To be presented at 1996 Intel Supercomputer Users' Group Conference, June 19-22, Knoxville, TN, and to be published in a special issue of International Journal of Computers and Mathematices with Applications

SAND96-0592C
CONF-9606195--5

A Parallel Algorithm for Transient Solid Dynamics Simulations with Contact Detection

Stephen Attaway, Bruce Hendrickson, Steve Plimpton, David Gardner, Courtenay Vaughan, Martin Heinstein, James Peery
Mail Stop 0437
Sandia National Labs
Albuquerque, NM 87185-0437

Abstract-Solid dynamics simulations with Lagrangian finite elements are used 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 will explore some of the issues involved in developing efficient, portable, parallel finite element models for nonlinear transient solid dynamics simulations.

The contact-detection problem poses interesting challenges for efficient implementation of a solid dynamics simulation on a parallel computer. The finite element mesh is typically partitioned so that each processor owns a localized region of the finite element mesh. This mesh partitioning is optimal for the finite element portion of the calculation 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. Load-imbalance can become a problem since the finite element decomposition divides the volumetric mesh evenly across processors but typically leaves the surface elements unevenly distributed.

In practice, these complications have been limiting factors in the performance and scalability of transient solid dynamics on massively parallel computers. In this paper we present a new parallel algorithm for contact detection that overcomes many of these limitations. In contrast to previous approaches, we use a different decomposition scheme for assigning surface elements to processors than is used for the volumetric finite element mesh.[1]

Keywords-parallel, finite element, contact, impact, dynamic

1. INTRODUCTION

In this paper we discuss the details of our contact algorithm and also its implementation in a parallel version of PRONTO3D, a large-scale transient solid dynamics code developed at Sandia.[1], [2], and [3]. We also present timing and scalability results for some large prototypical simulations which illustrate how the new contact-detection algorithm has enabled efficient parallelization of PRONTO3D. We have conducted simulations involving up to two million computational elements and have demonstrated that the new algorithm is scalable in practice to over 1000 processors of the Intel Paragon.

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

Sandia's PRONTO3D code is a Lagrangian finite element program for the analysis of the three-dimensional response of solid 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. The algorithms within PRONTO3D were designed to be accurate, dependable, and to execute rapidly.

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 hexahedral elements; Assumed Strain hourglass control for hexahedral elements; objective material coordinate system; and global contact (self-contact) with erosion of surfaces.

The structure of this paper is as follows: We review the components of an explicit time step in PRONTO3D, then in preparation for describing the parallel contact algorithm, we review 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.

2. Explicit time step integration

An outline of the computations performed every time step in the explicit time-step algorithm in PRONTO3D is shown in Figure 1.

(1.1) Define/redefine contact surface
(1.2) Integrate the equations of motion
(1.3) Compute the strain rate based on the motion of the nodes
(1.4) Compute the stress rate based on the strain rate and the material models
(1.5) Compute internal forces acting on nodes
(1.6) Predict the locations of the nodes assuming no contacts
(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

FIGURE 1.Outline of explicit time step algorithm.

Steps (1.2) through (1.5) may be viewed as the finite element (FE) portion of the calculation, while the contact algorithm is composed of steps (1.7)-(1.9).

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.

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, step (1.8). 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, list of potential contact pairs are built in the neighborhood identification phase, then each contact node is checked to see if it penetrated a contact surface in this list of neighboring surfaces.

The neighborhood identification step requires a global search of the simulation domain which can require 30-50% of the overall PRONTO3D run time 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.

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 neighboring 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 [4].

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.

3. 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 elements (a 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 based on CHACO [5]. 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. 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 will have the same result regardless of which processor it is on. Thus, each processor will have a "carbon" copy of the boundary nodes that can be independently integrated through time.

Because the mesh connectivity does not change during the simulation 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 [6], [7], [8], [9], and [10].

4. CONTACT ALGORITHM

4.1 Location Phase

In this section and the ones that follow, we will outline the different parts of contact algorithm and discuss the strategies for implementing them in parallel. The location phase is the most time consuming part of the contact detection algorithm. For a detailed account of the general formulation and numerical treatment of finite deformation contact problems in finite elements, see [11]. A more detailed description of efficient schemes for spatially sorting and searching for contacts can be found in [4].

4.1.1 Serial Location Phase. 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 search 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 determining the contact nodes that fall within a capture box. The *points-in-box search* algorithm developed by Swegle and coworkers [12] bounds the space occupied by a contact surface at its known location and at its predicted location. Any contact nodes inside the capture box will be considered for potential contact with the contact surface. The great advantage of using the points-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.

4.1.2 Parallel Location Phase . The most commonly used approach for performing the location phase in parallel [7], [8], [9], [10] 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 volume. Global communication is performed to exchange the bounding volume's extent with all processors. Then, each processor sends contact surface and node information to all processors with overlapping bounding volumes. Though simple in concept, this approach will not efficiently load balance the contact detection for general problems.

A better load balance has been obtained by using separate decompositions for the contact detection and the finite element analysis [6]. A three-dimensional grid is used divide the entire computational domain into a set of buckets, with the nodes and surfaces being distributed to the buckets based on their location [13], [14], [15]. The nodes from a given bucket are combined with the nodes from neighboring buckets to form a node pool for contact checking.

For the parallel implementation of the bucket sort, Hoover, et al. [6] divided the coarse grid along one dimension into slices, one for each processor. 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 [6] suffered from load imbalance and did not scale to large numbers of processors.

Since the FE decomposition described in Section (3.) load balances the entire FE mesh (both interior and surface nodes), it will not in general load balance the contact surfaces and nodes. 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, which change with time. To achieve a better load balance than the static FE decomposition, we have devised a dynamic or adaptive decomposition technique.

Because we use a different decomposition for contact detection than for the finite element calculation, we can optimize each decomposition independently. The key difference is that for the contact detection decomposition at each time step, we use a dynamic technique known as *recursive coordinate bisection* (RCB). We have found several advantages to this approach. First, and foremost, RCB load balances the contact detection calculation. Second, an RCB decomposition can be updated efficiently if it begins from a nearly balanced starting point. We use the RCB result from the previous time step, which, due to the explicit time-stepping algorithm, 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.

Before we can describe the dynamic decomposition algorithm, we will review in detail some aspects of unstructured communications and the RCB method.

4.1.3 Unstructured Communications. The parallel contact algorithm involves several unstructured communication steps. In these operations, each processor has some information it wants to share with a few 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. 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 2.

```
(2.1) Form vector of 0/1 denoting who I send to
(2.2) Fold vector over all $P$ processors
(2.3) Gather number of receives for my processor, $nrecvs$ = vector($q$)
(2.4) For each processor I have data for, send message containing size of the data
(2.5) Receive $nrecvs$ messages with sizes coming to me
(2.6) Send and receive $nrecvs$ messages
```

FIGURE 2.Unstructured communications.

In steps (2.1)- (2.3) each processor learns how many other processors want to send it data. In step (2.1) 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 in step (2.2) 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 [5].

In step (2.4) each processor sends a short message to the processors it has data for, indicating how much data each should expect. These short messages are received in step (2.5), and a processor can now allocate the appropriate space for all the incoming data and post receive calls.

4.1.4 Recursive Coordinate Bisectioning. The recursive coordinate bisectioning (RCB) algorithm was first proposed as a static technique for partitioning unstructured meshes [16]. 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 [17]. The subdomains produced by RCB are geometrically compact and well-shaped. The algorithm can be parallelized in a fairly inexpensive manner. Also, small changes in the geometry induce only small changes in the partitions. Most partitioning algorithms do not exhibit this behavior.

Figure 3 shows schematically how an 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 FE decomposition may scatter the points anywhere in the domain. Some processors in the FE decomposition may have many points, while others may have none.

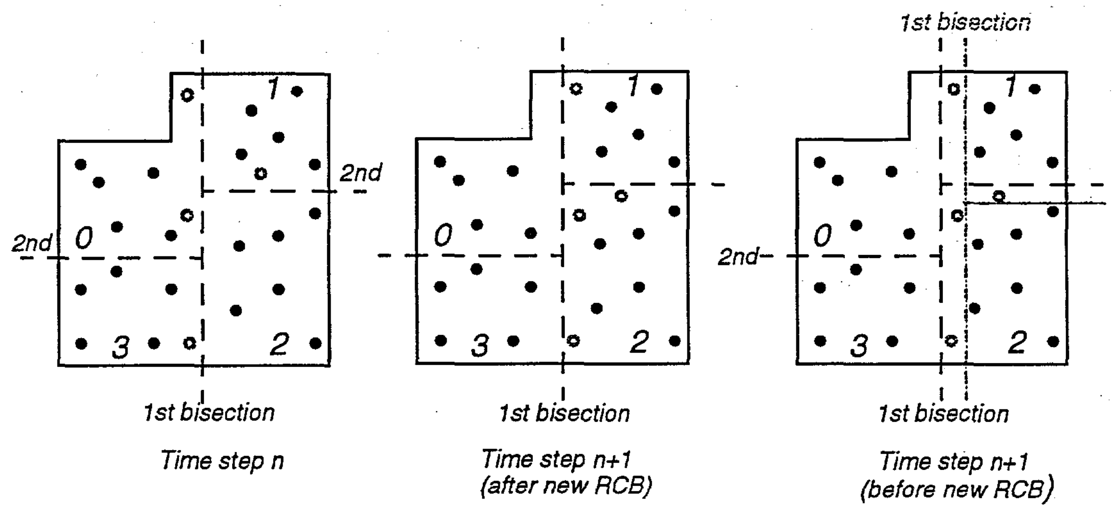

FIGURE 3.Recursive Coordinate Bisection

The first step in the RCB is to choose one of the coordinate directions, *x, y*, or *z*. for bisecting the domain. 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.

(4.1) Choose a coordinate axis *(xyz)*
(4.2) Position cut so as to partition points equally
(4.3) Send points that lie on far side of cut
(4.4) Receive points that lie on my side of cut
(4.5) Recurse

FIGURE 4.Steps used in RCB algorithm

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 4.

After the first pass through steps (4.1)-(4.4), 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 3 for a four-processor example. The final geometric sub-domain owned by each processor is a regular parallelepiped.

4.1.5 Details of the Dynamic Decomposition . Our parallel algorithm for contact detection is outlined in Figure 5. In step (5.1), 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. (This step is simply skipped on the first time step.) The above communication step involves unstructured communication as detailed in the Section 4.1.3. 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 (5.2) we perform the RCB decomposition as described in the previous section to rebalance the contact surfaces and nodes based on their current positions.

```
(5.1) Send contact data from FE decomposition to old RCB decomposition
(5.2) Perform parallel RCB to rebalance
(5.3) Share RCB cut info with all processors
(5.4) For all my surfaces
          If surface capture box extends beyond my RCB box
          Determine what other processors need it
(5.5) Send overlapping surfaces to nearby processors
(5.6) Find contacts within my RCB box
          Call points-in-box search routines
          Detailed contact check
(5.7) Send contact results to FE owners
```

FIGURE 5.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 (5.3) 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 [18].

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 (5.4), 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 (5.3). The information in this vector enables a processor to know the bounds of the RCB sub-domain owned by every other processor. In step (5.5), the data for overlapping contact surfaces is communicated to the appropriate processors.

Calling the serial routines at this point, step (5.6), enables the code to take advantage of the points-in-box sorting and searching features the serial routine used to efficiently find contacts.

4.2 Detailed Contact Check

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 detailed contact check considers the position 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. A complete derivation of the serial detailed contact check can be found in [4].

For the parallel implementation of the detailed contact check, we use the same routines as in the serial algorithm. The difference is that the detailed contact check is made from within the RCB decomposition. Thus, each processor will check the contact node-surfaces pairs that it owns for penetration.

The parallel contact detection problem, at this point, 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 (5.6).

At the end of the detailed contact check the 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 contact enforcement phase of the contact problem.

4.3 Enforcement of Contact Constraints

Several different approaches are available for contact enforcement. [9], [15], [19], [20]. Here, we will only consider the partitioned kinematic approach used in PRONTO3D [1], [4].

4.3.1 Serial implementation. For the contact enforcement, a predictor-corrector method is used. First, the location of contact surfaces and contact nodes, assuming no contacts is predicted and a detailed contact check is used to calculate a depth of penetration for each contact node into the contact surface.

A contact constraint is defined to removed the predicted penetration. The contact constraint is satisfied by simultaneously applying a contact force to the contact node and to the contact surface. The application of this penalty force will result in both surfaces moving. An iterative method is used determine the location of the interface (both surfaces) as a result of applying the penalty force. The movement of the interface surface is used to adjust the penalty force until the location of the interface surface and the penalty force converge.

4.3.2 Parallel implementation. For the parallel implementation, 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, so any penalty for load imbalance is small.

The existing serial algorithm for contact enforcement 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. The initial penalty force 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.

The penalty forces from each contact node in contact with a surface must be summed to the corners of the contact surface.Thus, each processor must receive penalty forces to be accumulated in the surface sum. The assembly of the total force acting on the contact surface, 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 can be used to correct the penalty force. One more communication is required to compute the acceleration of the contact surface using this new penalty force.

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 are small.

5. RESULTS

5.1 Measures of Parallel Computer Performance

Here we will review some measures of parallel computer performance.

We define the grind time, T_{grind}, as

$$T_{grind} = \frac{T_{exe}}{N_{elements} N_{cycles}} \quad (1)$$

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

We define the fixed-size speedup, S_f, as:

$$S_f = \frac{T_{m,1}}{T_{m,p}} \quad (2)$$

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 we define it as:

$$S_s = \frac{T_{m,1} p}{T_{m \times p, p}} \quad (3)$$

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

We define the parallel efficiency as

$$E = \frac{S}{p} \quad (4)$$

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 ratio of computation to communication work performed by each processor decreases and eventually, the communications will dominate the calculation time. Thus, for fixed problem size, there is a limit to how fast one can make parallel calculations run by increasing the number of processors.

The scaled parallel efficiency, $E_s = S_s/p$, indicates the efficiency of the parallel algorithm when the problem size increases in proportion to the number of processors. 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 work 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?

5.2 Brick Wall

This example considers a wall of bricks being hit by an elastic-plastic rod. The impact of the bricks by the rod scatters the bricks geometrically and unpredictably (Figure 6). A large number of contacts must be enforced during the early stages of this problem. 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 algorithm's time is spent searching for contacts.

Figure 7 shows the performance of the fixed-size brick wall problem. For this calculation, 8400 elements were used (54 elements were used to model each brick). The number of elements was held fixed while the number of processors increased. The calculation efficiency fell off when the number of processor exceeded 64. At this point each processor had only 130 elements.

Figure 8 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 per processor. The computational efficiency continued to increase as the number of processors increased.

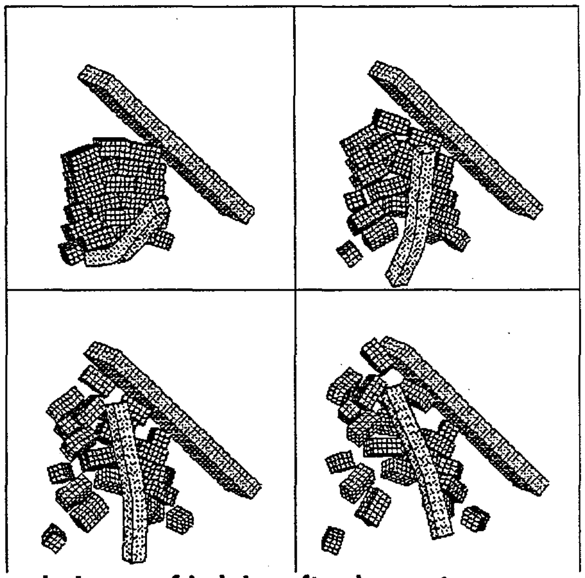

FIGURE 6.Deformed shape of bricks after impact.

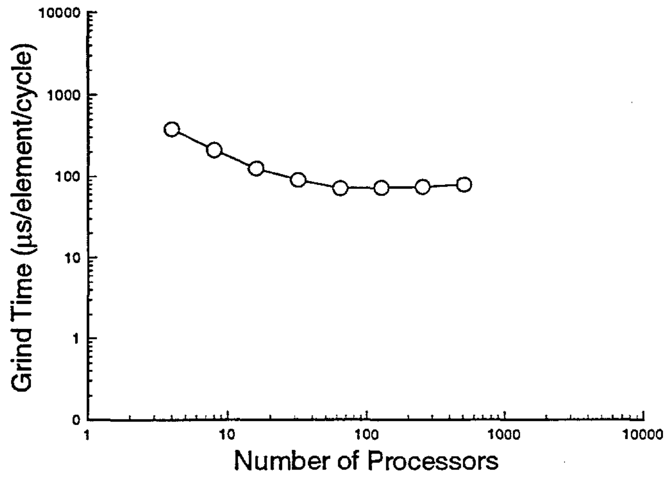

FIGURE 7.Grind time for the fixed-size brick wall problem for 8400 elements.

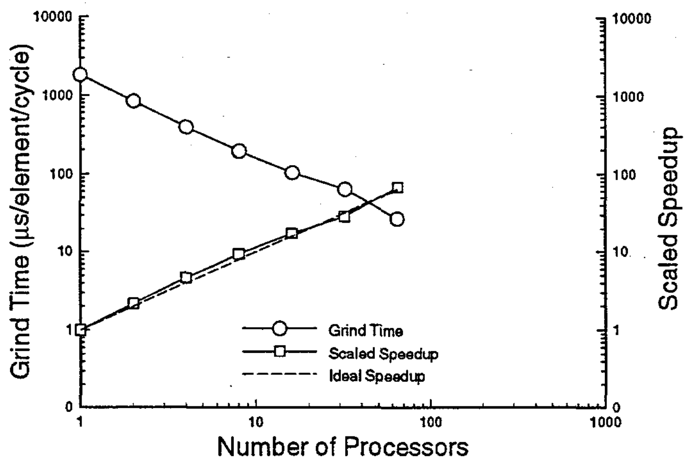

FIGURE 8.Grind time and scaled speedup for the scaled brick wall problem for 1890 elements per processor.

5.3 Can Crush

This example considers a simulation of a steel shipping container being crushed due to an impact with a flat inclined plate [4]. 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.

The performance of the fixed-sized problem is shown in Figure 9. For this calculation 7152 elements were used and both the container and wall were meshed to three elements thick, so roughly two-thirds of the elements are on a surface. The figure shows the average CPU time per time step for various numbers of Paragon processors from 4 to 1840. Both the finite element and contact 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.

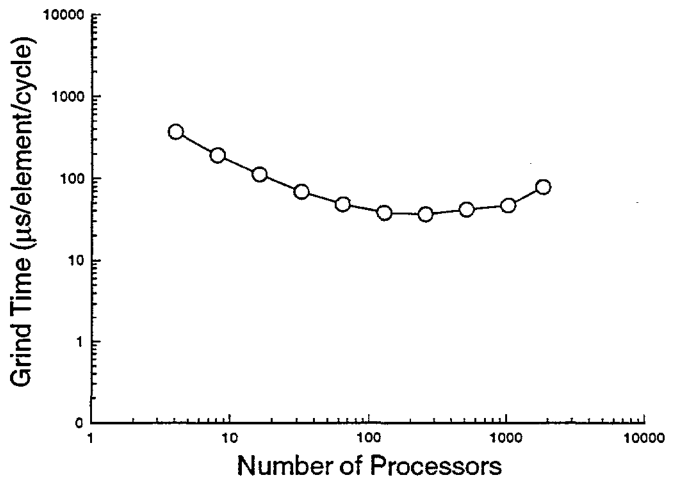

FIGURE 9.Grind time for the fixed-size can crush problem with 7152 elements.

Figure 10 shows the performance of the scaled can crush problem. In this simulation 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.

Figure 10 shows excellent scalability. Analysis of the timings reveals 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.

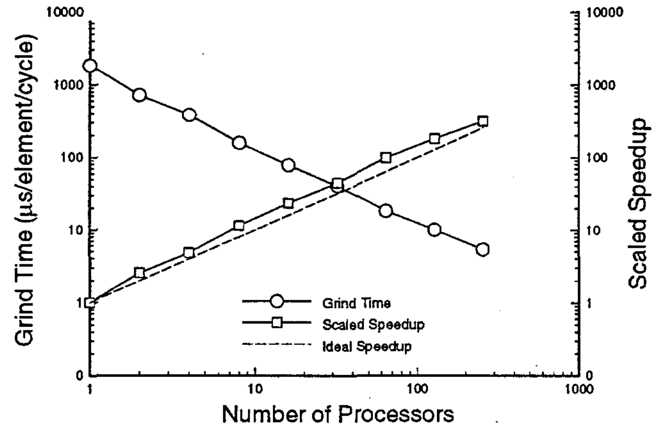

FIGURE 10.Grind time and scaled speedup for the scaled can crush problem with 1875 elements per processor.

There are several factors that may 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 reduces the work done in any one time step by the serial contact search portion of the contact algorithm (step (5.6) in Figure 5), since contact surfaces and nodes are not moving as far in a single time step, and the bounding box for each contact surface is expanded by the maximum amount that a contact node can move. In parallel, the bounding box is expanded only by the maximum that a node moves on a each processor. Thus, smaller 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.

Another reason for superlinear speedup is more subtle. 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 two 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-domain. 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 11 shows the CHACO and the RCB decompositions for the can crush problem. In each frame the subdomain is colored by the processor which owns in either the CHACO or RCB decomposition. Figure 11a–c show that the surface ownership in the RCB decomposition is dynamic. Figure 11d shows the element ownership in the CHACO decomposition, which is static. 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 11e and Figure 11f, the RCB decomposition at time t=1.6 ms and t = 3.2 ms is shown mapped to the undeformed shapes, to show more clearly how the decomposition changes with time.

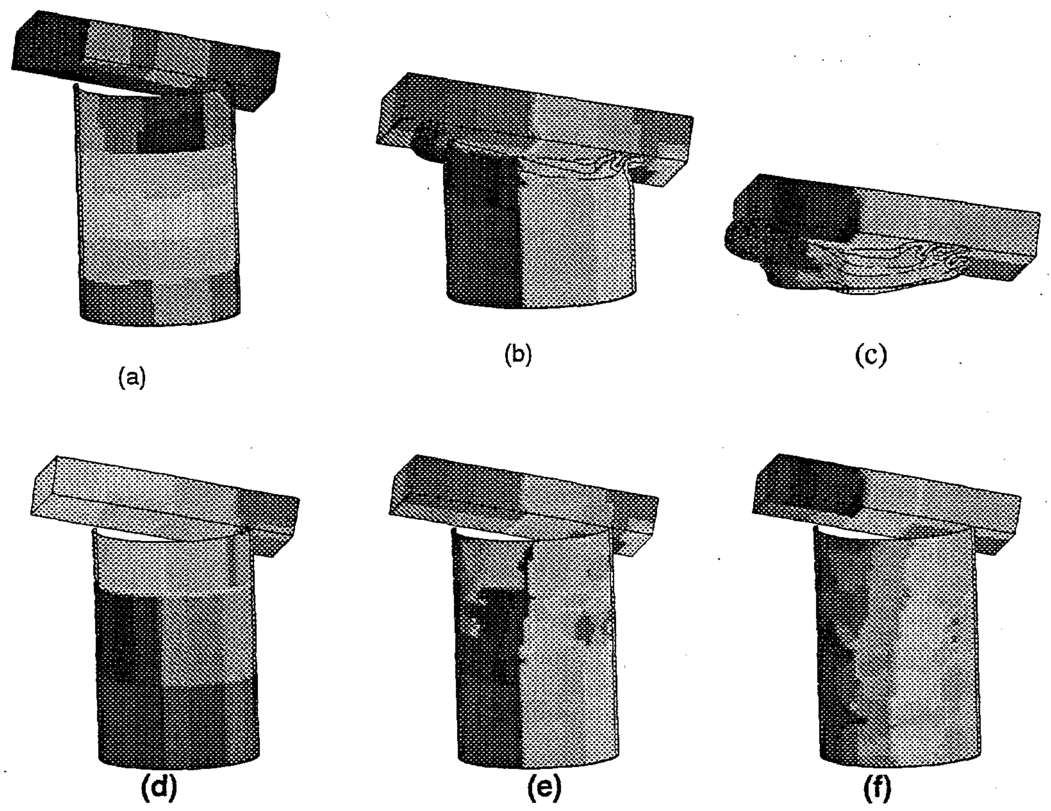

FIGURE 11.Decomposition generated by a) CHACO and d) RCB at time zero, RCB mapping for times b)t=1.6 ms and c) t= 3.2ms. The RCB decomposition mapped to the undeformed mesh is shown in (e) and (f).

6. SUMMARY

An efficient, scalable, parallel algorithm for treating material surface contacts in solid mechanics finite element programs has been implemented in a modular way. 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 for the contact portion of the calculation.

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. 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.

7. REFERENCES

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

2. Attaway, S. W., 1990, "Update of PRONTO 2D and PRONTO3D Transient Solid Dynamics Program," SAND90-0102, Sandia National Laboratories, Albuquerque, New Mexico, November, 1990.

3. Bergmann, V.L., 1991, "Transient Dynamics Analysis of Plates and Shells with PRONTO3D," SAND91-1182, Sandia National Laboratories, Albuquerque, New Mexico, September 1991.

4. 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.

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

6. 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.

7. 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.

8. 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.

9. Malone, J. G. and Johnson, N. L., 1994, "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.

10. Malone, J. G. and Johnson, N. L., 1994,"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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

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

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

18. 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.

19. 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.

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