takes about twice as along as the FE calculation.

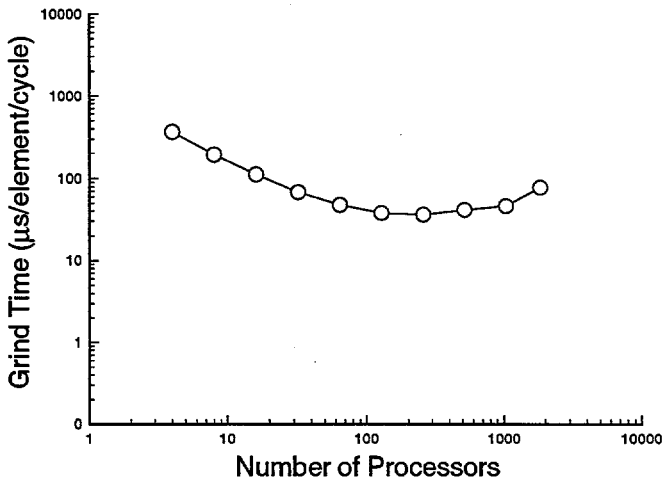

FIGURE 17.Grind time for the fixed-size can crush problem.

The average CPU time per time step for simulating this problem on various numbers of Paragon processors from 4 to 1840 is shown in Figure 17. For this problem, both portions of the code speed-up adequately on small numbers of processors, but begin to fall off when there are approximately one hundred elements per processor.

Scaled Problem Size. Figure 18 shows performance on a scalable version of the crush simulation, where the container and surface are meshed more finely as more processors are used. On one processor a 1875-element model was run. Each time the processor count was doubled, the number of finite elements was also doubled by halving the mesh spacing in a particular dimension. Thus, all the data points are for simulations with 1875 elements per processor; the largest problem was 480,000 elements on 256 processors.

As a point of reference, the grind time for a 60,000 element problem on the Cray Y-MP was T_{grind} = 41.25 µs/element/cycle. The grid time on the 32 processor problem was 40.75 µs/element/cycle. The maximum grind time for the 256 processor problem was ~ 5µs/element/cycle.

In contrast to the previous graph, we now see excellent scalability. A breakdown of the timings shows that the performance of the contact detection portion of the code is now scaling as well or better than the FE computation, which was our original goal with

this work. In fact, since linear speed-up would be a linear line with slope 1.0 on this plot, we see apparent super-linear speed-up for some of the data points! This is due to the fact that we are really not exactly doubling the computational work each time we double the number of finite elements.

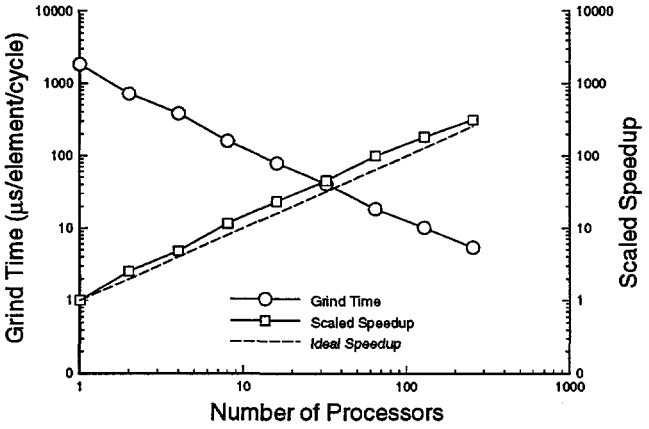

FIGURE 18.Grind time and parallel scaled efficiency for the scaled can crush problem.

There are several factors that could explain the superlinear speedup. First, the mesh refinement scheme we used does not keep the surface-to-volume ratio of the meshed objects constant, so that the contact algorithm may have less (or more) work to do relative to the FE computation for one mesh size versus another. Second, the time step size is reduced as the mesh is refined. This actually reduces the work done in any one time step by the serial contact search portion of the contact algorithm (step (8.6) in Figure 8), since contact surfaces and nodes are not moving as far in a single time step. The bounding box for each contact surface must be expanded by the maximum amount that a contact node can move. In parallel, the bounding box must be expanded only by the maximum that a node moves on a given processor. Thus, tighter bounding boxes will mean less work. More generally, the number of actual contacts that occur in any given time step for a given processor will not exactly double just because the number of finite elements is doubled.

The second reason for super linear speedup is due to a more subtle effect in the parallel contact algorithm. When the serial contact algorithm searches for contacts in a large region it performs various sorts and searches to optimize its operation. Now consider what happens if we use 2 processors to perform a parallel contact search on the same region. The RCB decomposition effectively sorts the contact surfaces and nodes at a high-level, so that the serial algorithm working on each processor operates on a smaller geometric sub-region. If the RCB decomposition running in parallel is more efficient than the serial algorithm at performing this geometric sort, as it sometimes is in practice, then our parallel contact detection algorithm is actually reducing the total amount of work performed.

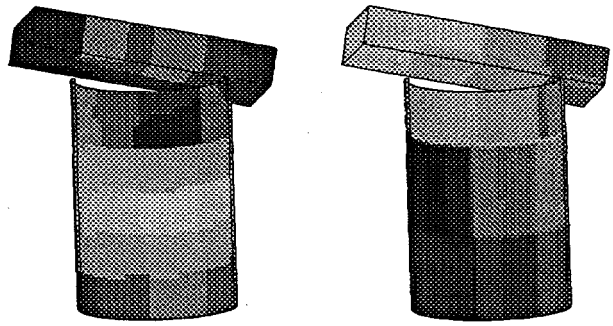

FIGURE 19.Decomposition generated by a) CHACO and b) RCB at time zero

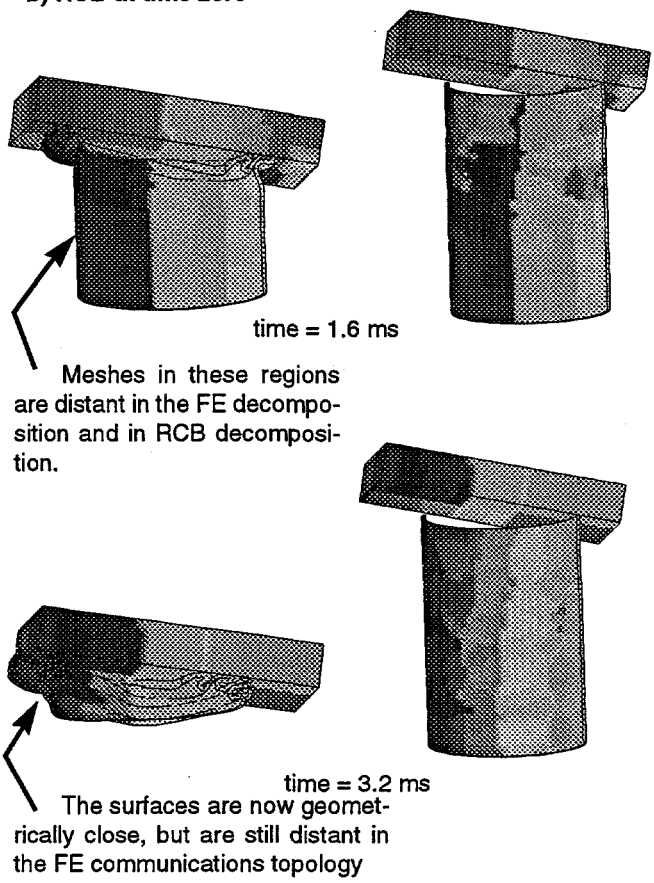

FIGURE 20.RCB mapping for times t=1.6 ms and t= 3.2ms. Deformed and undeformed mesh.

Mesh Decomposition. Figure 19 shows the CHACO and the RCB decompositions at time zero. Figure 19a shows a plot coded according to the processor that owns the element. Figure 19b shows a color coded plot where the colors correspond to which processor owns the surface. Note that the CHACO and RCB decompositions are not the same even at time zero. In fact, the top of the mesh is assigned to processor zero in the CHACO

mesh, while in the RCB decomposition, the contact surfaces in this same region are assigned to processor 31.

In Figure 20, the RCB decomposition at time t=1.6 ms and t = 3.2 ms is shown mapped onto the deformed and undeformed shapes. The plots show how the surfaces are locally mapped to a given processor. Some of the surfaces that start on processor 0 at the start of the problem, remain on that processor for the duration of the calculation. While other surfaces (i.e. the top of the block) are mapped onto processor zero as the calculation progresses.

SUMMARY

A scalable contact algorithm was developed for the transient finite element code PRONTO3D. Two different problem decomposition schemes were used to insure load balancing on all processors. The algorithm used a static decomposition for the finite element mesh (provided by CHACO) and a dynamic decomposition (provided by recursive coordinate bisection, or RCB) for determining the contacting surfaces. The static decomposition distributes the three-dimensional mesh into compact subdomains in which processors communicate only with their nearest neighbors. The dynamic decomposition redistributes the contact surfaces equally over all the processors and provides a decomposition in which processors communicate only with nearest neighbors.

A well load-balanced contact search was developed. All the processors contributed equally, rather than only those which "owned" contact surfaces in the static finite element decomposition. The contact search by each processor is over a small subdomain. The same efficient search algorithm used in the serial code was used on each subdomain, and is more efficient on the subdomain than on the global domain. At each time step, the RCB algorithm starts from a near-optimal decomposition from the previous time step. The decomposition is dynamic but incremental, and is, thus, very efficient.

The new algorithm has some additional costs: extra memory is required to perform the dynamic RCB decomposition; communication is required within the RCB decomposition; and communication is required between the static and dynamic decompositions.

Despite these costs, we have found in practice that the contact algorithm is almost perfectly scalable. The speedup in the execution increases as the number of processors increases.

REFERENCES

Flanagan, D. P., and Belytschko, T., 1982, "A Uniform Strain Hexahedron and Quadrilateral with Orthogonal Hourglass Control", *J. Comp. Meths. Appl. Mechs. Eng.*, Vol 30.

Belytschko, T. and Lin, J.I., 1987 "A Three-Dimensional Impact-Penetration Algorithm with Erosion," *Computers and Structures*, Vol. 25, No. 1, pp. 95-104.

Berger, M.J. and Bokhari, 1987 "A partitioning strategy for nonuniform problems on multiprocessors", *IEEE Trans. Computers*, C-36, pp 570-580.

Taylor, L.M. and Flanagan, D.P., 1987 *PRONTO2D: A Two-Dimensional Transient Solid Dynamics Program*, SAND86-0594, Sandia National Laboratories, Albuquerque, NM 87185.

Fox, G. C.; Johnson, M. A.; Lyzenga, G.A.; Otto, S.W.; Salmon J. K. and Walker, D. W., 1988 *Solving Problems on Concurrent Processors: Volume 1*, Prentice Hall, NJ.

Belytschko, T. and Neal, M. O., 1989 "Contact-impact by the pinball algorithm with penalty, projection and Lagrangian methods," *Proc Symp. on Computational Techniques for Impact, Penetration, and Perforation of Solids*, ASME AMD 103, pp 97-140.

Taylor, L.M. and Flanagan, D.P., 1989 *PRONTO3D: A Three-Dimensional Transient Solid Dynamics Program*, SAND89-1912, Sandia National Laboratories, Albuquerque, NM 87185.

Attaway, S. W., 1990, "Update of PRONTO 2D and PRONTO 3D Transient Solid Dynamics Program," SAND90-0102, Sandia National Laboratories, Albuquerque, New Mexico, November, 1990.

Benson, B.J. and Hallquist, J.O., 1990 "A Single Surface Contact Algorithm for the Post-Buckling Analysis of Structures," *Computer Methods in Applied Mechanics and Engineering*, Vol. 78, pp. 141-163.

Belytschko, T. and Neal, M.O., 1991 "Contact-Impact by the Pinball Algorithm with Penalty and Lagrangian Methods," *Int. J. Numerical Methods Eng.*, Vol. 31, pp. 547-572.

Bergmann, V.L., 1991, "Transient Dynamics Analysis of Plates and Shells with PRONTO 3D," SAND91-1182, Sandia National Laboratories, Albuquerque, New Mexico, September 1991.

Laursen, T.A., & J.C. Simo (1991), ``On the Formulation and Numerical
Treatment of Finite Deformation Frictional Contact Problems," in Nonlinear Computational Mechanics -- State of the Art, P. Wriggers & W. Wagner, eds., Springer-Verlag, Berlin, pp. 716-736.

Plaskacz, E.J.; Belytschko, T. and Chiang, H. Y., 1992, "Contact-Impact Simulations on Massively Parallel SIMD Supercomputers," *Computing Systems in Engineering*, Vol 3, Nos 1-4, pp 347-355.

Heinstein, M.W.; Attaway, S. W.; Mello, F. J.; and Swegle, J. W., 1993 "A general-purpose contact detection algorithm for nonlinear structural analysis codes," SAND92-2141, Sandia National Laboratories, Albuquerque, NM.

Sjaardema, G. D., 1993, "Overview of the Sandia National Laboratories Engineering Analysis Code Access System," SAND 92-2292, Sandia National Laboratories, Albuquerque, NM.

Attaway, S.W., Heinstein, M.W., and Swegle, J. W., 1994, "Coupling of smooth particle hydrodynamics with the finite element method," *Nuclear Engineering and Design*, 150, pp 199-205.

Jones, M. and Plassman, P., 1994 "Computational results for parallel unstructured mesh computations," *Computing Systems in Engineering*, 5, pp 297-309.

Laursen, T.A., & V.G. Oancea (1994), ``Automation and Assessment of Augmented Lagrangian Algorithms for Frictional Contact Problems," *Journal of Applied Mechanics*, 61, pp 956-963.

Longsdale, G.; Clinckemaillie, J.; Vlachoutsis, S.; and Dubois, J., 1994 "Communications requirements in parallel

crashworthiness simulations," Proc. HPCN'94, Lecture Notes in Computer Science 796, Springer, pp 55-61.

Malone, J. G. and Johnson, N. L., 1994a "A Parallel Finite Element Contact/Impact Algorithm for Non-Linear Explicit Transient Analysis: Part I - The search Algorithm and Contact Mechanics," *International Journal for Numerical Methods in Engineering*, Vol. 37, 559-590.

Malone, J. G. and Johnson, N. L., 1994b "A Parallel Finite Element Contact/Impact Algorithm for Non-Linear Explicit Transient Analysis: Part II - Parallel Implementation," *International Journal for Numerical Methods in Engineering*, Vol. 37, 591-603.

Swegle, J.W.; Attaway, S.W.; Heinstein, M.W., and Hicks, D.L., 1994, "An Analysis of Smoothed Particle Hydrodynamics," SAND 93-2513, Sandia National Laboratories, Albuquerque, NM.

Whirly, R. G. and Engelman, B. E., 1994 "Automatic contact algorithm in DYNA3D for crashworthiness and impact problems," *Nuclear Engineering and Design*, Vol 150, pp 225-233.

Zhong, Z. H. and Nilsson, L., 1994 "Contact-Impact Algorithms on Parallel Computers", *Nuclear Engineering and Design*, Vol. 150, pp 253-263.

Heinstein, M.W., F.J. Mello & T.A. Laursen (1995), ``Augmented Lagrangian Algorithms for Enforcement of Contact Constraints in Explicit Dynamicand Matrix-Free Quasistatic Applications," in Contact Mechanics II:Computational Techniques, M.H. Aliabadi & C. Alessandri, eds., Computational Mechanics Publications, Southampton, pp. 289-296.

Hendrickson, B. and Leland, R., 1995 "The Chaco User's Guide: Version 2.0," SAND94-2692, Sandia National Labs, Albuquerque, NM, June.

Hoover, C. G.; DeGroot, A.J.; Maltby, J. D.; and Procassini, R. D., 1995 "Paradyn: Dyna3d for massively parallel computers," Presentation at Tri-Laboratory Engineering Conference on Computational Modeling, October.

Longsdale, G.; Elsner, B.; Clinckemaillie, J.; Vlachoutsis, S.; De Bruyne, F. and Holzner, M., 1995 "Experiences with industrial crashworthiness simulations using the portable, message-passing PAM-CRASH code," *Proc HPNC'95, Lecture Notes in Computer Science* 919, Springer, pp 856-862.

DISCLAIMER