Tinkertoy Parallel Programming: Complicated Applications from Simple Tools*

Bruce Hendrickson[†] ***and Steve Plimpton***[†]

Abstract:

Developing parallel software for unstructured problems continues to be a difficult undertaking, particularly for distributed memory machines. Framework and library support are limited for non-standard applications and developers are often forced to code from scratch. This is particularly true for complex, unstructured applications. In this paper, we show that this needn't always be the case. We describe a set of simple primitives which can be combined to provide solutions to a variety of unstructured parallel computing problems. Specifically, we show how a small set of tools can yield efficient parallel algorithms for particle modeling, crash simulations and transferring data between two independent grids in multiphysics simulations. The use of such tools allows the application developer to program at a higher level without sacrificing performance.

1 Introduction

As has long been the case, the greatest impediment to the use of parallel computers is the difficulty of writing parallel software. This difficulty is particularly acute for distributed memory computers. One way to reduce this burden is to build upon existing parallel tools and utilities. The construction of appropriate parallel libraries has been a major focus of activity within the parallel computing community. A key

*This work was funded by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research and performed at Sandia, a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the U.S. DOE under contract number DE-AC-94AL85000.

[†]Sandia National Labs, Albuquerque, NM 87185-1110. Email {bah, sjplimp}@cs.sandia.gov.

challenge in this activity is in identifying the most appropriate abstraction level for the library interface. At the lowest level, message passing standards like MPI, PVM and Open-MP are robust and portable interfaces. But they provide only very primitive functionality, leaving a huge burden to the programmer. At the other extreme, some high level frameworks reduce the effort of the programmer to mere script writing. This allows for rapid development of parallel codes, but it is not a panacea. Good frameworks are an enormous undertaking to develop and generally support only a very limited class of applications. Users with non-mainstream needs can find frameworks to be too restrictive and limited.

Middle ground between these two extremes can be found in the many parallel utility libraries for linear algebra (e.g. [15, 1]), partitioning (e.g. [6, 16]) and other common kernel operations. Although we personally prefer this library approach to parallel software development, it is not without its problems. A key challenge for the developers of parallel libraries is the design of easy to use and efficient interfaces. For instance, a linear algebra library which presumes the matrices are assigned to processors in a complicated manner can be difficult for applications programmers to use. Another important question is the range of functionality that can be captured by libraries. Can libraries be flexible and powerful enough to address complex or nontraditional applications?

We believe the answer is yes, and in this paper we show how a few simple parallel primitives can be easily combined to build efficient, high level functionality. The primitives that we use are sufficiently simple to obviate the interface challenges. But, as we will show, their combination can provide efficient solutions to complex problems. Specifically, we provide parallel algorithms for the following three problems in scientific computing, which are described in more detail later.

1. In molecular dynamics and smoothed particle hydrodynamics, physical systems are modeled as a set of particles interacting via short-range forces. A variety of parallel algorithms have been devised for such systems, but dynamic load balancing challenges remain.

2. Crash simulations, like virtual car safety studies, are built from two principle computational operations. First is a finite element analysis to determine the stresses, strains and material response of the entities in the simulation. Second is the detection of grid intersection, which identifies entities that have come in contact with each other. These two computational steps have quite different load balancing needs, which makes the development of scalable parallel crash algorithms difficult.

3. When performing multiphysics computations, it is sometimes necessary to interpolate data from one grid onto a second grid. There are efficient serial algorithms for this problem, but they require an understanding of the geometric intersection properties of the two grids. Specifically, given a node in the second grid, determine which (if any) element of the first grid it lies within. In parallel, when the two grids are partitioned independently among processors, this determination can be difficult to accomplish.

In §2 we will introduce a set of simple parallel tools which will prove sufficient to parallelize these complex applications. Details of the parallel algorithms will be presented in §§3–5.

2 Parallel Primitives

To address the applications described above, we will rely on several simple parallel kernel operations. For simplicity of use, we want these kernels to have simple interfaces, but as we will show in the subsequent sections, they have sufficient power when combined properly to solve complex problems.

Primitive I. Recursive coordinate bisectioning (or RCB) is an old and unsophisticated load balancing strategy first proposed by Berger and Bokhari [2]. It works on entities in a geometric domain by recursively slicing the domain orthogonal to a coordinate axis, creating P partitions with an equal number (or weight) of entities contained within each. Although not a particularly effective static partitioning method, this simple algorithm has a number of under-appreciated features, particularly when used as a dynamic load balancer.

• It is fast and efficient to parallelize. (See, e.g. [5] or [10]).

• The boundary of each subdomain is a rectangular parallelepiped.

• The entire partition can be concisely described by the set of $P - 1$ cutting planes.

• New points (and extended entities) can be added to the partition efficiently by following the recursive sequence of cuts.

• Small changes in the locations of entities induce only an incremental change in the partition.

All of these properties will prove to be useful to us in the applications discussed below.

Primitive II. A processor which owns a set of entities needs to **identify all other entities which interact** with those it owns. In the general case of arbitrary decompositions, this can be a challenging problem. In the case where the interaction pattern is known, but ownership of entities is dynamic, an efficient, general-purpose algorithm has been devised by Pınar and Hendrickson [8]. However, for the common situation in which interaction is a function of geometric proximity, this neighbor identification problem is greatly facilitated by the recursive coordinate bisectioning decomposition. For each entity a processor owns, construct a bounding box which is large enough to contain all the possible other entities it overlaps. By following the sequence of RCB cuts, identify all the processors that could own items overlapped by this entity. Then share the relevant information with all of these processors.

Primitive III. A common situation in unstructured parallel computations is that each processor has a (sparse) set of messages to send to a known set of recipients, but does not know what or from whom it will receive. **Determining who I will receive from given the knowledge of who I will send to** is a key step. An efficient protocol to complete this communication operation is essential. We address this need by the algorithm in Fig. 1.

In steps (1)–(3) of this algorithm, the number of messages being sent to each

```
(1)  Form $P$-length 0/1 vector marking who I send to
(2)  Reduce-scatter 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 all my data
```

Figure 1. *A parallel algorithm for unstructured communication for processor q.*

processor are summed, so each processor learns the number of messages it will receive. With this information, in steps (4) and (5) each processor can tell its recipients about the data it wants to send. The data can then be exchanged in steps (6)–(9).

3 Particle Simulations

Molecular dynamics simulations are commonly used to model the mutual interactions of large numbers of atoms or molecules to study biological systems or material properties. Typically, classical Newtonian physics is employed. The non-Coulombic forces are of short range and so can be modeled by particles interacting only with near neighbors. Smoothed particle hydrodynamics and related mesh-free methods are used to simulate fluid motion or solids undergoing deformation. Again, particles only interact if they are geometrically near each other.

As particles move, the set of interacting partners changes dynamically, which can lead to load imbalance. The most efficient parallelizations of large-scale simulations utilize a spatial decomposition of the domain. That is, each processor is responsible for a region of space, and handles the calculations associated with all the particles currently in its region. Among the possible variants of spatial decompositions, several authors have used recursive coordinate bisectioning. Pilkington and Baden used RCB as one method for performing SPH simulations [7]. Plimpton, et al. used RCB for the more complex version of SPH in which the particle's interaction spheres can grow dynamically, and also for coupling SPH with structures simulations [10]. Srinivasan, et al. have applied RCB to the parallelization of molecular dynamics simulations [13].

The basic structure behind each of these parallelizations is broadly the same and fairly straightforward. Here we describe [10] to emphasize the utilization of our primitives. Particles whose interaction region extends beyond their processor's subdomain need to be communicated to the relevant processors. This determination can be facilitated by the simple geometry of the RCB subdomains and by the ease of determining which processor's subdomains intersect a bounding box around the particle's interaction region. At this point a processor knows what data it has to send, but not what it needs to receive, so the unstructured communication kernel is required. As particles move about, the load may need to be periodically rebalanced. The incremental nature of RCB ensures that a new decomposition will be similar to the current one, so little data will need to be exchanged before continuing.

This basic algorithm can be enhanced in several ways. One worth noting is to employ persistent data structures (commonly called neighbor lists in molecular dynamics) to preserve the set of possible interaction pairs for multiple timesteps. By greatly reducing the work required to identify interactions, this can significantly improve overall performance. More details can be found in the aforementioned references and in Plimpton [12].

4 Crash Simulations

Transient dynamics simulations are widely used to model low speed crash and impact phenomena. They are most commonly solved using explicit methods on Lagrangian grids. A prototypical example is the virtual crash-worthiness analyses performed by car manufacturers. The car is modeled by a finite element mesh which deforms upon impact. As the car bumper deforms, it can hit the radiator which will induce new physics which must be captured in the simulation. In the computation of the deforming mesh, this *contact* is observed when a mesh node passes through a face of a finite element. When this happens, new forces must be added into the simulation to model the interaction between the contacting objects. The two dominant operations in crash simulations are the finite element analysis and the detection of contacts.

The parallelization of unstructured grid finite element simulations is a well studied problem. The basic idea is to divide the mesh among processors in such a way that the inter-processor boundaries are kept small. This minimizes the amount of communication required in the explicit updates. Although the mesh will move, if its topology doesn't change then a single decomposition can be used for the duration of the simulation.

Contact detection has very different computational characteristics. Contact is fundamentally a geometric property, and topologically distant parts of the mesh can come into contact. The contact problem changes dynamically as the calculation proceeds. These properties argue for a dynamic, geometric partitioning strategy for contact detection. Here, we describe the use of recursive coordinate bisectioning for this problem. We have implemented our approach in the PRONTO code [14]. Our parallel contact detection algorithm is sketched in Fig. 2 and described in greater detail in [10, 3].

In steps (1) and (2) we are updating the recursive coordinate decomposition from the previous timestep. The incremental nature of RCB keeps this operation efficient. In step (3) we exploit the concise description of an RCB decomposition. Recall that a contact occurs when a mesh node passes through a face. Unlike nodes, faces have finite extent, and so can intersect several RCB subdomains. In step (4) we make sure that all each processor knows about all the faces which

```
(1)  Send contact data to old RCB decomposition
(2)  Update RCB to rebalance
(3)  Share RCB cut info with all processors
(4)  For all my faces
         If face extends beyond my sub-domain
             Determine which processors it overlaps
(5)  Send overlapping faces to nearby processors
(6)  Find contacts within my sub-domain
```

Figure 2. *A parallel algorithm for contact detection.*

intersect its subdomain and so might be in contact with its nodes. This step exploits the geometric simplicity of RCB subdomains, and the ease of determining which subdomains intersect a bounding box around the surface. In step (5) we utilize the unstructured communication primitive to exchange information efficiently. Finally, in step (6) each processor can invoke the (quite complex) serial code for contact detection as a local computation.

Note that with our approach, the contact detection operation uses a completely different parallel decomposition than the finite element calculation. So at each timestep, data is transferred back and forth between these two decompositions. Within the same code we have parallelized a version of smoothed particle hydrodynamics to allow the simulation of fluid/structure interactions. Our SPH parallelization uses yet a third decomposition, also based upon recursive coordinate bisectioning as described in §3.

To test the performance of our approach to contact detection we constructed a set of problems of varying sizes which were all based upon the geometry depicted in Fig. 3. This model depicts a tilted aluminum block moving downwards at high velocity and crushing an aluminum shipping container. As the container crumples, complex folding and buckling behavior is exhibited, which provides a stringent test of the contact detection routines. The model exploits bilateral symmetry.

We ran a set of these models on varying numbers of processors in such a way that the number of finite elements per processor remained a constant 3800. We ran the code on the ASCI Red, Intel parallel computer at Sandia National Labs. At the time these runs were done, ASCI Red had 200 Mhz Pentium Pro processors and a proprietary communication network with 10–20 microsecond latencies and bandwidth of about 300 megabytes per second. Our results are depicted in Fig. 4 as elapsed time per timestep. The lower curve is finite element computation time; the middle curve includes both finite element and contact detection time; and the upper curve is total CPU time. The data show almost perfect scalability out to 3600 processors.

One additional implementation detail is worth mentioning. As with neighbor lists which were mentioned in §3, the use of persistent data structures enhances the performance of our contact detection routine. Instead of performing a complete search for contacts each timestep, we can keep a list of face/node pairs which are

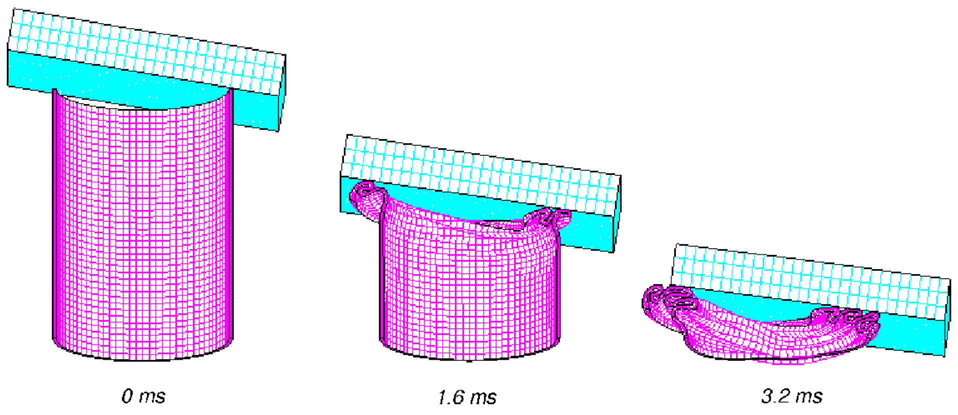

Figure 3. *Simulation of a crushed shipping container from initial impact to final state after 3.2 milliseconds.*

near each other and only search in within this list. Every few timesteps the list needs to be regenerated via a full contact search. This idea consumes space, but in our experiments it halved the average time for contact detection [9].

Prior to our work, no parallelization of these types of computations had exhibited scalability beyond a few tens of processors. This parallel code has enabled a number of simulations which were previously intractable, a few of which are described in [10].

5 Grid Transfer

When performing multi-physics simulations like fluid-structures interactions, a key operation is transferring information between the different physics models. For instance, the fluid will exert a force on the structure, and the deformation of the structure will change the geometry for the fluid. Another example in which the intersection between the simulations is 3D instead of 2D arises in thermal-structural analysis. The temperature will effect the material properties, but simultaneously the material properties effect the thermal conductivity. For sophisticated simulations the different physical phenomena may be modeled with different meshes, each of which may be adapting over time.

To transfer information between physics modules it is necessary to determine the geometric intersection between the grids. Specifically, to determine the temperature of a node of the structures mesh, you must first determine the thermal finite element it lies inside, and then do interpolation from the nodes of that thermal element. Various serial algorithms for this geometric kernel have been proposed.

However, parallelization of this operation is difficult when the two meshes are distributed across processors. If the simulations are adaptive then the meshes will be periodically redistributed to preserve load balance. Thus, no assumptions can be

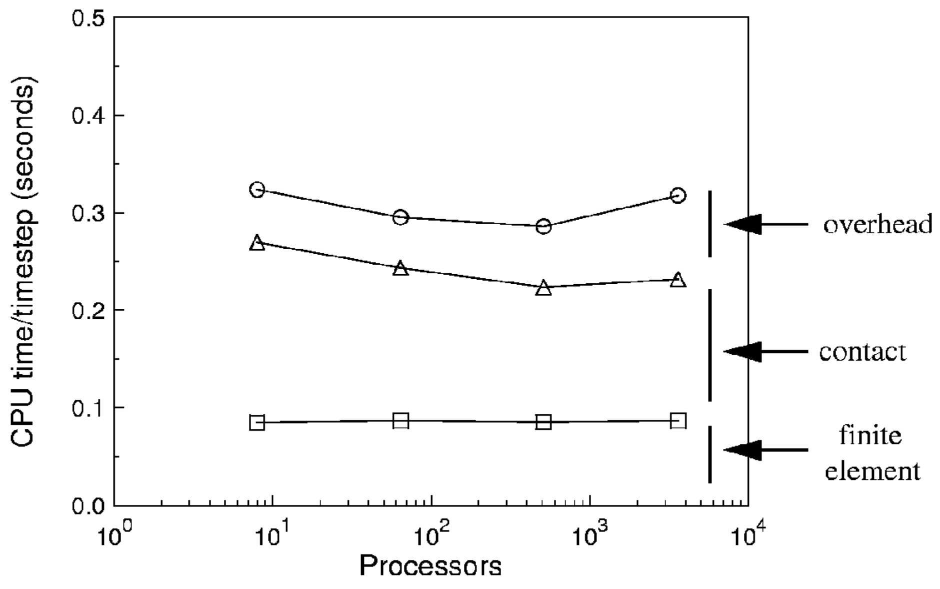

Figure 4. *Average CPU time per timestep on the ASCI Red computer to crush a container meshed at varying resolutions.*

made about which processor owns the thermal element that a particular structural node needs to know about.

We have developed a parallel *rendezvous* algorithm for this problem which uses recursive coordinate bisectioning as an intermediate decomposition. The algorithm is sketched in Fig. 5. More details can be found in [11].

```
(1)   Compute box that bounds thermal & structural mesh intersection.
(2)   Create rendezvous decomposition via RCB on the thermal mesh.
(3)   Send element geometry from thermal to rendezvous decomposition.
(4)   Find which rendezvous processor's subdomain has each structural node.
(5)   Send node geometry from structural to rendezvous decomposition.
(6)   Clone thermal elements which overlap into nearby RCB sub-domains.
(7)   Find which thermal element each structural node is inside.
(8)   Send element/node pairings from rendezvous to thermal decomposition.
(9)   Interpolate solutions from thermal nodes to structural nodes.
(10)  Send structural node solutions from thermal to structural decomposition.
```

Figure 5. *A parallel algorithm for grid transfer.*

In steps (1) and (2) we create a new decomposition for the problem using RCB.

We send the geometric information from both the thermal and structural meshes to this rendezvous decomposition in steps (3)–(5), The determination of where to send data exploits the compact representation of the RCB decomposition and the ease of adding new points to an existing decomposition. The data transfers use the unstructured communication primitive from Fig. 1. In step (6) we determine which processors need to know about possibly overlapping thermal elements. This mirrors step (5) of the contact detection algorithm in Fig. 2 and again exploits the ease of determining intersections with RCB. The geometric analysis is now reduced to a collection of independent serial computations which are performed in step (7). The output of this determination is sent back to the thermal decomposition in step (8). There, the actual numerical interpolations are performed and the answers are sent to the structural decomposition in step (10), using our unstructured communication kernel yet again.

As with the previous two examples, performance of this algorithm can be enhanced by persistent data structures. One way in which this can occur is if the geometry does not change between two timesteps. In this case, the element/node pairings of step (8) can be stored between timesteps. Steps (1-8) can then be skipped and only the interpolation and communication of steps (9-10) performed. An intermediate option can be exploited if the meshes move only a small amount. The full search in step (7) can be replaced by a faster, more localized search.

To test our algorithm, we used a simple problem whose size was easy to vary. We used two regular 3D hexahedral meshes of equal size, but slightly rotated and translated from each other. We decomposed the thermal mesh into regular 3D bricks, but we partitioned the structural mesh in a quasi-2D fashion into long columns. These choices of decompositions insure that all the data exchanges in the algorithm are irregular in nature and require each processor to send its data to many others.

Fig. 6 shows run times for a scaled-size problem running on 1 to 1024 processors of ASCI Red. The two meshes were successively doubled in different dimensions as processors were added so that there were always 8000 elements (and nodes) of each mesh per processor. On this plot perfect scalability of the algorithm would thus be a horizontal line. For the full algorithm the run time rises from about 0.3 second per timestep to nearly 1.0 second per timestep. Since the on-processor computations (search and interpolation) scale nearly perfectly (about 0.1 seconds/timestep), the growth in total time is due primarily to communication and to a lesser extent on the logarithmic dependence of the RCB operation on grid size. On the largest problems, an immense volume of data is moving between large numbers of processors in an irregular pattern; the communication bandwidth of even the ASCI Red machine eventually saturates and the algorithm runtime increases.

The lower curve in Fig. 6 is the time to perform the interpolation on timesteps when the meshes do not move relative to each other. Again there is a modest rise in runtime from 0.03 to 0.08 sec/timestep on the largest problems, due to the communication saturation. However, the most important result for these timings is that the grid transfer operation can be performed on extremely large meshes very rapidly on the ASCI Red machine – in less than 1/10 of a second on typical timesteps when searches are not necessary, and in under 1 second even when the

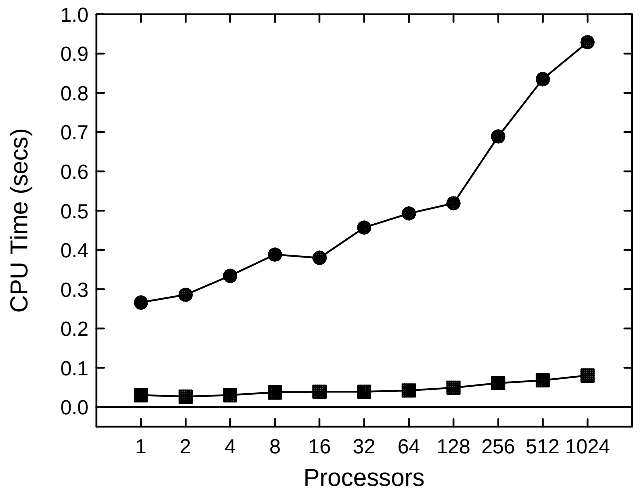

Figure 6. *CPU time for performing a grid transfer operation between pairs of varying sized grids. Each data point represents grids with 8000 elements per processor. Circles represent the full algorithm timing; squares are for just interpolation and subsequent solution communication.*

entire search operation must be redone (e.g. due to mesh adaptation or motion). This is likely to be a very small fraction of the time needed to solve the physics equations of interest each timestep on two separate multi-million element meshes.

6 Conclusions

We have shown how a simple set of parallel kernels can be used to efficiently parallelize several unstructured algorithms that arise in complex applications. By using these kernels, application developers are able to program at a higher level of abstraction, reducing development and debugging effort. Although low level implementation details are beyond the scope of this paper, we have written these kernel operations as a set of library routines with simple interfaces and a variety of options. For instance, our implementation of the communication operation from Fig. 1 allows for fixed or variable sized entities, data transfers to be buffered or not, and preservation of the computed communication pattern. It is our belief that well designed and constructed libraries are a key to addressing the software challenge on parallel machines.

The kernels described here have become part of the public domain Zoltan dynamic load balancing tool [4]. Zoltan adheres to our philosophy of providing sophisticated functionality via simple interfaces. Zoltan also provides significant additional capabilities for graph partitioning, data migration and other common parallel operations.

Bibliography

[1] S. Balay, W. D. Gropp, L. Curfman McInnes, and B. F. Smith, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11 - Revision 2.0.29, Argonne National Laboratory, 2000. http://www-fp.mcs.anl.gov/petsc/.

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

[3] K. Brown, S. Attaway, S. Plimpton, and B. Hendrickson, *Parallel strategies for crash and impact simulations*, Comp. Meth. Appl. Mech. Eng., 184 (2000), pp. 375–390. Invited paper.

[4] K. D. Devine, B. A. Hendrickson, E. G. Boman, M. M. St.John, and C. Vaughan, *Zoltan: A dynamic load-balancing library for parallel applications – user's guide*, Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999. http://www.cs.sandia.gov/Zoltan/.

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

[6] G. Karypis and V. Kumar, *Parmetis: Parallel graph partitioning and sparse matrix ordering library*, Tech. Rep. 97-060, Department of Computer Science, University of Minnesota, 1997.

[7] J. R. Pilkington and S. B. Baden, *Partitioning with spacefilling curves*, CSE Technical Report CS94–349, Dept. Computer Science and Engineering, University of California, San Diego, CA, 1994.

[8] A. Pinar and B. Hendrickson, *Communication support for adaptive computation*, in Proc. Tenth SIAM Conf. Parallel Proc. for Sci. Comput., March 2001.

[9] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner, *Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics*, in Proc. Supercomputing '96., November 1996.

[10] ——, *Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics*, J. Parallel Distrib. Comput., 50 (1998), pp. 104–122.

[11] S. Plimpton, B. Hendrickson, and J. Stewart, *A parallel rendezvous algorithm for interpolation between multiple grids*, in Proc. SC'98, ACM and IEEE, November 1998.

[12] S. J. Plimpton, *Fast parallel algorithms for short-range molecular dynamics*, J. Comp. Phys., 117 (1995), pp. 1–19.

[13] S. G. Srinivasan, I. Ashok, H. Jonsson, G. Kalonji, and J. Zahorjan, *Dynamic-domain-decomposition parallel molecular-dynamics*, Computer Physics Comm., 103 (1997), pp. 44–58.

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

[15] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid, *Official Aztec user's guide: Version 2.1*, Tech. Rep. SAND99-8801, Sandia National Labs, 1999. http://www.cs.sandia.gov/CRF/aztec1.html.

[16] C. Walshaw, M. Cross, and M. Everett, *Parallel dynamic graph-partitioning for unstructured meshes*, Mathematics Research Report 97/IM/20, Centre for Numerical Modeling and Process Analysis, University of Greenwich, London, SE18 6PF, UK, March 1997.