International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:4, No:8, 2010

Dynamic Meshing for Material Point Method Computations

Wookuen Shin, Gregory R. Miller, Pedro Arduino, and Peter Mackenzie-Helnwein

Abstract—This paper presents strategies for dynamically creating, managing and removing mesh cells during computations in the context of the Material Point Method (MPM). The dynamic meshing approach has been developed to help address problems involving motion of a finite size body in unbounded domains in which the extent of material travel and deformation is unknown a priori, such as in the case of landslides and debris flows. The key idea is to efficiently instantiate and search only cells that contain material points, thereby avoiding unneeded storage and computation. Mechanisms for doing this efficiently are presented, and example problems are used to demonstrate the effectiveness of dynamic mesh management relative to alternative approaches.

Keywords—Numerical Analysis, Material Point Method, Large Deformations, Moving Boundaries.

I. Introduction

THE Material Point Method (MPM, see, e.g., [1], [2]) is a hybrid Lagrangian/Eulerian computational approach for solving continuum mechanics problems, and its hybrid nature is such that it can be particularly effective in the context of large deformation, flow-like behavior of solids or granular materials [3], [4]. In such contexts it is not uncommon for motions to occur in partially unbounded domains, and it is often the case that the ultimate subset of the containing space that will be visited by moving material is unknown prior to the analysis. Because the MPM relies on a background mesh during the momentum and position update part of each time step, it is necessary to have some way of representing unbounded domains for these classes of problems.

The simplest mesh generation approach relies on using a statically allocated grid set up at the start of an analysis and then used throughout. This approach has two primary shortcomings: (i) it includes many unneeded cells (especially in cases where the domain occupied by particles is relatively small compared to that covering the trajectory of these particles); and (ii) the predicted domain may not bound the ensuing motion (this is especially the case of situations like landslide modeling where a priori prediction of the domain of the motion can be difficult). Including unneeded cells in an analysis has a direct computational cost and in some instances can create impractically large memory demands. For example, assuming an MPM node can be represented minimally by seven double-precision (8-byte) numbers (nodal velocity and force plus mass), a $500 \times 500 \times 500$ background mesh would require on the order of 6.5 GB of storage alone. Similarly, motions that exceed predetermined grid boundaries are likely to require the analysis to be halted and restarted. Thus computational efficiency, robustness, and feasibility are all affected by the grid creation scheme, and although global static meshing of a predetermined domain is the common approach used for the MPM (see, e.g., [2], [5], [6], [7], [8], [9], [10]), this model does not scale well to general situations.

Department of Civil & Environmental Engineering, University of Washington, Box 352700, Seattle, WA 98195-2700.

In this paper we present a dynamic meshing implementation capable of avoiding the cost of empty, unneeded cells while adding negligible additional mesh management overhead. This makes it possible to demonstrate the magnitude of the empty cell overhead costs in terms of both memory and computational time, and also provides a means for dramatically improved performance. The specific approach used in this paper is tuned for use with uniform rectangular background grids, but it could be extended to handle more general cases by means of appropriate location-to-cell mappings.

The general problem of handling sparse spatial data arises in many applications, and there have been a number of techniques and approaches developed over the years to efficiently represent and compute such data sets. Several basic mechanisms for managing sparse discrete blocks of space were introduced in 1979 in [11] in the context of database searching, and there has been substantial activity in recent years in computer graphics and image processing under the general heading of spatial hashing (e.g., [12]). A recent paper that is closely related to the present work considers the application of spatial partitioning in the context of Smooth Particle Hydraulic (SPH) fluid modeling [13]. This approach gives rise to similar particle-in-cell issues that arise in the MPM algorithm, and the spatial representation used in [13] has conceptual similarities to the approach presented in the present paper. The implementation details differ from the current approach, though, due to an orientation of that work around fixed-size domains in the context of GPU-based modeling.

This work has grown out of the development of an MPM-based approach to the modeling of landslides and debris flows, in which solid and liquid phases are treated as distinct, interacting fields [4]. Although the examples used in this paper do not consider multiple fields, the approach works well in that context.

The paper is organized as follows. A minimalistic introduction to the material point method will be given in section II to identify the basic algorithmic tasks and data required to perform the relevant computations. The concept of the employed dynamic meshing approach is developed in section III and its implementation is covered in section IV. A performance analysis of the proposed dynamic meshing approach based on representative numerical examples is given in section V. Summary and conclusions complete the paper.

II. MPM Overview

This section provides a simple MPM overview, the goal being to illustrate the basic algorithmic steps required to implement the method. More detailed descriptions of the method are available in, e.g., [1], [2], [5].

The MPM works by using a fixed (Eulerian) background grid to satisfy conservation of momentum (i.e., to solve the equations of motion) and a set of freely distributed material points which carry with them mass, constitutive behavior, and state information as they move from one cell of the background grid to another. Figure 1 provides an overview of a typical computational cycle, depicting the roles of the background grid nodes and the material points. As shown in Figure 1a, the equivalent nodal mass and internal forces corresponding to a configuration of material points can be computed via normal finite element shape function interpolation. In equation form this can be expressed as

$$m_i = \sum_{p=1}^{N_p} N_i(\boldsymbol{X}_p) m_p \tag{1}$$

for the nodal mass at node i and

$$\boldsymbol{f}_i = \boldsymbol{f}_i^{\text{ext}} - \sum_{p=1}^{N_p} \frac{m_p}{\rho_p} \boldsymbol{\sigma}_p \cdot \nabla N(\boldsymbol{X}_p) \tag{2}$$

for the equivalent force at node i. In these expressions, N_p represents the number of material points in a given cell, $\boldsymbol{X}_p$ represents the location of material point p, ρ_p is the mass density for material point p, $\boldsymbol{\sigma}_p$ is the current stress tensor at material point p, and $\boldsymbol{f}_i^{\text{ext}}$ is any externally applied nodal load at node i.

Given a set of assembled nodal forces and corresponding nodal masses, the equation of motion can be used to update the nodal velocities as shown in Figure 1b. A simple explicit equation relating unknown velocity, $\boldsymbol{v}_i^{k+1}$, at time step $k+1$ to the known velocity, $\boldsymbol{v}_i^k$, at time step k can be written as

$$\boldsymbol{v}_i^{k+1} = \boldsymbol{v}_i^k + \frac{\boldsymbol{f}_i}{m_i}\Delta t \tag{3}$$

in which $\Delta t = t^{k+1} - t^k$ is the time increment, and the nodal mass and force quantities are as defined above.

The background velocity field consistent with the updated nodal velocities can then be used to compute velocity gradients (and thus strain increments) and updated velocities at each material point (Figure 1c). In equation form this becomes

$$\dot{\boldsymbol{\epsilon}}_p = \frac{1}{2}\left(\nabla + \nabla^T\right) \sum_{i=1}^{N_n} N_i(\boldsymbol{X}_p) \boldsymbol{v}_i^{k+1} \tag{4}$$

for the strain rate and

$$\boldsymbol{v}_p^{k+1} = \boldsymbol{v}_p^k + \sum_{i=1}^{N_n} N_i(\boldsymbol{X}_p)\left(\boldsymbol{v}_i^{k+1} - \boldsymbol{v}_i^k\right) \tag{5}$$

for the particle velocity update. Here the subscripts p and i denote material point and nodal quantities, respectively, and N_n is the number of nodes in a given cell (typically, $N_n = 4$ in 2D and $N_n = 8$ in 3D).

The updated velocities are used to convect the material points to new locations as shown in Figure 1d (i.e., $\boldsymbol{X}_p^{k+1} = \boldsymbol{X}_p^k + \boldsymbol{v}_p \Delta t$), the strain increments are used to compute internal stress increments using appropriate constitutive relations, and the entire cycle begins again.

For purposes of this paper, a key point to note is that grid cells that do not contain material points do not participate in the computation in regards to either solving the equations of motion or updating particle states. In effect, in an expression like equation 1, the number of particles, N_p, in an empty cell is zero, and so there is no mass and additional computation is not necessary. Empty cells that have been instantiated thus need only a cursory check involving minimal computation, but it still can require significant effort and memory access to visit all instantiated cells when there are many of them.

Algorithmically, the computational cycle outlined above can be viewed as a set of iterations over material points, cells, and grid nodes, with cells managing the connectivity between grid nodes and material points. Thus step (a) in Figure 1 can be implemented as an outer loop over the grid cells with an inner loop over the material points in each cell. Step (b) consists of a loop over the grid nodes and doing an explicit time step update. Step (c) is once again a nested loop over cells and contained material points, and step (d) is a simple loop over all particles.

III. Method of Approach

The basic idea for the dynamic meshing approach is simple: in any given step of the computation, only background mesh cells containing particles are instantiated—the rest of the mesh is virtual. The trick with this variation on *lazy evaluation* [14] is to accomplish it without introducing so much overhead as to cancel out the benefits of reduced mesh cell counts. It is also desirable that the approach be relatively straightforward to understand and simple to implement.

The fundamental tasks for the cell management algorithm to accomplish are twofold: (i) new grid cells must be created (instantiated) as material points move into formerly unoccupied space; and (ii) existing grid cells should be deleted when they no longer contain material points. The key functionality supporting these tasks is mapping between a particle location and the associated grid cell—that is, given an arbitrary point in space, one needs to be able to determine quickly the mesh cell to which the point belongs.

Referring to Figure 1, it can be seen that within the MPM algorithm the creation task can be carried out as part of step (d). As each updated particle position is computed it can be determined if it is leaving its previous containing cell. If so, it can be reassigned to a new cell, either preexisting (when the particle location maps to an existing cell) or newly created (when the particle position does not map to an existing cell).

Similarly, the empty cell deletion process can be carried out as part of step (a). As each cell is visited to update nodal quantities, cells determined to be empty can be deleted.

In both these tasks it is necessary to manage the cell-node relationships properly, as well. In general, nodes are shared among cells, and so deletion and creation must be managed accordingly to avoid dangling pointers or memory leaks.

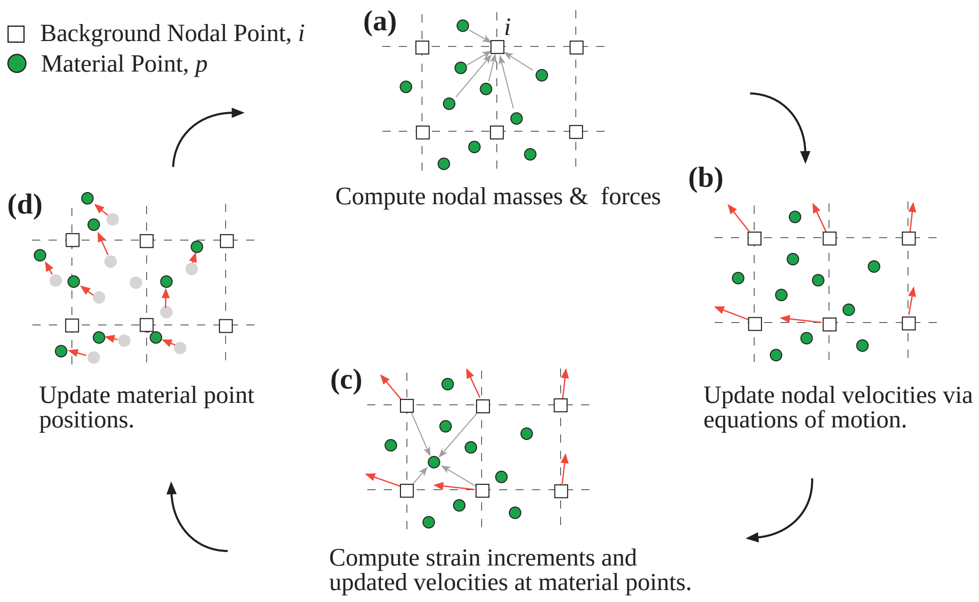
Background Nodal Point, i
Material Point, p
(a)
i
Compute nodal masses & forces
(b)
Update nodal velocities via equations of motion.
(c)
Compute strain increments and updated velocities at material points.
(d)
Update material point positions.

Fig. 1. The basic Material Point Method computational cycle.

The simplest, though brute-force approach for this task is a pointer array. However, returning to the 500×500×500 3D mesh, a pointer array would require 0.5 GB (1.0 GB) on a 32 bit (64 bit) architecture, and require repeated checking for null-pointers. An efficient alternative is to use a multi-dimensional, sparse, spatial array. Its characteristics and implementation details are discussed in the next section.

IV. Implementation

This section outlines the specifics of the implementation used to generate the results presented in the next section. It is convenient to consider initially a 1-D case before considering the full 3-D case, and so the presentation is composed of two subsections. This presentation focuses on the core computational aspects of the standard, explicit MPM algorithm—a full description of the implementation code architecture for the overall program can be found in [3].

A. 1-D Implementation Example

Figure 2 shows a simple 1-D configuration of a set of cells, nodes, and material points as needed for MPM analysis. Using C++ (or other object-oriented language) each of these entities can be represented using classes. There are various possible ways to represent the relationships among the classes, but to accommodate the MPM algorithm as depicted in Figure 1 it is necessary to have available either explicitly or implicitly the following data structure groupings and capabilities:

- A list of all active cells to enable the outer iteration of steps (a) and (c).
- A list of particles and nodes belonging to each cell to perform the inner loop particle→node mapping/iteration of step (a) and its reverse for step (c).
- A list of all active nodes for the iteration of step (b).
- A list of all particles for the position update iteration of step (d) (this step can actually be subsumed into the iteration of step (c)).
- A mechanism for mapping particle locations to mesh cells.

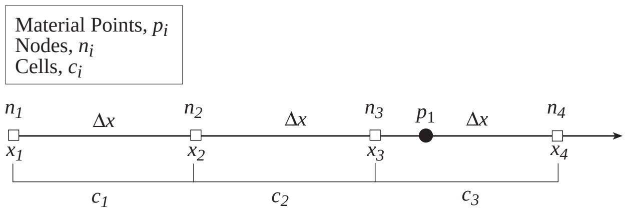

Fig. 2. 1-Dimensional MPM implementation example

In the case of static mesh allocation approaches and regular grids, the mesh data can be implemented directly as static arrays using straightforward looping, with a relatively simple grid-snap approach to map particle locations to cells, similar to the data ranging approach in [11]. For example, in reference to Figure 2, a 3-element array of cells could represent the mesh, iteration would consist of 3-step loops, and a particle's cell could be determined by its index into the cell array. Thus for the configuration shown, the cell index can be calculated from the particle position, x_p, and the uniform cell size, Δx, as

$$i = \text{floor}\left[\frac{(x_p - x_1)}{\Delta x}\right] \tag{6}$$

in which the floor operation truncates to return the integer part of its argument. Once the cell index is determined, the corresponding shape function arguments and nodal values can be determined directly.

In the case of dynamic meshing for this same configuration, the cell data structure would consist of only a single element

(c_3), because that is the only cell containing particles at the time shown. It can be seen that in principle 1/3 the memory would be required to represent the cells (and 1/2 the memory for the nodes), but now the direct array-index lookup approach based on Equation 6 to map particles to cells breaks down, so some alternative must be identified. One approach would be to use a search algorithm for each particle, essentially checking each cell to see if each particle belongs to it using a geometric inclusion check. It is not difficult to see that this kind of nested looping does not scale well since it needs to be carried out at each time step, and can thus be quite expensive as the number of particles and cells becomes large. A related alternative would be to use an initial exhaustive search combined with ongoing checks for particles leaving cells (and entering neighboring cells), but this requires extensive representation of neighboring cells (which may or may not exist prior to the time step in question), and relatively intricate consideration of all possible excursion cases.

What is needed is an approach that preserves the efficiency and simplicity of the direct spatially-based index lookup, but allows for the dynamic flexibility of only including active cells and nodes in the computation. A common solution to this kind of problem is to use an associative map, and that is the approach taken here. In particular, location-based key-value association via the built-in C++ *std::map* data structure [15] is used to achieve the desired capabilities. The *std::map* class supports rapid addition and deletion of elements, random access lookup, and iteration. Because these tasks tend to run at cross purposes (e.g., optimal iteration speed relies on data ordering and locality, while rapid insertion/deletion generally implies distributed data with implicit rather than explicit ordering), this does introduce some performance compromises, but as will be seen, the gains can greatly outweigh the costs.

Referring again to Figure 2, the *std::map* representation for this cell mesh would be set up by using a cell-based index as the lookup key via something like the pseudocode in Figure 3. Line 1 defines the *MeshCells* data structure as a map between an integer key and pointer to a *Cell* instance. Line 2 corresponds to the creation of a new *Cell* instance for cell index i computed according Equation 6, which is then inserted into the map in line 3. To find the cell associated with a given particle then boils down to using Equation 6 to map the particle location to an integer index, followed by a lookup in the *MeshCells* map using the relevant index.

```
std::map<int, Cell*> MeshCells ;
Cell* newCell = new Cell(...);
MeshCells[key] = newCell ;
```

Fig. 3. Simple cell creation and storage using the C++ std::map.

In practice, the construction of the *MeshCells* data structure is driven from the particle side as outlined in the code in Figure 4. Here the coordinate-based index is computed for each particle, and then the index is looked up in the *MeshCells* data structure (line 3). If the cell exists, it is hooked up to the particle in question (line 4). If the particle does not exist, it is created, added to the cell database, and linked to the particle in question (lines 6-8). The details of linking particles and cells is not shown, but the basic idea is to associate particles with cells for the algorithmic purposes identified earlier, and so the *AddParticle* function involves both the addition to the given cell's list of particles and the removal from the previous cell's particle list, as needed. Tagging particles with their current cell accelerates the lookup process, and it is only in the event of cell crossings that anything more involved than a quick pointer check is required to determine if a particle's cell has changed. Because the number of particles in a cell remains relatively small, the amount of list processing work needed to transfer a particle among cells is relatively small, as well.

```
for eachParticle in Particles do
    int key = std::floor( (eachParticle→ GetLocation()
    - Origin)/CellSize );
    if MeshCells.find(key) != MeshCells.end() then
        // ---found cell, link it to particle
        MeshCells[key]→ AddParticle(eachParticle);
    else
        // ---cell does not exist, so create
           it
        Cell* newCell = new Cell(...);
        MeshCells[key] = newCell;
        newCell→AddParticle(eachParticle);
    end
end
```

Fig. 4. Material point-driven cell creation

Once the particle/cell linkages have been updated via the process in Figure 4, empty cells can be identified and removed. This ensures that only cells that contain particles participate in the remainder of the processing for the time step in question.

Although not included in the above discussion, the management of nodes is an important part of the process, as well. The basic processes and techniques involved are essentially the same as those used for cells, and so the details have been left out to avoid redundancy. Also not included in the above pseudocode is the use of redundant lists (based on *std::vector*s) of cell and node pointers that simplify and streamline iteration, again with some additional overhead cost. The full code used to obtain the results presented later uses such additional iteration-friendly data structures, and the related costs will be shown to be minor.

B. 3D Implementation Overview

The previous subsection has outlined the basic ideas underlying the coordinate-key, map-based approach to dynamic mesh management in the case of 1-dimensional problems. To extend these ideas to 3-D it is simply necessary to accommodate 3-D position data in the mapping process (i.e., to map 3-D particle locations to associated cells). The overall MPM framework and procedures from the 1-D case remain essentially the same.

There are various ways to use direct spatial hashing for this kind of problem, but the approach taken in this work to accommodate higher dimensional location data has been to use nested C++ *std::map* structures. This nesting introduces some reduced baseline efficiency compared to direct coordinate hashing, but in the context of the overall MPM implementation, testing indicated that the performance difference was negligible. The approach has the advantage that it uses standard C++ data structures and is not sensitive to custom hash algorithm details.

Using the nested map approach the analog to the *MeshCells* data structure declaration for the 1-D case is defined as shown in Figure 5. Here the *MeshCells3D* data structure uses a nested lookup to associate three integer keys with a *Cell* pointer. In analogy with the 1-D case, the creation and storage of an individual cell with array indices

$$ix = \text{floor}\left[\frac{(x_p - x_1)}{\Delta x}\right] \quad (7)$$

$$iy = \text{floor}\left[\frac{(y_p - y_1)}{\Delta y}\right] \quad (8)$$

$$iz = \text{floor}\left[\frac{(z_p - z_1)}{\Delta z}\right] \quad (9)$$

can be accomplished as shown in Figure 6. Note that these indices should be interpreted in analogy with the 1-D case, corresponding to the array indices that would identify the cell in a statically-allocated, 3-D array.

```
std::map<int, std::map<int, std::map<int, Cell*> > >
MeshCells3D;
Cell* newCell = new Cell(...);
// ---ix, iy, iz specified
MeshCells3D[ix][iy][iz] = newCell ;
```

Fig. 5. Simple 3D cell creation and storage using nested map structures.

As in the 1-D case, the cell (and node) management is driven by the particles. Combining nested map with a straightforward expansion of the grid snapping expression from Equation 6 to account for three coordinate directions, the lookup code corresponding to that of Figure 3 can be written as shown in Figure 6. The similarity between the 1-D and 3-D cases is apparent.

There is one complicating factor in managing the 3-D case, and that involves the removal of cells that contain no particles. To avoid the creation of memory leaks it is necessary to use explicit nested deletion of the map entries. This is illustrated in Figure 7, in which the cell with indices ix, iy, iz is deleted (assuming it currently exists and has been found to contain no particles). The initial call to *delete* frees the memory associated with the cell object itself, and then the subsequent calls to the built in *std::map::erase* free up the memory associated with the pointer map storage slots themselves. This incremental freeing up of memory from vacated cells is insignificant for small problems but can be critical for problems experiencing large displacements and an associated use of a large number of different cells during the lifetime of an analysis.

As described earlier, C++ maps (or similar data structures) perform reasonably well for insertion, deletion, lookup, and iteration, but with some compromises relative to data structures optimized for only some of these tasks. It is therefore important to investigate the performance of the approach in actual

```
for eachParticle in Particles do
    int ix = std::floor( (eachParticle→ GetXLocation()
    - Origin.GetXLocation())/CellSizeX );
    int iy = std::floor( (eachParticle→ GetYLocation()
    - Origin.GetYLocation())/CellSizeY );
    int iz = std::floor( (eachParticle→ GetZLocation()
    - Origin.GetZLocation())/CellSizeZ );
    if MeshCells3D.find(ix) != MeshCells3D.end()
    and MeshCells3D[ix].find(iy) != MeshCells3D[ix].end()
    and MeshCells3D[ix][iy].find(iz) !=
    MeshCells3D[ix][iy].end()
    then
        // ---found cell, link it to particle
        MeshCells3D[ix][iy][iz]→ AddParticle(eachParticle);
    else
        // ---cell does not exist, so create
           it
        Cell* newCell = new Cell(...);
        MeshCells[ix][iy][iz] = newCell;
        newCell→AddParticle(eachParticle);
    end
end
```

Fig. 6. Material point-driven cell creation for the 3-D case

```
// ---Delete the unneeded cell
delete MeshCells3D[ix][iy][iz];
// ---Perform nested map cleanup
MeshCells3D[ix][iy].erase(iz);
if MeshCells3D[ix][iy].size() == 0 then
    MeshCells3D[ix].erase(iy);
end
if MeshCells3D[ix].size() == 0 then
    MeshCells3D.erase(ix);
end
```

Fig. 7. Cell removal for the 3-D case showing nested deletion to avoid memory leaks

problem contexts to evaluate the usefulness of the approach in enhancing overall computation time.

V. Example Problems and Performance Results

This section presents a set of example problems to illustrate the use of the dynamic MPM meshing approach described in the previous sections. Consideration is given to both memory demands and computational speed, although these two are not entirely independent, especially when memory demands exceed available physical RAM. Three cases are considered for each problem: (i) simple static allocation for the entire domain; (ii) dynamic cell creation without cell deletion; and (iii) fully dynamic cell creation and deletion. It should be noted that the simple static allocation case uses the same overall framework as the dynamically meshed cases, and so there is some additional iteration overhead associated with *std::vector* iteration versus raw array iteration.

The problems chosen represent cases involving large displacements and unpredictable domains of travel, since these are the classes of problems for which this approach has been developed. For problems involving few particle cell crossings, simpler reduced static allocation methods would be adequate, although the approach presented here can handle these cases, as well.

As a basic benchmark, it is not difficult to show that optimal basic memory demand and iteration size reductions should scale like the ratio of the body volume(s) to the volume of the containing space. For unbounded domains this clearly is not a meaningful metric, but using an effective bounding box on the motion for the containing space can provide a reasonable baseline alternative. Although in general this bounding box would not be known a priori, within the current benchmarking context it is easy to determine it in an *a posteriori* fashion once the analysis is complete.

Because of the variety of implementations, compilers, and hardware configurations and combinations that are possible, it is not feasible to investigate performance exhaustively, and so the results presented here should be considered indicative rather than definitive. These results were obtained using C++ with the gcc 4.0 compiler running on AMD Opteron 64 bit processors and 4 GB of RAM.

A. Tumbling Rod

The first example involves the analysis of a prismatic elastic bar bouncing off a flat horizontal surface, as shown in Figure 8. The bar behaves in a relatively rigid fashion, but it is not simple to predict where the bar will travel.

Figure 9 shows a set of images taken at a fixed point in time during the bar's motion, with each sub-image showing one of three mesh allocation strategies for the same instant in time. Figure 9(a) shows the case of a statically-allocated, unchanging background mesh, which would be the typical default approach used for a problem of this kind. Figure 9(b) shows the case in which mesh cells are allocated dynamically as needed, but which are not removed after all material has left. This approach trades off the computational cost of deallocation with the cost associated with carrying unneeded cells through the analysis. Finally, 9(c) shows the fully dynamic allocation approach in which only needed cells are used throughout the analysis. Although difficult to see in a figure of this size, the bar itself is represented by a set of material points arranged to match the bar's (original) geometry, and the material points carry forward the analysis.

Testing the performance of these three scenarios makes it possible to examine the advantages and disadvantages of each strategy, and to assess the overhead required to manage the background mesh dynamically. Figure 10 shows the absolute performance behavior for the three bouncing bar analyses. In particular, accumulated CPU time is shown as a function the simulation time used in the analysis. For the static and fully dynamic cases the observed behavior is essentially linear as expected, consistent with the fact the each time step takes a relatively constant amount of time to execute. The dynamic case without cell removal shows nonlinear behavior as the slope increases with time, an effect caused by the accumulating problem size associated with the wake of undeleted cells. Initially, though, the no-deletion approach outperforms the fully dynamic case, because of the saving of the overhead associated with identifying and deleting unused cells.

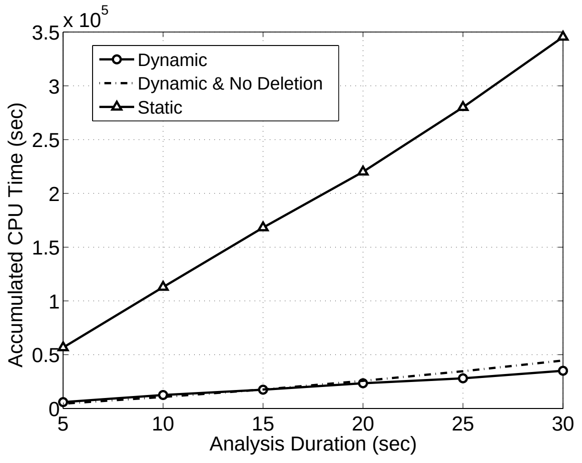

Fig. 10. Comparison of performance for the tumbling bar problem for three mesh management strategies.

Figure 11 shows the relative performance of the two dynamic schemes referenced to the static allocation case. The fully dynamic case remains steady around 10-11% of the static mesh time, while the dynamic, no-deletion case starts around 8% and then increases steadily as the analysis proceeds. The rod dimensions were 2 m$\times$8 m$\times$2 m, and the static mesh shown was 10 m$\times$32 m$\times$10 m, so the actual body/static background mesh volume ratio for this problem is 0.01 (i.e., 1%). Because a significant portion of the computations involves the particles, one would not expect a direct correlation between the cell ratio and the performance ratio, but this shows that the computational cost of including empty cells and unneeded nodes can be substantial. It should be noted that the problem size was chosen such that everything fit into physical memory—in the event that the extra storage associated with unneeded cells and nodes forces the problem to be solved using virtual memory, the static mesh approach could increase by orders of magnitude and become impractical. It is thus worth looking at the memory usage benefits of the dynamic approach, as well.

Memory usage comparison between the fully dynamic and static allocation schemes is shown in Figure 12. As can be seen in the figure, the memory use ratio remains relatively constant throughout the analysis as would be expected, with a value around 3.4%. Because total memory use is a composite of cell and node storage, particle storage, and general supporting data structure storage, this ratio would not be expected to match the volume ratio directly, since the volume ratio correlates to cell and node storage only. Nonetheless it can be seen to track this ratio more closely than in the case of the CPU-time metric presented above. Again, as mentioned above, these kinds of memory savings could have a significant impact on performance, and even feasibility for some classes of problems.

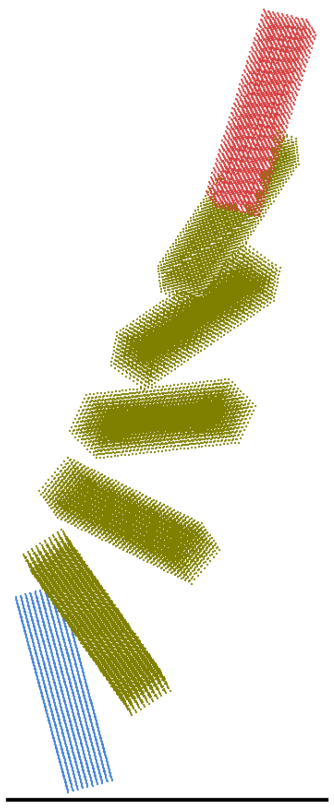

Fig. 8. Snapshots from a 3D analysis of a tumbling bar showing its initial configuration (blue) before contact with the horizontal surface, and a series of configurations in 5 second intervals for a duration of 30 seconds.

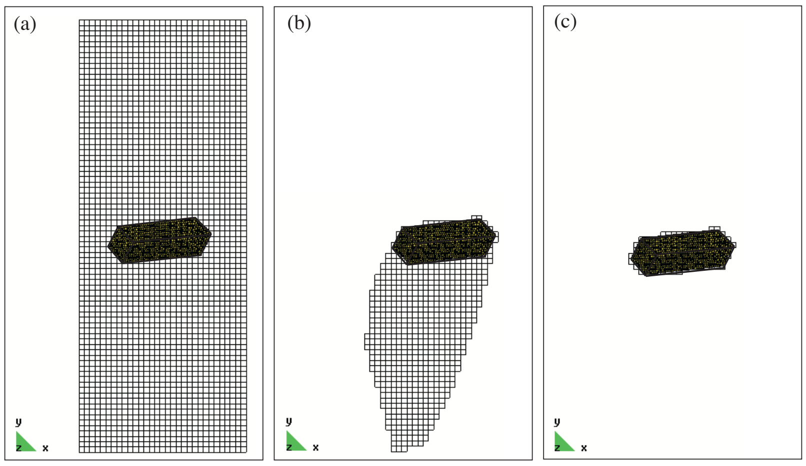

Fig. 9. Snapshots from a 3D analysis of a tumbling bar showing three different mesh management strategies: (a) static allocation; (b) dynamic allocation with no removal; and (c) fully dynamic allocation.

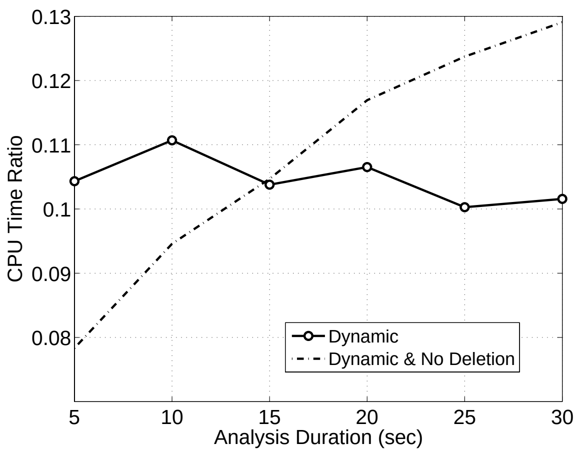

Fig. 11. Comparison of dynamic meshing with and without cell deletion: CPU time is shown relative to the static allocation case.

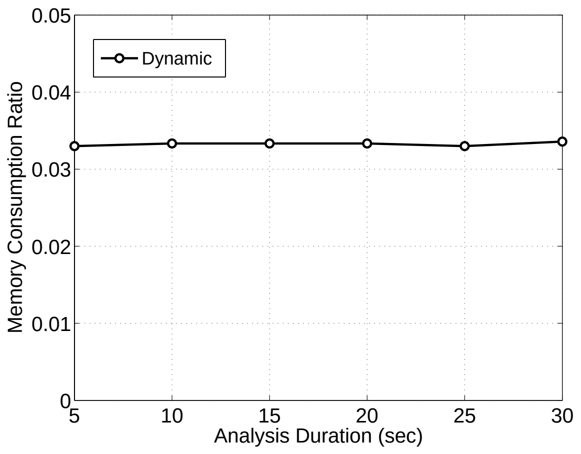

Fig. 12. Comparison of dynamic meshing with and without cell deletion: memory usage is shown relative to the static allocation case.

B. Flow Examples

The example in the previous section illustrated the ability of the dynamic meshing approach to introduce significant memory and performance improvements in efficient fashion. In this section some more challenging problems in regards to complex deformations and flow-like behavior are considered, the point being to show the relative robustness of the approach for general mesh tracking.

Figure 13 shows a sequence of images from a breaking dam water flow analysis. This particular analysis has been set up to constrain the motion to 2-D, although the underlying framework is 3-D. At $t = 0$ a rectangular container of water subjected to gravity is put into motion by instantaneously removing a constraining wall bounding the right side side of the fluid. The details of this analysis and correlation with experimental observations can be found in [3], but the key point for the present discussion is to observe that the mesh successfully tracks the flow as the material takes on complex geometries and even disaggregates as it splashes. Compared to a conventional meshing and array pointer, the memory saving and the reduction in particle/cell search effort are significant. Memory reduction for the dam-break problem is above 90 % at all times throughout the analysis.

Figure 14 shows an example of a channel debris flow analysis in which an erodable embankment is placed in the path of the sliding soil mass (again, the general aspects and details of this analysis can be found in [3]). The channel walls and the sliding surface are not shown, and this particular sequence of images depicts the material point particle representations rather the background mesh, but as in the previous example the dynamic meshing is capable of tracking the distinct body motions, deformations, and interactions while saving the overhead associated with unneeded cells throughout the analysis.

VI. Summary and Conclusions

This paper has presented a computationally efficient approach to background grid management for use with the MPM that ensures only grid cells that contain material points are instantiated at each time step. For problems with significant motion of material points in an unbounded domain, this removes the need to build an estimated bounding mesh a priori, and it results in both a reduction in the memory required to store the mesh and the iteration costs associated with computing on the mesh. The amount of reduction is related to the size of the material domain relative to the size of the background domain, which in the 3D case scales roughly like the ratio of the volume of the body to the rectilinear volume of the bounding space. For the test problems considered in this paper, these ratios ranged from 1% to 10%.

The specific nested-map, grid-snap-based mesh management method presented in this paper has been shown to be robust and efficient in terms of handling general problem types while introducing negligible additional overhead in regards to cell and node creation and deletion. Thus, complex material motions such as arise in debris flows or instances of tumbling, colliding bodies can be closely tracked with minimal grid cell counts, while preserving straightforward and efficient iteration.

The memory savings provided by the dynamic meshing algorithm can be in the order of 90–99 %, depending on the type and spatial dimension of the problem. This enables computation of larger problems at higher resolution on existing hardware.

The method in its current form is only directly applicable to uniform, rectilinear background meshes, but there are various ways to extend the method so that this could be achieved. However, the use of specialized meshes is not common for the classes of problems considered here involving motion/flow in unbounded domains. The method also has not yet been implemented to run in parallel environments, but this development work is currently underway and will be reported at a later date.

Acknowledgment

This research was supported through the UW Royalty Research Fund and NSF grant CMMI–0900318, which is gratefully acknowledged.

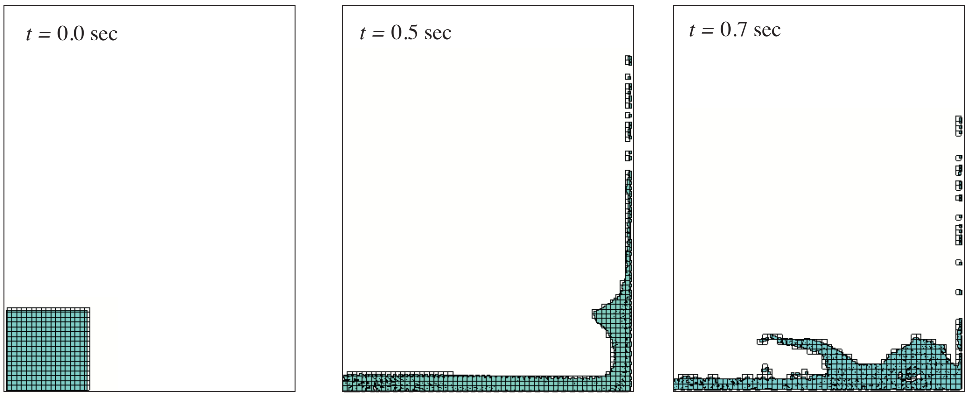

Fig. 13. Image sequence from an analysis of a fluid release and splashing in a container illustrating mesh tracking with disaggregating material.

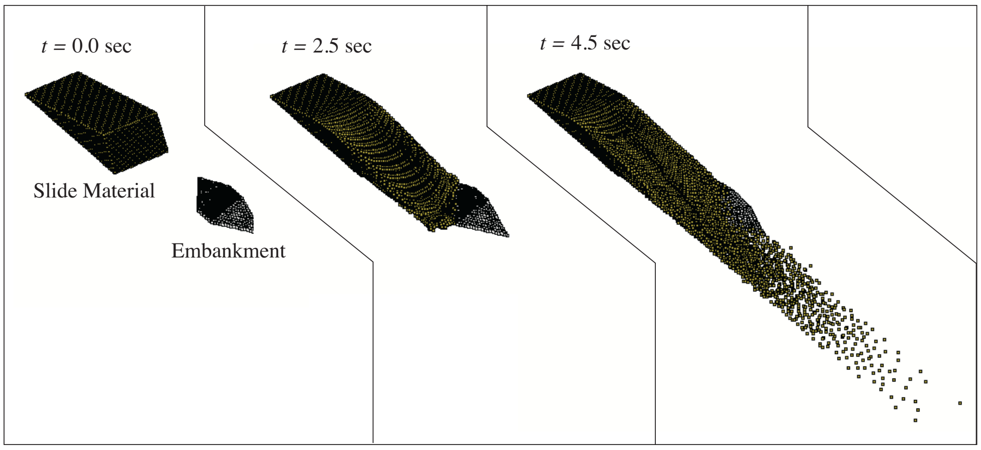

Fig. 14. Sequence of images from chute debris flow analysis with eroding embankment obstacle. Chute walls and background mesh not shown.

References

[1] D. Sulsky, Z. Chen, and H. L. Schreyer, "A particle method for history-dependent materials," *Computer Methods in Applied Mechanics and Engineering*, vol. 118, no. 1–2, pp. 179–196, 1994.

[2] D. Sulsky, S. Zhou, and H. L. Schreyer, "Application of a particle-in-cell method to solid mechanics," *Computer Physics Communications*, vol. 87, no. 1–2, pp. 236–252, 1995.

[3] W.-K. Shin, "Numerical simulation of landslides and debris flows using an enhanced material point method," Ph.D. dissertation, University of Washington, 2009.

[4] P. Mackenzie-Helnwein, P. Arduino, W. Shin, J. A. Moore, and G. R. Miller, "Modeling strategies for multiphase drag interactions using the material point method," *International Journal for Numerical Methods in Engineering*, 2010, in print.

[5] S. Bardenhagen, J. U. Brackbill, and D. Sulsky, "The material point method for granular materials," *Computer Methods in Applied Mechanics and Engineering*, vol. 187, no. 3–4, pp. 529–541, 2000.

[6] S. Bardenhagen, J. Guilkey, K. Roessig, J. Brackbill, W. Witzel, and J. Foster, "An improved contact algorithm for the material point method and application to stress propagation in granular material," *Computer Modeling in Engineering & Sciences*, vol. 2, pp. 509–522, 2001.

[7] W. Hu and Z. Chen, "A multi-mesh MPM for simulating the meshing process of spur gears," *Computers & Structures*, vol. 81, no. 20, pp. 1991–2002, 2003.

[8] J. A. Nairn, "Material point method calculations with explicit cracks," *Computer Modeling in Engineering and Sciences*, vol. 4, pp. 649–664, 2003.

[9] M. Steffen, R. M. Kirby, and M. Berzins, "Analysis and reduction of quadrature error in the material point method (MPM)," *Int. J. Numer. Meth. Engng.*, vol. 76, no. 6, pp. 922–948, 2008.

[10] Y. Zhang, J. Guilkey, J. Hoying, and W. J.A., "Mechanical simulation of multicellular structures with the material point method," in *c-CMBBE2004*, March 2004, p. (6 pages).

[11] J. Bentley and J. H. Friedman, "Data structures for range searching," *Computing Surveys*, vol. 11, no. 4, pp. 397–409, 1979.

[12] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM SIGGRAPH*, pp. 579–588, 2006.

[13] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Sliced data structure for particle-based simulations on gpus," in *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. New York, NY, USA: ACM, 2007, pp. 55–62.

[14] D. A. Watt, *Programming Language Concepts and Paradigms*. Prentice-Hall, 1990.

[15] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley Professional, 1997.