A NEW PARALLEL ALGORITHM FOR CONTACT DETECTION IN FINITE ELEMENT METHODS*

Bruce Hendrickson† Steve Plimpton Steve Attaway Courtenay Vaughan David Gardner

Abstract

In finite–element, transient dynamics simulations, physical objects are typically modeled as Lagrangian meshes because the meshes can move and deform with the objects as they undergo stress. In many simulations, such as computations of impacts or explosions, portions of the deforming mesh come in contact with each other as the simulation progresses. These contacts must be detected and the forces they impart to the mesh must be computed at each timestep to accurately capture the physics of interest. While the finite-element portion of these computations is readily parallelized, the contact detection problem is difficult to implement efficiently on parallel computers and has been a bottleneck to achieving high performance on large parallel machines. In this paper we describe a new parallel algorithm for detecting contacts. Our approach differs from previous work in that we use two different parallel decompositions, a static one for the finite element analysis and dynamic one for contact detection. We present results for this algorithm in a parallel version of the transient dynamics code PRONTO-3D running on a large Intel Paragon.

1 Introduction

Transient dynamics models are often formulated as finite element simulations on Lagrangian meshes. Unlike Eulerian meshes which remain geometrically fixed as the simulation proceeds, Lagrangian meshes can be easily fitted to complex objects and can deform as objects change shape during a simulation. Prototypical phenomena that are modeled in this way include car crashes, and metal forming and cutting for manufacturing processes. Commonly–used commercial codes that simulate these effects include LS–DYNA3D, ABACUS, and Pam-Crash. PRONTO-3D is a DOE code of similar scope that was developed at Sandia [11].

A complicated process such as a collision or explosion involving numerous complex objects requires a large number of mesh elements to model accurately. 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 timesteps can be very computationally intensive, and so is a natural candidate for the power of parallel computers.

The finite-element (FE) portion of the computation within a single timestep can be parallelized straightforwardly. In an explicit timestepping scheme, each mesh element interacts only with the neighboring elements it is connected to in the FE mesh topology. If each processor is assigned a small cluster of elements then the only interprocessor communication will be the exchange of information on the cluster boundary with a handful of neighboring processors. A variety of algorithms and tools have been developed that optimize this assignment task. For PRONTO-3D we use a software package called Chaco [4] which partitions the FE mesh so that each processor has an equal number of elements and interprocessor communication is minimized. In practice, the resulting FE computations are highly load–balanced and scale efficiently (over 90%) 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.

It is important to note that 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 partition the FE mesh as a pre–processing step before the transient dynamics simulation is run. Similar FE parallelization strategies have been used in other transient dynamics codes [6, 8, 9, 10].

In most simulations there is a second major computation which must be performed each timestep. This is the detection of *contacts* between unconnected ele-

*In Proc. High Performance Computing '96

†Sandia National Labs, Albuquerque, NM 87185-1110. Email: [bah,sjplimp,swattaw,ctvaugh,drgardn]@cs.sandia.gov.

ments. For example, in Fig. 1, initial and 5 millisecond snapshots are shown of a simulation of a steel rod colliding with a brick wall. Contacts occur any time a surface element on one brick interpenetrates a surface element on another brick. These contacts impart forces to the impacting objects which must be included in the equations-of-motion for the interpenetrating elements. Thus, PRONTO-3D performs the following computations every timestep: (1) detect contacts, (2) compute contact forces, and (3) push-back the contacting elements so they no longer interpenetrate. Steps (2) and (3) are actually minor computations since at any one timestep only a small fraction of the elements are in contact. However, the contact detection in step (1) requires a global search of the simulation domain and can require 30-50% of the overall run time when PRONTO-3D runs on a vector machine like the Cray Y–MP. This is because, in principle, any two surface elements anywhere in the simulation domain can come in contact with each other during a given timestep. This is true even for surface elements on the same object, as when a car fender is crumpled in a collision. Efficient schemes for spatially sorting and searching lists of elements have been devised to speed this computation in the serial version of PRONTO-3D [3].

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. This is 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 who owns the node acquire information about all surfaces that are geometrically nearby. Even if we devise a global communication scheme or new decomposition technique that provides this information it must be a *dynamic* or adaptive method instead of static, since the set of nearby surfaces changes as the simulation progresses.

Given these difficulties, how can we efficiently parallelize the task of contact detection? The most commonly used approach [8, 9, 10] has been to use a single, static decomposition of the mesh to perform both FE computation and contact detection. At each timestep,

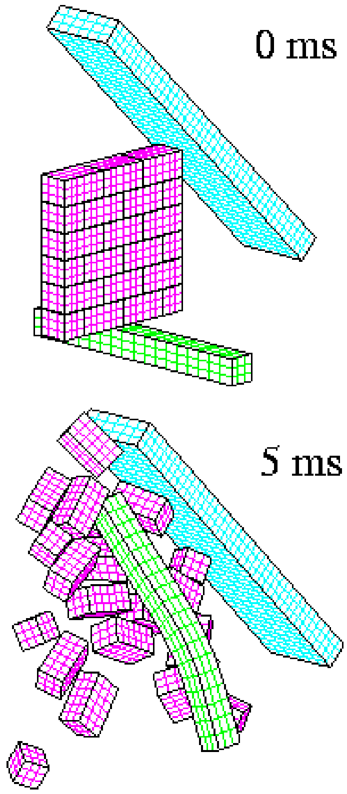

Figure 1: *Simulation of a steel rod hitting a brick wall.*

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 so that contact detection can be performed locally on each processor. Though simple in concept, this approach is problematic for several reasons. For general problems it will not load–balance the contact detection for the reasons given above. This is not as severe a problem in [10] because only meshes composed of "shell" elements are considered. Since every element is on a surface a single decomposition can balance both parts of the computation. However, consider what happens in Fig. 1 if one processor owns surface elements on 2 or more bricks. As those bricks fly apart, the bounding box surrounding the processor's elements becomes arbitrarily large and will overlap with many other processor's boxes. This will require large amounts of communication and force the processor to search a large fraction of the global domain for its contacts.

In this paper we describe a new strategy for contact detection which we have implemented in a version of PRONTO-3D developed for message–passing MIMD parallel computers such as the Intel Paragon and Cray T3D. An important aspect of our approach is that we use a different decomposition for contact detection than we use for the finite element calculation. This allows us to optimize each portion of the code independently. For contact detection we use a dynamic technique known as recursive coordinate bisection (RCB) to generate the decomposition anew at each timestep. We find several advantages to this approach. First, and foremost, since each processor ends up with the same number of contact nodes and surfaces, we can achieve nearly perfect load balance in the on–processor contact detection calculation. Second, the cost of performing an RCB decomposition is minimal if it begins with a nearly–balanced starting point. We use the result from the previous timestep, which will always be close to the correct decomposition for the current timestep. Third, the local and global communication patterns we use in our algorithm are straightforward to implement and do not require any complicated analysis of the simulation geometry. The price we pay for these advantages is that we must communicate information between the FE and contact decompositions at every timestep. Our results indicate that the advantage of achieving load balance greatly outweighs the cost of maintaining two decompositions.

We have recently become aware of independent work [6] which has some similarity to our approach. Like our technique, this approach uses a different decomposition for the contact detection than for the finite element analysis. In their method, they decompose the contact surfaces and nodes by overlaying a regular, coarse 3–D grid on the entire simulation domain. The coarse grid is then divided along one dimension into slices and each processor is responsible for contact detection within a slice. 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.

In the next section we provide some background material that will help explain our algorithm in §3. This is followed in §4 by some performance results from simulations using PRONTO-3D.

2 Background

Our 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 will 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 about the messages it will receive. We accomplish this with the approach sketched in Fig. 2.

(1) Form vector of 0/1 denoting who I send to
(2) Fold vector over all P processors
(3) $nrecvs$ = vector(q)
(4) For each processor I have data for,
    send message containing size of the data
(5) Receive *nrecvs* messages with sizes coming to me
(6) Allocate space & post asynchronous receives
(7) Synchronize
(8) Send all my data
(9) Wait until I receive my data

Figure 2: Parallel algorithm for unstructured communication for processor q.

In steps (1–3) each processor learns how many other processors want to send it data. In step (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 [2] in step (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 (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). 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 (7), each processor can now send its data. The processor can proceed once it has received all its data.

The recursive coordinate bisectioning (RCB) algorithm we use was first proposed as a static technique for partitioning unstructured meshes [1]. Although for static partitioning it has been eclipsed by better approaches, RCB has a number of attractive properties as a dynamic partitioning scheme which have been exploited by Jones and Plassmann [7]. The subdomains produced by RCB are geometrically compact and well–shaped. The algorithm can also be parallelized in a fairly inexpensive manner. And it has the attractive property that small changes in the geometry induce only small changes in the partitions. Most partitioning algorithms do not exhibit this behavior.

The collection of points we want to divide equally among P processors is the combined set of N contact

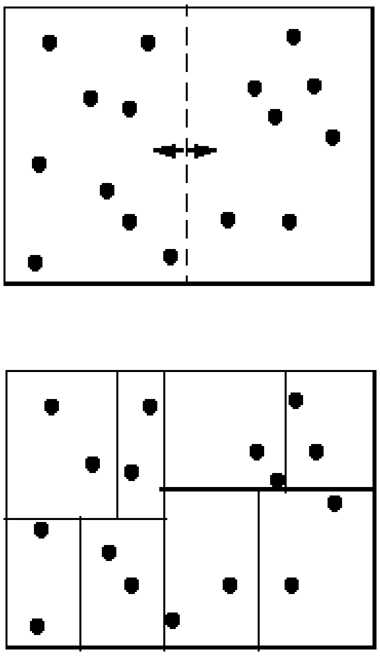

Figure 3: *Top: First cut of RCB decomposition. Bottom: Final partitioning for 8 processors.*

surfaces and nodes as shown in Fig. 3 for a 2–d example. For this operation we treat each surface as a single point. Initially each processor owns some subset of the points which may be scattered anywhere in the domain. The first step is to choose one of the coordinate directions, x, y, or z. We choose the direction for which the box bounding the points is longest, so that when we cut orthogonal to that direction, the resulting sub–domains will be 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 is equivalent to finding the median of a distributed set of values in parallel. We do this in an iterative fashion. First we try the 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, and likewise receives a set of points that lie on its side of the cut. These steps are outlined in Fig. 4.

After the first pass through steps (1–4), 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 Fig. 4 for an 8–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.

(1) Choose a coordinate axis (xyz)
(2) Position cut so as to partition points equally
(3) Send points that lie on far side of cut
(4) Receive points that lie on my side of cut
(5) Recurse

Figure 4: Parallel algorithm for recursive coordinate bisection.

3 Parallel Contact Algorithm

Our parallel algorithm for contact detection is outlined in Fig. 5. In step (1), the current position of each contact surface and node is communicated by the processor who owns and updated it in the FE decomposition to the processor who owned that surface or node in the RCB decomposition of the previous timestep. (On the first timestep this step is simply skipped.) This involves unstructured communication as detailed in the previous section. The purpose of this step is to give the RCB decomposition a starting point that is close to the correctly balanced answer, since the finite elements do not move far in any one timestep. In step (2) we perform the RCB decomposition as described in the previous section to rebalance the contact surfaces and nodes based on their current positions.

(1) Send contact data to old RCB decomposition
(2) Perform parallel RCB to rebalance
(3) Share RCB cut info with all processors
(4) For all my surfaces
        If surface extends beyond my RCB box
                Determine what other processors need it
(5) Send overlapping surfaces to nearby processors
(6) Find contacts within my RCB box
(7) Send contact results to FE owners

Figure 5: A parallel algorithm for contact detection.

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 (3) we communicate this cut information so that every processor has a copy of the entire set of cuts. This is done via an *expand* operation [2]. 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 (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 is done using the RCB vector of cuts created in step (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) the data for overlapping contact surfaces is communicated to the appropriate processors.

In step (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 PRONTO-3D contact detection routine to accomplish step (6). This enables the code to take advantage of the special sorting and searching features the serial routine uses 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 (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 element push–back.

In summary, steps (1), (5), and (7) all involve unstructured communication of the form outlined in Fig. 2. Steps (2) and (3) also consist primarily of communication. Steps (4) and (6) are solely on–processor computation. A fuller explanation of the details of this algorithm are given in [5].

4 Results

Fig. 6 shows the results of a PRONTO–3D simulation of a steel shipping container being crushed due to an impact with a flat inclined wall. The front of the figure is a symmetry plane; actually only one half of the container is 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 parallel contact algorithm.

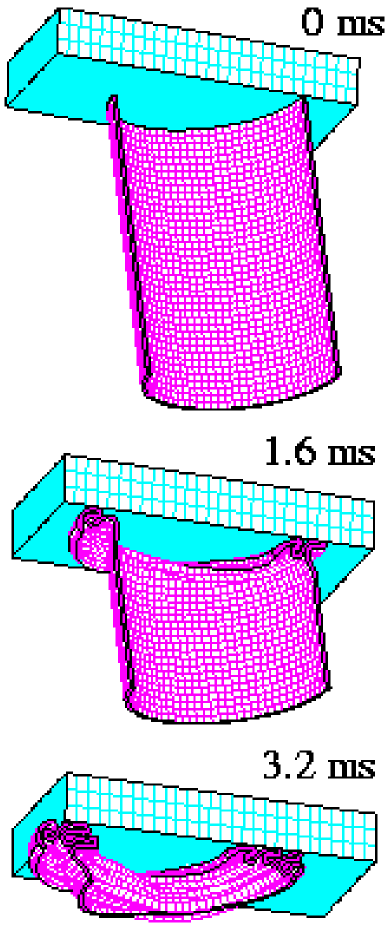

Figure 6: Simulation of a crushed shipping container from initial impact to final state after 3.2 milliseconds.

The first set of timing results we present is for a fixed–size problem geometry containing 7152 finite elements. Both the container and wall were meshed 3 elements thick, so roughly 2/3 of the elements are on a surface. Since each surface element contributes both a surface and node, there were about 9500 contact surfaces and nodes in the problem. The average CPU time per timestep for simulating this problem on various numbers of Intel Paragon processors from 4 to 1840 is shown in Fig. 7. Whether in serial or parallel, PRONTO–3D spends virtually all of its time in two portions of the timestep calculation — FE computation and contact detection. For this problem, both portions of the code speed–up adequately on small numbers of processors, but begin to fall off when there are only a few dozen elements per processor.

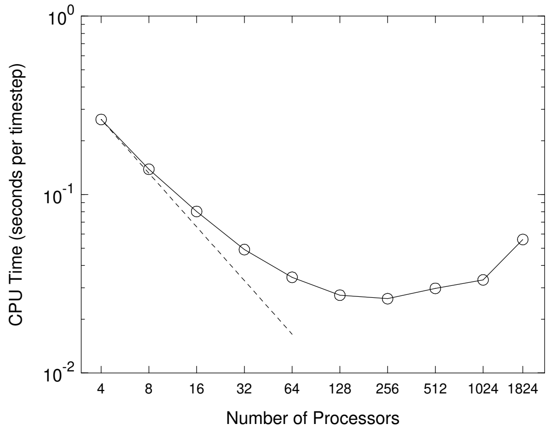

Figure 7: Average CPU time per timestep to crush a container with 7152 finite elements on the Intel Paragon. The dotted line denotes perfect speed–up.

Fig. 8 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 is 480,000 elements on 256 processors.

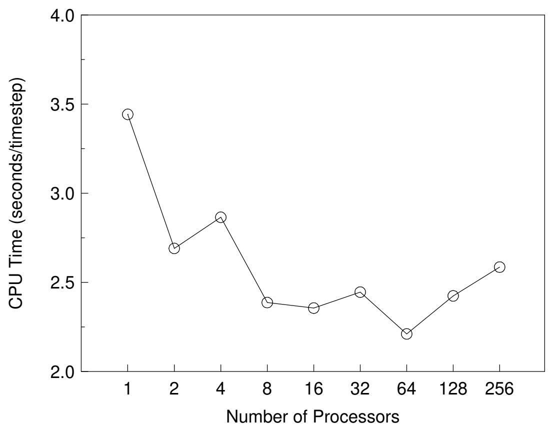

Figure 8: Average CPU time per timestep on the Intel Paragon to crush a container meshed at varying resolutions. The mesh size is 1875 finite elements per processor at every data point.

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 horizontal line 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. 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 timestep size is reduced as the mesh is refined. This actually reduces the work done in any one timestep by the serial contact search portion of the contact algorithm (step (6) in Fig. 5), since contact surfaces and nodes are not moving as far in a single timestep. More generally, the number of actual contacts that occur in any given timestep will not exactly double just because the number of finite elements is doubled.

5 Conclusions

The chief advantages of the parallel contact detection algorithm we have proposed are as follows:

(1) The contact surfaces and nodes are nearly perfectly spread across processors, ensuring that the contact detection is load–balanced.

(2) The RCB decomposition technique takes advantage of the fact that the partitioning does not change dramatically from one timestep to the next.

(3) The parallel code can use the same single–processor routine used in the original serial code to perform the actual work of contact detection.

The chief disadvantage of our method is that we must communicate data back–and–forth between the FE and RCB decompositions each timestep. In practice we observed this to be a very minor cost. Almost all of the time in the parallel contact detection was spent performing the RCB decomposition and in the on–processor contact detection effort. There is also a memory cost in our method for the contact surface and node data to be duplicated by the processors that store it in the RCB decomposition. This has not been a major bottleneck for us because the duplication is only for surface elements and because we are typically computationally bound, not memory bound, in the problems that we run with PRONTO–3D.

Acknowledgements

We benefited from helpful discussions about the parallel contact algorithm with David Greenberg and Rob Leland. Martin Heinstein provided insight into the sequential contact detection algorithm in PRONTO–3D.

References

[1] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, C-36 (1987), pp. 570–580.

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

[3] M. W. Heinstein, S. W. Attaway, F. J. Mello, and J. W. Swegle, *A general–purpose contact detection algorithm for nonlinear structural analysis codes*, Tech. Rep. SAND92–2141, Sandia National Laboratories, Albuquerque, NM, 1993.

[4] B. Hendrickson and R. Leland, *The Chaco user's guide: Version 2.0*, Tech. Rep. SAND94–2692, Sandia National Labs, Albuquerque, NM, June 1995.

[5] B. Hendrickson, S. Plimpton, S. Attaway, C. Vaughan, and D. Gardner, *A new algorithm for parallelizing the detection of contacts in finite element simulations.* In preparation.

[6] C. G. Hoover, A. J. DeGroot, J. D. Maltby, and R. D. Procassini, *Paradyn: Dyna3d for massively parallel computers*, October 1995. Presentation at Tri–Laboratory Engineering Conference on Computational Modeling.

[7] M. Jones and P. Plassman, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[8] G. Lonsdale, J. Clinckemaillie, S. Vlachoutsis, and J. Dubois, *Communication requirements in parallel crashworthiness simulation*, in Proc. HPCN'94, Lecture Notes in Computer Science 796, Springer, 1994, pp. 55–61.

[9] G. Lonsdale, B. Elsner, J. Clinckemaillie, S. Vlachoutsis, F. de Bruyne, and M. Holzner, *Experiences with industrial crashworthiness simulation using the portable, message-passing PAM-CRASH code*, in Proc. HPCN'95, Lecture Notes in Computer Science 919, Springer, 1995, pp. 856–862.

[10] J. G. Malone and N. L. Johnson, *A parallel finite element contact/impact algorithm for nonlinear explicit transient analysis: Part II – parallel implementation*, Intl. J. Num. Methods Eng., 37 (1994), pp. 591–603.

[11] L. M. Taylor and D. P. Flanagan, *Update of PRONTO–2D and PRONTO–3D transient solid dynamics program*, Tech. Rep. SAND90–0102, Sandia National Laboratories, Albuquerque, NM, 1990.