 CMES, vol.38, no.2, pp.119-147, 2008

Shared Memory OpenMP Parallelization of Explicit MPM and Its Application to Hypervelocity Impact

P. Huang[1,2], X. Zhang[1,3], S. Ma[1] and H.K. Wang[1]

Abstract: The material point method (MPM) is an extension of particle-in-cell method to solid mechanics. A parallel MPM code is developed using FORTRAN 95 and OpenMP in this study, which is designed primarily for solving impact dynamic problems. Two parallel methods, the array expansion method and the domain decomposition method, are presented to avoid data races in the nodal update stage. In the array expansion method, two-dimensional auxiliary arrays are created for nodal variables. After updating grid nodes in all threads, the auxiliary arrays are assembled to establish the global nodal array. In the domain decomposition method, the background grid is decomposed into some uniform patches, and each thread deals with a patch. The information of neighbor patches is exchanged through shared variables. After updating nodes in all patches, their nodal variables are assembled to establish the global nodal variables. The numerical tests show that the domain decomposition method has much better parallel scalability and higher parallel efficiency than the array expansion method. Therefore, a parallel computer code, MPM3DMP, is developed based on the domain decomposition method. Finally, MPM3DMP is applied to a large-scale simulation with 13,542,030 particles for obtaining the high-resolution results of debris cloud in hypervelocity impact.

Keywords: Material point method, PIC, parallel methods, OpenMP, hypervelocity impact.

1 Introduction

The material point method (MPM) [Sulsky, Chen, and Schreyer (1994); Sulsky, Zhou and Schreyer (1995)] is an extension of the particle-in-cell (PIC) method [Harlow (1963); Brackbill, Kothe, and Ruppel (1988)]. MPM is applied to the

[1] School of Aerospace, Tsinghua University, Beijing 100084,China
[2] Institute of System Engineering, China Academy of Engineering Physics, Mianyang 621900, China
[3] Correspondence author. E-mail: xzhang@tsinghua.edu.cn

solid dynamic problems by updating the stress in material points rather than in grid, so that the history dependent material can be modeled conveniently. Being a fully Lagrangian particle method, MPM discretizes a material domain using a set of material points, which are also called particles. The particles carry all state variables such as displacement, stress, strain and temperature. The momentum equations are solved on a predefined regular background grid, so that the grid distortion and entanglement are completely avoided. MPM possesses the advantages of both Lagrangian method and Eulerian method. Recently, some researchers developed the generalized interpolation material point (GIMP) and the other variants for improving the accuracy of MPM [Bardenhagen and Kober (2004); Steffen, Wallstedt, Guilkey, Kirby and Berzins (2008)].

MPM and its variants have been successfully applied to solve many complicated engineering problems, such as Taylor bar impact [Sulsky and Schreyer (1996)], upsetting problems [Sulsky and Kaul (2004)], hypervelocity impact [Zhang, Sze, and Ma (2006)], explosive processes [Hu and Chen (2006); Guilkey, Harman and Banerjee(2007)] and multiphase flows [Zhang, Zou, VanderHeyden and Ma (2008)]. Some problems, which involve material failure [Schreyer, Sulsky, and Zhou (2002); Sulsky and Schreyer (2004); Chen, Gan and Chen (2008)], dynamic fracture [Guo and Nairn (2004); Guo and Nairn (2006), Ma, Lu, and Komanduri (2006)] and film delamination [Shen and Chen (2005)], have also been studied using MPM and its variants.

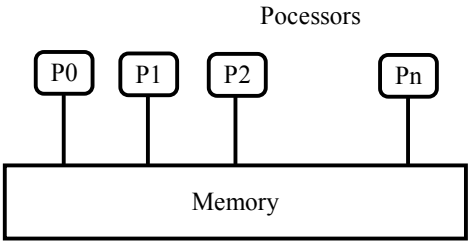


Figure 1: A shared-memory architecture.

Parallelization of MPM is desirable to obtain high-resolution results for large-scale engineering problems. Parker (2006) developed a parallel MPM code using Message Passing Interface (MPI). A parallel MPM computation with 16 million particles was executed by Parker and co-workers in Massively Parallel Processing (MPP) machines [Parker, Guilkey and Harman (2006)]. MPI is an application program interface (API) to design a parallel code, which supports distributed-memory machines, such as cluster and MPP machines. In the parallel MPM code

developed by Parker (2006), the computational domain was decomposed to many patches. The information of neighbor patches was exchanged explicitly by using MPI_ Send and MPI_ Receive operation. Based on the Structured Adaptive Mesh Refinement Application Infrastructure (SAMRAI), Ma and co-workers presented parallel GIMP simulations for the multi-scale problems [Ma, Lu, Wang, Roy, Hornung, Wissink and Komanduri (2005)]. SAMRAI is designed to provide a framework for multi-processor distributed memory computations, which is developed using C++ and MPI. In addition, the Parallel Object Oriented Methods and Applications (POOMA) framework is also developed by C++ and MPI, which is specially designed to provide a flexible environment for data parallel programming of Particle-in-cell (PIC) type codes [Reynders, Hinker, Cummings, Atlas, Banerjee, Humphrey, Karmesin, Keahey, Srikant, and Tholburn (1996)]. A parallel high-performance 3D Maxwell PIC code, called Capone, has been implemented using POOMA II framework on Linux [Candel, Dehler and Troyer (2006)].

In shared memory machines, parallelization for MPM could be achieved using OpenMP. OpenMP has the significant advantage of allowing programs to be incrementally parallelized [Quinn (2004)]. Annoying message passing between different sub domains is avoided in OpenMP. It is easier to parallelize a serial code using OpenMP than MPI. OpenMP is designed to provide a standard interface for FORTRAN and C/C++ programs. OpenMP is suitable for shared-memory machines, such as multi-core computers and Symmetric Multi-Processor machines (SMP). Fig. 1 shows the architecture of this class of computers. In fact, this class of machines implicitly supports the message exchange between different processors through shared memory. Although shared-memory machines can never compete with MPP machines for large-scale simulations, the high performance computational environment can be setup by this class of machines easily and cheaply.

OpenMP has been successfully applied to parallel computations of molecular dynamics [Couturier and Chipot (2000), Goedecker (2002)]. The speedup of a parallel molecular dynamics code was reported between 6.0 and 7.0 under eight processors [Couturier and Chipot (2000)]. Using OpenMP and FORTRAN 77, a parallel smoothed particle hydrodynamics (SPH) code named HYDRA_OMP was developed by Thacker and Couchman (2006). Moreover, OpenMP was applied to parallel computations for Ab initio quantum chemistry [Sosa, Scalmani, Gomperts and Frisch (2000)] and fluid dynamics [Martin, Papada and Doallo (2004); Ayguadea, Gonzaleza, Martorella and Jostb (2006)]. In this study, two parallel methods are presented for parallelization of MPM based on OpenMP, which are the array expansion method and the domain decomposition method. A parallel computer code is developed for solving the impact problems.

This paper is organized as follows. In section 2, the algorithm of explicit MPM is

reviewed briefly. In section 3, the parallel modes of OpenMP are described briefly. In section 4, MPM is parallelized by two parallel methods based on OpenMP. A parallel MPM code called MPM3DMP is developed. In section 5, Taylor bar impact is simulated by using the MPM models with different sizes for comparing the parallel efficiency. Then, MPM3DMP is further applied to hypervelocity impact for obtaining the high-resolution results of debris cloud. Lastly, section 6 is a summary of our discussions and our conclusions.

2 Review of the MPM algorithm

In MPM, each body is discretized by a set of particles which carry all state variables (see Fig.2). In each time step, the MPM computation consists of two parts: updating variables for grid nodes and updating variables for particles. Because the particles are rigidly attached to a regular background grid, they move with the grid during each time step. The variables can be mapped between particles and grid nodes using the standard shape functions of the finite element method (FEM).

MPM was compared with other numerical methods such as FEM and SPH for investigating its accuracy and performance [Guilkey and Weiss (2003); Ma, Zhang and Qiu (2009)]. The more accuracy simulations of MPM could be performed by changing the shape functions [Steffen, Wallstedt, Guilkey, Kirby and Berzins (2008)], using different computational schemes [Bardenhagen (2002)] and adopting an adaptive algorithm [Tan and Nairn (2002)]. In addition, some researchers presented contact MPM algorithms for solving contact and friction between of bodies. [York, Sulsky and Schreyer (1999); Bardenhagen, Brackbill and Sulsky (2000); Hu and Chen (2003); Pan, Xu, Zhang, Zhu, Ma and Zhang (2008)].

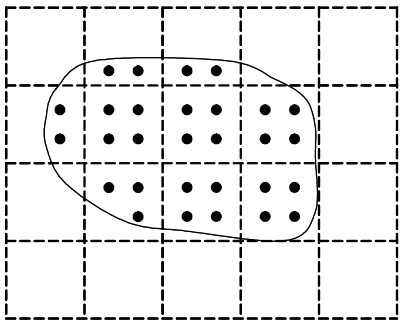


Figure 2: Material discretization in MPM. Solid line denotes the boundary of material domain. Solid dot denotes material point and dash lines denote background grid.

The standard MPM algorithm is described briefly as follows. In the following equations, the subscript i denotes the value of grid node i, the subscript p denotes the value of particle p, and the superscripts k and k+1 denote the value at time step k and k+1, respectively. $S_{ip} = N_i(\mathbf{X}_p)$ is the shape function of node i evaluated at particle p, and $\mathbf{G}_{ip} = \nabla N_i(\mathbf{X}_p)$ is the gradient of the shape function of node i evaluated at particle p. MPM computation in each time step can be completed over four steps [Sulsky, Chen, and Schreyer (1994); Zhang, Sze and Ma (2006)].

(1). Map the particle variables to grid nodes to establish their momentum equations.

The mass of grid node i can be obtained by mapping the mass of those particles located in the cells connected to grid node i, namely

$$m_i^k = \sum_p m_p S_{ip}^k \tag{1}$$

The momentum of grid node i can be obtained in the same way as

$$\mathbf{p}_i^k = \sum_p m_p \mathbf{v}_p^k S_{ip}^k \tag{2}$$

The force of grid node i can be obtained as

$$\mathbf{f}_i^k = \left(\mathbf{f}_i^{int}\right)^k + \left(\mathbf{f}_i^{\text{ext}}\right)^k \tag{3}$$

where $\left(\mathbf{f}_i^{int}\right)^k$ is the internal force of grid node i and given by

$$\left(\mathbf{f}_i^{int}\right)^k = -\sum_p \sigma_p^k \cdot \mathbf{G}_{ip}^k \frac{m_p}{\rho_p^k} \tag{4}$$

$(\mathbf{f}_i^{\text{ext}})^k$ is the external force of grid node i and given by

$$\left(\mathbf{f}_i^{\text{ext}}\right)^k = \sum_p m_p S_{ip} \mathbf{b}_p + \int_{\Gamma_t} S_{ip} \mathbf{t} d\Gamma \tag{5}$$

where $\mathbf{b}$ is the specific body force, and $\mathbf{t}$ is the prescribed traction on boundary Γ_t.

(2). Update the momentum of grid node i using the explicit time integration,

$$\mathbf{p}_i^{k+1} = \mathbf{p}_i^k + \mathbf{f}_i^k \Delta t \tag{6}$$

Both $\mathbf{p}_i^{k+1}$ and $\mathbf{f}_i^k$ are set to zero on the fixed boundary.

(3). Map the nodal results back to particle p to update its position $\mathbf{x}_p^{k+1}$ and velocity $\mathbf{v}_p^{k+1}$,

$$\mathbf{x}_p^{k+1} = \mathbf{x}_p^k + \bar{\mathbf{v}}_p^{k+1}\Delta t \tag{7}$$

$$\mathbf{v}_p^{k+1} = \mathbf{v}_p^k + \mathbf{a}_p^k \Delta t \tag{8}$$

where

$$\bar{\mathbf{v}}_p^{k+1} = \sum_{i=1}^{8} \frac{\mathbf{p}_i^{k+1}}{m_i^k} S_{ip}^k \tag{9}$$

$$\mathbf{a}_p^k = \sum_{i=1}^{8} \frac{\mathbf{f}_i^k}{m_i^k} S_{ip}^k \tag{10}$$

(4). Update the particle stress using a constitutive model.

To obtain more accurate nodal velocities, the updated particle velocity is mapped back to grid nodes in the Modified Update Stress Last (MUSL) scheme [Chen and Brannon (2002); Nairn (2003)], namely,

$$\mathbf{v}_i^{k+1} = \frac{1}{m_i^k} \sum_p m_p \mathbf{v}_p^{k+1} S_{ip}^k \tag{11}$$

The incremental strain and vorticity of particle p can be evaluated by

$$\Delta \boldsymbol{\varepsilon}_p^k = \frac{\Delta t}{2} \sum_{i=1}^{8} \left[\mathbf{v}_i^{k+1} \left(\mathbf{G}_{ip}^k \right)^T + \mathbf{G}_{ip}^k \left(\mathbf{v}_i^{k+1} \right)^T \right] \tag{12}$$

$$\Delta \boldsymbol{\omega}_p^k = \frac{\Delta t}{2} \sum_{i=1}^{8} \left[\mathbf{v}_i^{k+1} \left(\mathbf{G}_{ip}^k \right)^T - \mathbf{G}_{ip}^k \left(\mathbf{v}_i^{k+1} \right)^T \right] \tag{13}$$

The density and Cauchy stress of particle p are updated by

$$\rho_p^{k+1} = \rho_p^k \Big/ \left(1 + tr \left(\Delta \boldsymbol{\varepsilon}_p^k \right) \right) \tag{14}$$

$$\boldsymbol{\sigma}_p^{k+1} = \boldsymbol{\sigma}_p^k + \Delta \mathbf{r}_p^k + \Delta \boldsymbol{\sigma}_p^k \tag{15}$$

where $\Delta \mathbf{r}_p^k = \Delta \boldsymbol{\omega}_p^k \cdot \boldsymbol{\sigma}_p^k - \boldsymbol{\sigma}_p^k \cdot \Delta \boldsymbol{\omega}_p^k$.The incremental stress $\Delta \boldsymbol{\sigma}_p^k$ is updated by a constitutive model. For hypervelocity impact problems, the particle pressure is updated by an equation of state (EOS), such as Mie-Grüneisen EOS.

Based on the aforementioned scheme, a serial computer code called MPM3D is developed using FORTRAN 95 and object-oriented programming techniques. The

code has a highly modularized and object-oriented property because of using modules and derived data types. The flowchart of MPM3D code is given in Fig. 3. It can be seen that the MPM computation can be divided into two parts: updating variables for grid nodes (Eqs.(1) $\sim$ (6)) and updating variables for particles (Eqs.(7) $\sim$ (15)). When updating the nodal variables, the state variables of particles keep unchanged. When updating the state variables of particles, the nodal variables keep unchanged. The variables of grid nodes and particles are updated through looping over particles [Chen and Brannon (2002)].

3 Brief review of OpenMP

The standard mode of parallelization in OpenMP is the fork/join mode [Quinn (2004)], which is illustrated in Fig.4. The master thread executes the serial part of a code. At the beginning of a parallel code, the master thread forks some slave threads. The master thread and slave threads work concurrently across the parallel section. All threads return the single master thread at the end of a parallel code. OpenMP supports the incremental parallelization and shared memory. When multiple threads write the same shared variables at the same time, an unpredictable result will be obtained due to data races. Avoiding data races is a serious issue in writing a correct parallel code using OpenMP.

OpenMP have two common styles of programming [Chandra, Dagum, Kohr, Maydan, McDonald and Menon (2001)]: loop-level parallelization and code-block parallelization. In the loop splitting method, all loop iterations are equally decomposed and distributed across all the threads. For executing a loop in a parallel manner, the directive '!$omp parallel do' is placed before the do loop code.

With the code-block parallelization, portions of code will be replicated, and the same computational process is executed in every thread. For executing the code-block parallelization, the directive '!$omp parallel' is placed at the beginning of the parallel area, and the directive '!$omp end parallel' is placed at the end of the parallel area. Using the code-block parallelization, an entire loop domain is divided into different loop sub domains according to different threads.

Regardless of which parallel mode is employed, the data dependence must be removed in different threads. Different synchronization mechanisms can be used in OpenMP to avoid data races, such as the critical section directive and reduction clause. To avoid writing a shared variable by multiple threads at the same time, the critical section is created in the parallel area. In the critical section, only one thread at a time can update the value of a shared variable. However, if a critical region is very large, the program performance might be poor [Pantalé (2005); Chapman, Jost and VanderPas (2007)]. OpenMP provides the reduction clause for perform-

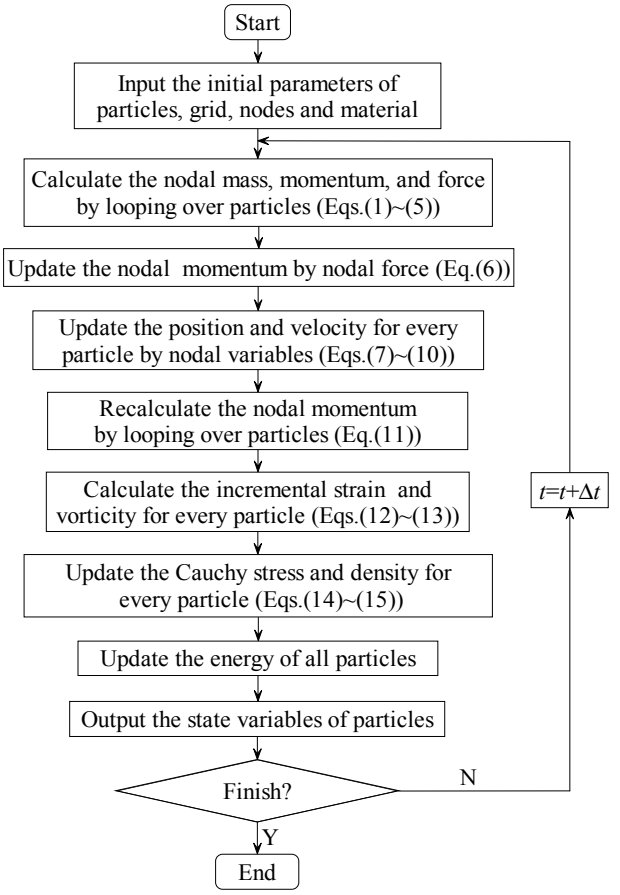


Figure 3: Flowchart of MPM3D code.

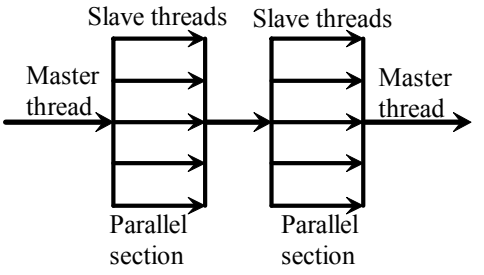


Figure 4: A fork/join parallel mode.

ing recurrence calculations and avoiding data races. The code adopting reduction clause has better parallel performance than the code using the critical section directive. However, only scalar variables are allowed to use a reduction clause. The reduction of an array can be achieved using the array expansion method. In this method, a two-dimensional auxiliary array *A_list*(:, *nthreads*) is created for a one-dimensional array *A*(:) in a parallel area. The parameter *nthread* is the total number of threads used in a parallel code. Thread *j* only operates the j^{th} column of the array *A_list*, so that data races are avoided. At the end of the parallel area, the array *A*(:) can be assembled by

$$\mathrm{A}(:) = \sum_{j=1}^{nthreads} \mathrm{A\_list}(:, j) \tag{16}$$

4 The OpenMP parallelization of MPM

In each time step, the MPM computation is completed over two stages. The first stage updates variables for grid nodes through looping over particles (Eqs.(1) $\sim$ (6)). However, the loops over particles are dependent of each other because different particle loops may operate the same grid node at the same time. Therefore, special attention must be paid to data races when parallelizing this stage using OpenMP.

The second stage updates variables for particles through looping over particles (Eqs.(7) $\sim$ (15)). Data races don't occur in this stage because each particle loop only operates the particles belonging to this loop. Consequently, the OpenMP parallelization of this stage can be achieved straightforwardly by using the loop splitting method.

4.1 Parallelization of nodal variable updates

Let *Mg*(*N*), *Pxg*(*N*) and *Fxg*(*N*) denote the nodal mass, momentum and force arrays, respectively, where *N* is the total number of grid nodes. These arrays are calculated by looping over particles. As above mentioned, the loops over particles are dependent of each other in this stage. Therefore, the array expansion method and the domain decomposition method are used to avoid data races, respectively.

4.1.1 The array expansion method

In the array expansion method, the one-dimensional array of a nodal variable is expanded to a two-dimensional array. For example, two-dimensional arrays *Mg_list*(*N*, *nthreads*), *Pxg_list*(*N*, *nthreads*) and *Fxg_list*(*N*, *nthreads*) are created as the auxiliary arrays for the nodal mass *Mg*(*N*), momentum *Pxg*(*N*) and force *Fxg*(*N*), re-

spectively. Thread i only operates the i^{th} column of these arrays, so that the data races are avoided. The update stage of nodal variables can be parallelized as

```
!$omp parallel do &
!$omp private(state variables of particles) &
!$omp default(shared)
do p =1, number_ particles
thread = omp_get_thread_num()+1
...
do n = 1,8
node = Influence_Node(n)
update Mg_list(node,thread)
update Pxg_list(node,thread)
update Fxg_list(node,thread)
end do
end do
```

where the function *Influence_Node*(n) returns the index of n^{th} node in the cell connected to particle p. Because the loop is over particles, the state variables of particles need to be declared as private variables for each thread. After updating nodal variables in all threads, the auxiliary array *Mg_list* is assembled into the nodal mass array *Mg* according to Eq.(16). Moreover, the auxiliary arrays *Pxg_list* and *Fxg_list* are assembled into arrays *Pxg* and *Fxg*, respectively. By using the array expansion method, the critical region is eliminated in the parallel code, so that the parallel efficiency could be enhanced. However, the memory requirement will increase significantly with increasing threads.

The time integration of the nodal momentum is given by Eq.(6). There are no data races in this loop, so that the time integration can be parallelized easily using the loop splitting method as

```
!$omp parallel do
do n = 1, number_gridnodes
Pxg(n)= Pxg(n)+ Fxg(n)*dt
end do
!$omp end parallel do
```

where *number_gridnodes* is the total number of grid nodes, and *dt* is the time step interval.

4.1.2 The domain decomposition method

In the domain decomposition method, the background grid is decomposed into some uniform patches. The number of patches is set equal to the number of threads, so that each thread deals with one patch. An example is shown in Fig.5, in which the computational grid is decomposed into four uniform patches. The global IDs of particles are shown in Fig.5(a), and the local IDs of these particles in each patch are shown in Fig.5(b). In our MPM implementation, the same computational grid is used in all time steps, so that the global ID and local ID of a grid node in each patch keep unchanged in all time steps.

Actually, MPM can be viewed as a special Lagrangian FEM, in which the particles rather than the Gauss points serve as the integral points. In FEM, the Gauss points keep in the fixed location in an element in all time steps. After partitioning the finite element domain into patches, the Gauss points are automatically partitioned into different patches. All Gauss points stay in their original element in all time steps. However, the particles in MPM are only rigidly attached to the grid within one time step, and they may locate in different grid cells in different time steps. Therefore, it is necessary to determine the local ID of a particle in a patch. A two-dimensional array called *Pindex* is created to determine the local ID of a particle in a patch, and its element (i, j) determines the global ID of the particle in patch j with local ID i. For example, the array *Pindex* for the problem shown in Fig.5 is given as

$$\mathbf{Pindex} = \begin{bmatrix} 2 & 1 & 8 & 4 \\ 3 & 6 & 10 & 9 \\ 5 & 7 & 14 & 12 \\ 11 & 16 & 0 & 13 \\ 0 & 0 & 0 & 15 \end{bmatrix}$$

The particles are divided into several groups to generate the array *Pindex* in a parallel manner, and each thread deals with a group of particles. For the problem shown in Fig.5, let the first thread deal with the particles 1 to 4, and let the second thread deal with the particles 5 to 8. Let the third thread deal with the particles 9 to 12, and let the fourth thread deal with the particles 13 to 16. A two-dimensional array *patnp_th* is created for counting the particles, whose element (i, j) stores the number of particles in patch j counted by thread i. For the example shown in Fig.5, the array *patnp_th* is given as

$$\mathbf{patnp\_th} = \begin{bmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix}$$

The array *patnp_th* can be generated in a parallel manner. Based on the array *patnp_th*, a one-dimensional array *patnp* is created by summing up each column of the array *patnp_th*, namely, $patnp(i) = \sum_{j=1}^{\text{nthreads}} patnp\_th(j,i)$. The i^{th} element of the array *patnp* is the number of particles in patch *i*. For the example shown in Fig.5, the arry *patnp* is given as

$$\mathbf{patnp} = \begin{bmatrix} 4 \\ 4 \\ 3 \\ 5 \end{bmatrix}$$

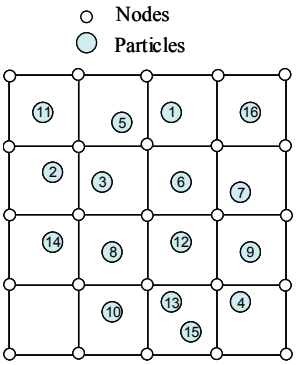
The maximum element of the array *patnp* determines the size of the array *Pindex*. Finally, the array *Pindex* can be created by arrays *patnp_th* and *patnp* in a parallel manner.

Each thread deals with a patch for avoiding data races in the domain decomposition method. According to Eqs. (1) $\sim$ (5), the local variables of nodes in one patch are updated by the following parallelized code.

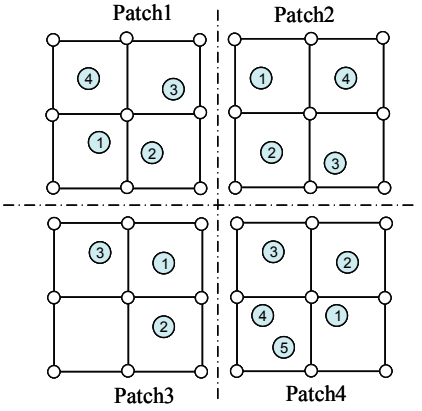
```
!$omp parallel do &
!$omp private (state variables of particles) &
!$omp default(shared)
do patch = 1, patch_number
do i = 1, patnp(patch)
p = Pindex(i,patch)
...
do n = 1,8
subnode = subnodefun(Influence_Node(n),patch)
update Subgrid_list(subnode,patch)%Mg
update Subgrid_list(subnode,patch)%Pxg
update Subgrid_list(subnode,patch)%Fxg
end do
end do
end do
```

where *patch_number* is the total number of patches, and the function *subnode-fun*(i, j) returns the local ID of the grid node i in patch j. *Subgrid_list*(s, j) contains the mass, momentum and force of a grid node, whose local ID is s in patch j.

The nodal mass *Mg*(), momentum *Pxg*() and force *Fxg*() can be assembled from the array *Subgrid_list* by the following parallelized code.



(a) A computational domain and the global IDs of particles



(b) Patches and the local IDs of particles

Figure 5: Decomposition of a background grid domain.

```
do patch = 1, patch_number
!$omp parallel do &
!$omp private(variables) &
!$omp default(shared)
do subnode = 1, patch_node_number
node = globalnode(subnode,patch)
Mg(node)= Mg(node)+Subgrid_list(subnode,patch)%Mg
Pxg(node)=Pxg(node)+Subgrid_list(subnode,patch)%Pxg
Fxg(node)=Fxg(node)+Subgrid_list(subnode,patch)%Fxg
end do
```

```
end do
```

where the function *globalnode*(s, j) returns the global ID of the node with local ID $s$in patch j. Therefore, the nodal variables can be caculated in a parallel manner using the code presented in this section.

4.2 Parallelization of particle variable updates

After solving the momentum equations at grid nodes, the nodal variables are mapped back to particles to update their state variables. This stage is completed over three steps:

(1) Update the position and velocity for each particle.

(2) Update the nodal velocity again.

(3) Update the density and stress for each particle.

The position and velocity of each particle are updated according to Eqs. (7) $\sim$ (10). There are no data races in this step, so that this step can be parallelized directly using the loop splitting method as

```
!$omp parallel do &
!$omp private (state variables of particles) &
!$omp default(shared)
do p =1, number_ particles
update the position of particle p
update the velocity of particle p
end do
```

where *number_ particles* is the total number of particles. To obtain accurate values of nodal velocities, the updated velocities of particles are mapped back to the grid nodes again according to Eq. (11). Data races occur in this step, so that the array expansion method or domain decomposition method need to be used to parallelize this step (see Section 4.1).

The particle strain and stress are updated according to Eqs. (12) $\sim$ (15). There are no data races in this step, so that this step can be parallelized directly using the loop splitting method as

```
!$omp parallel do &
!$omp private (state variables of particles) &
!$omp default(shared)
```

```
do p =1, number_particles
calculate the incremental strain of particle p
calculate the incremental vorticity of particle p
calculate stress of particle p
End do
```

The total kinetic energy is the sum of kinetic energy of all particles. Different particle loops may read and write the value of the total kinetic energy at the same time, so that data races occur. The data races can be avoided using the reduction clause as

```
!$omp parallel do &
!$omp private (state variables of particles) &
!$omp reduction(+: kinetic_energy) &
!$omp reduction(+: internal_energy)
do p =1, number_particles
update kinetic_energy
update internal_energy
end do
```

where *kinetic_energy* and *internal_energy* are the total kinetic energy and internal energy of all particles, respectively.

4.3 Load balance

In the domain decomposition method, particles may move from their original patch to a new patch after some time steps. The number of particles in different patches may have a significant difference, and some patches may have no particles in some time steps. Consequently, a load balance algorithm is required to enhance the parallel efficiency.

A simple load balance algorithm is adopted here. An adaptive background grid is proposed to avoid void patches. In each time step, the size of the background grid is recalculated to fit the new positions of all particles. For example, in Taylor bar impact simulation shown in Fig.6, many cells are not occupied by any particles at time t (see Fig.6(b)). If the traditional background grid is used, the load imbalance between threads is significant. The adaptive background grid shown in Fig.6(c) not only improves the load balance, but also reduces the number of grid nodes. In the future, the better and sophisticated algorithms need to be developed to improve the load balance much more.

5 Numerical examples

Based on aforementioned parallel methods, a parallel 3D MPM code, MPM3DMP, is developed to solve impact dynamics problems. This parallel code is tested on a HP DL140G3 server with two Quad-Core Intel Xeon 5355 processors (2.66GHz) and 8 GB memory. The operating system is Red Hat Enterprise Linux AS release 4. The parallel code is compiled using the Intel FORTRAN 10.0 compiler with OpenMP option.

5.1 Description of benchmark tests

The impact of Taylor bar is taken as a standard benchmark to compare the performance of two parallel methods. The cylinder's material is oxygen-free high-conductivity (OFHC) copper. The initial dimensions of the cylinder are a length of 25.4 mm and a diameter of 7.6 mm. The cylinder impacts against a rigid planar surface with an initial velocity of 40 m/s. In this simulation, the deviatoric stress is updated by the elastic-plastic constitutive model with isotropic hardening,

$$\sigma_y = A + B\bar{\varepsilon}_p^n \tag{17}$$

where A, B and n are the material constants, σ_y is the flow stress, and $\bar{\varepsilon}_p$ is the effective plastic strain. The pressure is updated by the Mie-Grüneisen EOS

$$p = \begin{cases} \frac{\rho_0 c_0^2 \mu(1+\mu)}{[1-(s-1)\mu]^2}\left(1-\frac{\gamma\mu}{2}\right) + \gamma_0 E_n & \mu > 0 \\ \rho_0 c_0^2 \mu + \gamma_0 E_n & \mu < 0 \end{cases} \tag{18}$$

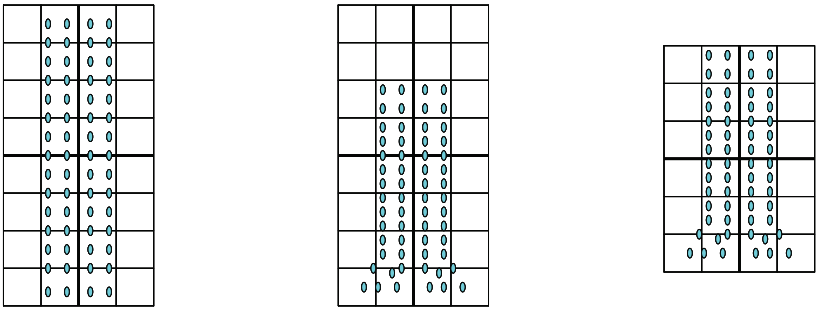
where ρ_0, c_0, s and γ_0 are the material constants, p is the pressure, μ is the compression ratio of relative volume, γ is the Grüneisen coefficient, and E_n is the internal energy per initial volume. The material constants of the copper are given by Johnson and Holmquist (1988), and listed in Table 1.To compare the performance of two parallel methods, different models of Taylor bar are simulated with different number of threads. The number of particles, cells and grid nodes in each model are listed in Table 2.

5.2 Parallel performance

5.2.1 Performance of the array expansion method

The termination time of Taylor bar simulation is set to 40 μs. The results obtained by MPM3DMP agree well with those by LS-DYNA software. The speedup s_p is defined as

$$s_p = \frac{T_s}{T_m} \tag{19}$$



(a) Initial background grid (b) Traditional background grid at t (c) Adaptive background grid at t

Figure 6: Adaptive background grid for load balance.

Table 1: Material constants of copper

	Copper
Density(kg/m^3)	8930
Young's Modulus (GPa)	117.0
Poisson's Ratio	0.35
Constitutive Model Constants	
A(MPa)	157.0
B(MPa)	425.0
n	1.0
Equation of State Constants	
c_0(m/s)	3940
s	1.49
γ_0	1.96

Table 2: The parameters of different MPM models

	Particles	Cells	Nodes
Model1	56056	49152	53361
Model2	244524	165888	175273
Model3	1155192	1022208	1053493
Model4	12416320	8177664	8302185

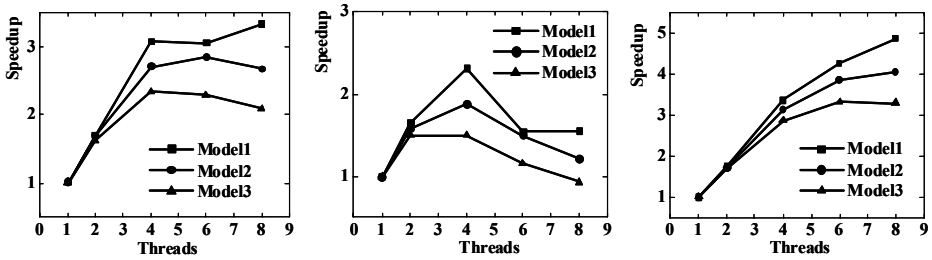
where T_s is the CPU time using single thread, and T_m is the CPU time using multiple threads. The parallel efficiency e_f is defined as

$$e_f = \frac{s_p}{n} \tag{20}$$

where n is the total number of threads used in the parallel computation. To investigate the effects of compiler optimization on parallel performance, the parallel code is compiled by Intel FORTRAN 10.0 compiler with O3 optimization and without using any optimization options, respectively.

The speedup and parallel efficiency of the array expansion parallel method with O3 compiler optimization are shown in Fig.7 and Fig.8, respectively. Meanwhile, the speedup and parallel efficiency without using any optimization compiler options are shown in Fig.9 and Fig.10, respectively. Fig.9 and Fig.10 show that the parallel performance without using any compiler optimization is higher than that with O3 compiler option.

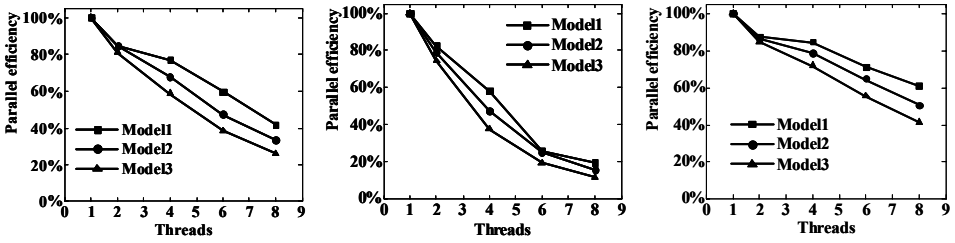
In the array expansion method, the sizes of auxiliary arrays *Mg_list*, *Pxg_list* and *Fxg_list* increase significantly with increasing threads, which will lead to a significant increase in computational effort and memory requirement. Consequently, the parallel performance of the array expansion method becomes poor with the increase of model size, as shown in Fig.9 (a) and Fig. 10 (a).



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

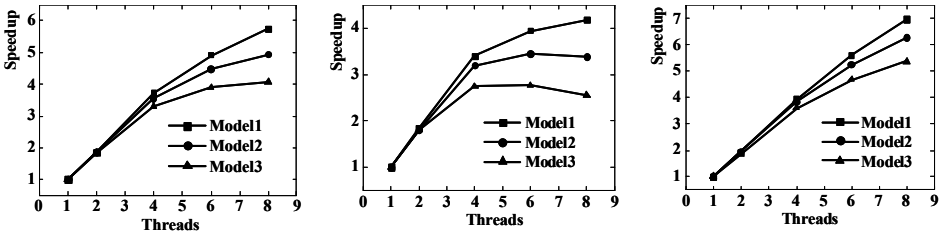
Figure 7: Speedup of the array expansion method with O3 compiler optimization.

To investigate the parallel performance of different stages in MPM, the speedup and parallel efficiency for updating grid nodes and particles are also shown in Fig.9 and Fig.10, respectively. Fig.9 (b) and Fig.9 (c) show that the code for updating particle has better parallel performance than the code for updating grid nodes. Thus, as a bottleneck of the array expansion method, the grid nodes update stage may lead to poor parallel performance, especially when many threads or large-scale models are used.



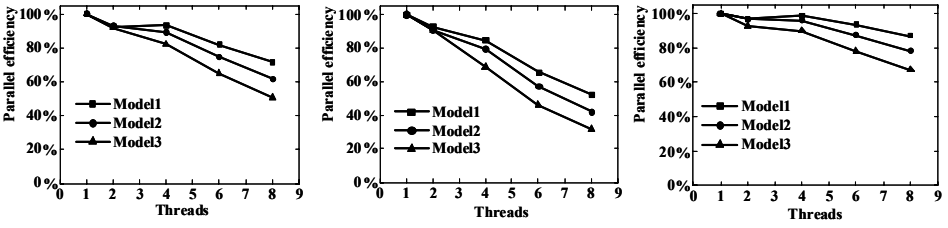
(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 8: Parallel efficiency of the array expansion method with O3 compiler optimization.



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 9: Speedup of the array expansion method without using compiler optimization.



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 10: Parallel efficiency of the array expansion method without using compiler optimization.

In addition, the memory requirement increases significantly with increasing threads in the array expansion method. Consequently, the results for model 4 with 12, 416, 320 particles are not available under 4 to 8 threads because there is no enough

memory available in the server. This study shows that the array expansion method is not well suitable for the large-scale MPM models.

5.2.2 Performance of the domain decomposition method

In the domain decomposition method, $1\times1\times1$, $2\times1\times1$, $2\times2\times1$, $2\times1\times3$ and $2\times2\times2$ patches are used for 1, 2, 4, 6 and 8 threads, respectively. The speedup and parallel efficiency of the domain decomposition method with O3 compiler optimization are shown in Fig.11 and Fig.12, respectively. Meanwhile, the speedup and parallel efficiency without using any compiler optimization options are shown in Fig.13 and Fig.14, respectively. Fig.13 and Fig.14 also show that the parallel performance without using any compiler optimization is higher than that with O3 compiler optimization.

In the domain decomposition method, the memory requirement almost keeps constant with the increase of threads. For example, when the number of threads is increased from 1 to 8, the memory requirement is only increased from 4.61 GB to 4.63 GB for the model 4.

Note that this test is conducted on a server with two quad-core processors, so that communications between two processors are required if more than 4 threads are used. Consequently, the parallel efficiency will decrease when the number of threads exceeds 4, as shown in Fig.13 and Fig.14.

5.3 Application to hypervelocity impact

Due to the extreme mesh distortion, conventional finite element codes have been proven inefficient in the simulation of hypervelocity impact problems like the design of orbital debris shielding. The hypervelocity impact problems were solved by some new numerical methods such as SPH [Mehra and Chaturvedi (2006)], MLPG [Han, Liu, Rajendran and Atluri (2006)], the generalized particle algorithm [Beissel, Gerlach and Johnson (2006)] and the hybrid particle-finite element method [Park and Fahrenthold (2005)]. To illustrate the capability of our code in the simulation of hypervelocity impact problems, the impact of a spherical lead projectile with a velocity of 6.58 km/s on a thin lead plate is examined. The large-scale MPM simulations for hypervelocity impact are performed using MPM3DMP code based on the domain decomposition method.

The radius and mass of the projectile are 7.5 mm and 20 g, respectively. The thickness of the plate is 6.35 mm. The deviatoric stress of each particle is updated by an elastic-plastic constitutive model with isotropic hardening, which is described by Eq.(17). The pressure of each particle is updated by the Mie-Grüneisen equation of state, which is described by Eq.(18). A failure model is used to simulate the fail-

ure of material. Failure is assumed to occur based on either or all of the following conditions:

(1) The effective plastic strain reaches the plastic strain $\bar{\varepsilon}_f$ at fracture.

(2) The pressure is less than the failure pressure $p_{\min}$.

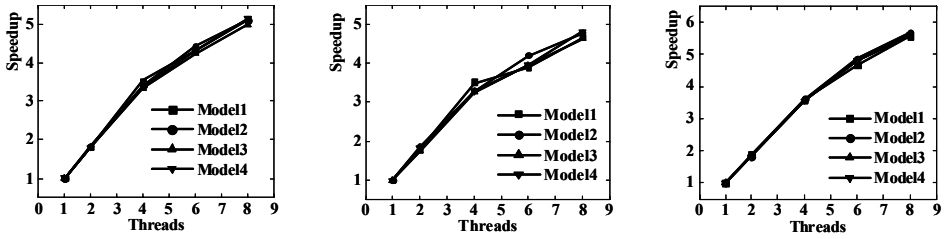
where $p_{\min}$ and $\bar{\varepsilon}_f$ are user-defined parameters. Once failure has occurred, the stress of the failure particle is set to zero. All material constants of lead are listed in Table 3.

Table 3: Material constants of lead

	Lead
Density(kg/m^3)	11350
Young's Modulus (GPa)	22.4
Poisson's Ratio	0.42
Constitutive Model Constants	
A(MPa)	12.0
B(MPa)	125.0
n	1.0
Equation of State Constants	
c_0(m/s)	2092
s	1.45
γ_0	2.0
Failure Parameters	
$\bar{\varepsilon}_f$	3.0
$p_{\min}$(MPa)	-1500

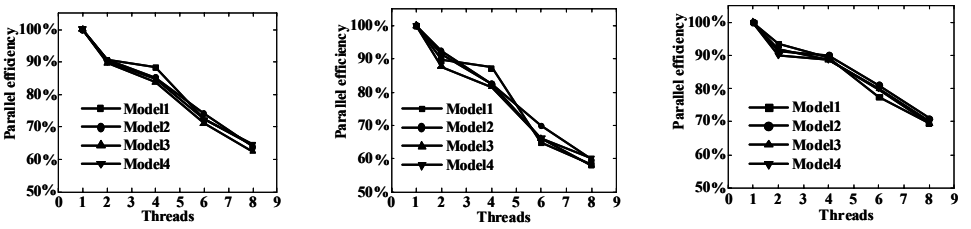
Fig.15 shows the experimental radiograph of debris cloud at 30.6 μs [Anderson, Trucano and Mullin (1990)]. A small-scale model is first used to investigate the effect of grid cell sizes on computational results, which includes 847,888 particles with an initial particle space of 0.53 mm. With the fixed number of particles, Fig.16 compares the debris cloud shapes at 30.6 μs obtained by using different cell sizes. Fig.16 also shows that the results obtained by a cell length of 2.12 mm, namely $4\times4\times4$ particles per cell, have better agreements with the experiment results.

Furthermore, a large-scale model is used to investigate the effect of the number of particle on simulation results, which includes 13,542,030 particles with an initial particle space of 0.21 mm. The length of the grid cell is set to 1 mm in this large-scale simulation, which approaches to the cell length of 1.06 mm used in the



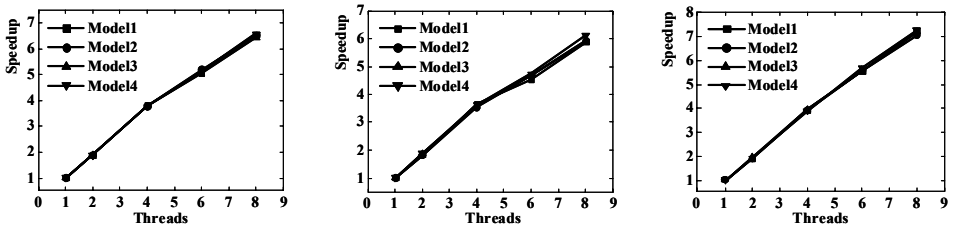
(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 11: Speedup of the domain decomposition method with O3 compiler optimization.



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

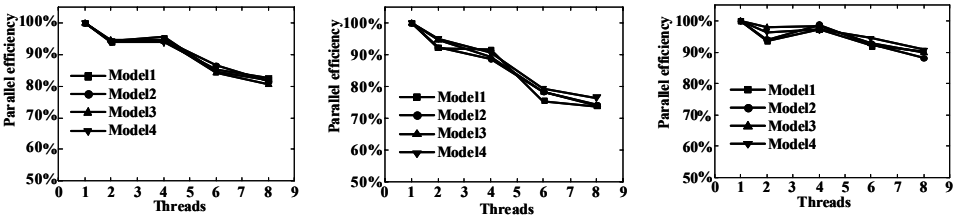
Figure 12: Parallel efficiency of the domain decomposition method with O3 compiler optimization.



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 13: Speedup of the domain decomposition method without using compiler optimization.

small-scale simulation (see Fig.16(a)). The debris cloud at 30.6 μs obtained by the large-scale model is shown in Fig.17. In the experimental radiograph (see Fig. 15), the front of the debris cloud is approximately 200 mm away from the plate, and the width of debris cloud is approximately 145 mm. From the large-scale computational results shown in Fig. 17, the front of the debris cloud is 198 mm away from



(a) Total computation (b) Stage for updating grid nodes (c) Stage for updating particles

Figure 14: Parallel efficiency of the domain decomposition method without using compiler optimization.

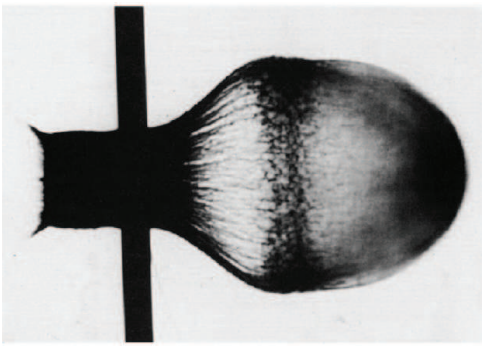
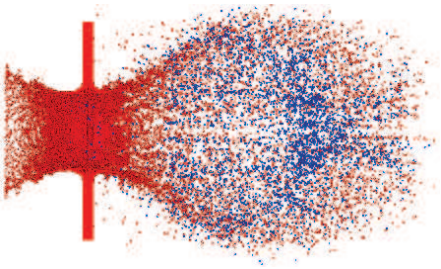


Figure 15: Experimental radiograph at 30.6 μs.

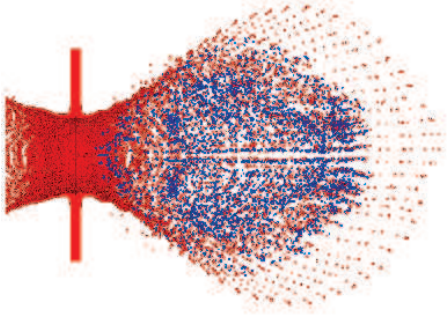
the plate, and the width of debris cloud is 142 mm. Therefore, the profile and size of debris cloud obtained by the large-scale model agree well with the experimental results. Fig.16 and Fig.17 show that the shapes of clouds obtained by the small-scale and large-scale models are similar, but the large-scale model improves the results accuracy significantly. This study shows that the large-scale MPM computation is required to obtain the high-resolution debris cloud results for hypervelocity impact.

6 Discussions and conclusions

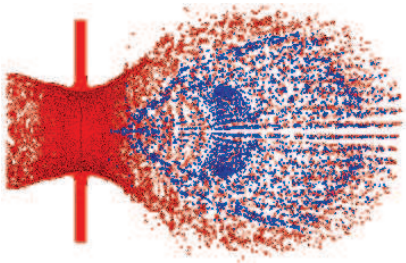
In many situations, the large-scale MPM computation is required to obtain high-resolution results. A parallel MPM code, MPM3DMP, is developed using OpenMP for solving impact dynamic problems. The particle update stage can be easily parallelized by using the loop splitting method, whereas the parallelization of the nodal update stage is much more complicated due to data races. In this study, the array expansion method and the domain decomposition method are used to avoid the data



(a) Cell length =1.06 mm



(b) Cell length =1.59 mm



(c) Cell length =2.12 mm

Figure 16: Effect of grid cell sizes on the simulation results. The model includes 847,888 particles with an initial particle space of 0.53 mm.

races, respectively.

In the array expansion method, the sizes of auxiliary arrays for node variables will increase with increasing threads, which result in poor parallel efficiency in large-scale MPM computation. Moreover, the memory requirement will also increase with increasing threads, so that the array expansion method is only effective in small-scale MPM computation.

In the domain decomposition method, the background grid is decomposed to some

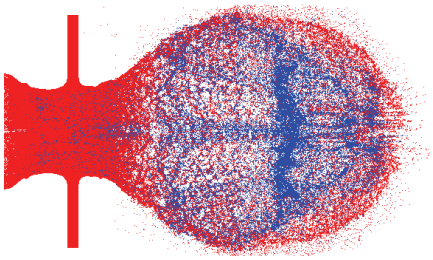


Figure 17: Debris cloud at 30.6 μs obtained by the large-scale model, which includes 13,542,030 particles with an initial particle space of 0.21 mm. The cell length is set to 1.0 mm.

uniform patches, and each thread computes one patch, so that the data races of updating nodes are avoided. The numerical tests show that the domain decomposition method possesses much better scalability and higher efficiency than the array expansion method.

The parallel code MPM3DMP is used to solve hypervelocity impact problems. This study shows that large-scale MPM computation is essential to obtain the high-resolution debris cloud results. Although the shapes of debris cloud obtained by the small-scale and large-scale models are similar, the large-scale model improves the results accuracy significantly. In near future, the parallel efficiency of MPM3DMP will be further improved using a better load balance algorithm and the hybrid OpenMP and MPI programming technique.

Acknowledgement: This project was supported by a grant from the National Natural Science Foundation of China (10872107), and the Science Fundation of Computational Physics, IAPCM,China. The authors gratefully acknowledge Ph.D candidate Jie Cai of Australian National University, and Prof. Qingnan Huang of China Academy of Engineering Physics (CAEP) for their helpful discussions of parallel methods. The authors also acknowledge Dr. Xiaowei Chen of CAEP for his helpful suggestions.

References

Ayguadea, E.; Gonzaleza, M.; Martorella, X.; Jostb, G. (2006): Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. *Journal of Parallel and Distributed Computing*, vol. 66, pp. 686-697.

Anderson, C. E. J.; Trucano, T. G.; Mullin, S. A. (1990): Debris cloud dynamics.

International Journal of Impact Engineering, vol. 9, pp. 89-113.

Bardenhagen, S. (2002): Energy conservation error in the material point method for solid mechanics. *Journal* of *Computational Physics*, vol. 180, pp. 383-403.

Bardenhagen, S.; Kober, E. (2004): The generalized interpolation material point method. *CMES: Computer Modeling in Engineering and Sciences*, vol. 5, pp. 477-495.

Bardenhagen, S. G.; Brackbill, J.U.; Sulsky, D. (2000): The material point method for granular materials. *Computer Methods in Applied Mechanics Engineering,* vol. 187, pp. 529-541.

Beissel, S. R.; Gerlach, C. A.; Johnson, G .R. (2006): Hypervelocity impact computations with finite elements and meshfree particles. *International Journal of Impact Engineering*, vol. 33, pp. 80-90.

Brackbill, J. U.; Kothe, D. B.; Ruppel, H. M. (1988): FLIP: a low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, vol. 48, pp.25-38.

Candel, A. E.; Dehler, M. M.; Troyer, M. (2006): A massively parallel particle-in-cell code for the simulation of field-emitter based electron sources, *Nuclear Instruments and Methods in Physics Research A*, vol. 558, pp. 154-158.

Chandra, R.; Dagum, L.; Kohr, D.; Maydan, D.; McDonald, J.; Menon, R. (2001): *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers.

Chapman, B.; Jost, G.; VanderPas, R. (2007): *Using OpenMP : portable shared memory parallel programming*, The MIT Press.

Chen, Z.; Brannon, R. (2002): An Evaluation of the Material Point Method. Technical Report SAND2002-0482, Sandia National Laboratory, Sandia, 2002.

Chen, Z.; Gan, Y.; Chen, J. K. (2008): A coupled thermo-mechanical model for simulating the material failure evolution due to localized heating, *CMES: Computer Modeling in Engineering and Sciences*, vol. 26, no. 2, pp. 123-137.

Couturier, R.; Chipot, C. (2000): Parallel molecular dynamics using OPENMP on a shared memory machine. *Computer Physics Communications*, vol. 124, pp. 49-59.

Goedecker, S. (2002): Optimization and parallelization of a force field for silicon using OpenMP. *Computer Physics Communications*, vol. 148, pp. 124-135.

Guilkey, J. E.; Harman, T. B.; Banerjee, B. (2007): An Eulerian-Lagrangian approach for simulating explosions of energetic devices. *Computers and Structures*, vol. 85, pp. 660-674.

Guilkey, J. E.; Weiss, J. A. (2003): Implicit time integration for the material point

method: quantitative and algorithmic comparisons with the finite element method. *International Journal for Numerical Methods in Engineering*, vol. 57, pp. 1323-1338.

Guo, Y.; Nairn, J. A. (2004): Calculation of *J* integral and stress intensity factors using the material point method. *CMES: Computer Modeling in Engineering and Sciences*, vol. 6, pp. 295-308.

Guo, Y.; Nairn, J. A. (2006): Three-Dimensional Dynamic Fracture Analysis Using the Material Point Method. *CMES: Computer Modeling in Engineering and Sciences*, vol. 16, pp. 141-156.

Hallquist, J. O. (1998): LS-DYNA Theoretical Manual. Livermore Software Technology Corporation, 1998.

Han, Z. D.; Liu, H. T.; Rajendran, A. M; Atluri, S. N. (2006): The applications of Meshless Local Petrov-Galerkin (MLPG) approaches in high-speed impact, penetration and perforation problems. *CMES: Computer Modeling in Engineering and Sciences,* vol. 14, no. 2, pp. 119-128.

Harlow, F. H. (1963): The particle-in-cell computing method for fluid dynamics. *Methods in Computational Physics,* vol. 3, pp. 319-343.

Hu, W.; Chen, Z. (2003**):** A multi-mesh MPM for simulating the meshing process of spur gears. *Computers and Structures,* vol. 81, pp. 1991-2002.

Hu, W.; Chen, Z. (2006): Model-based simulation of the synergistic effects of blast and fragmentation on a concrete wall using the MPM. *International Journal of Impact Engineering*, vol. 32, pp. 2066-2096.

Johnson, G. R.; Holmquist, T. J. (1988): Evaluation of cylinder-impact test data for constitutive model constants. *Journal of Applied Physics,* vol. 64, pp. 3901-3910.

Ma, J.; Lu, H.; Komanduri, R. (2006): Structured mesh refinement in generalized interpolation material point (GIMP) method for simulation of dynamic problems. *CMES: Computer Modeling in Engineering and Sciences*, vol. 12, no. 3, pp. 213-227.

Ma, J.; Lu, H.; Wang, B.; Roy, S.; Hornung, R.; Wissink, A.; Komanduri, R. (2005): Multiscale simulations using generalized interpolation material Point (GIMP) method and SAMRAI parallel processing. *CMES: Computer Modeling in Engineering and Sciences,* vol. 8, no. 2, pp. 135-152.

Ma, S.; Zhang, X.; Qiu, X. M. (2009): Comparison study of MPM and SPH in modeling hypervelocity impact problems. *International Journal of Impact Engineering,* vol. 36, pp. 272-282.

Martin, M. J.; Papada, M.; Doallo, R. (2004): High Performance Air Pollution

Simulation Using OpenMP. *The Journal of Supercomputing*, vol. 28, pp. 311-321.

Mehra, V.; Chaturvedi, S. (2006): High velocity impact of metal sphere on thin metallic plates: a comparative smooth particle hydrodynamics study. *Journal of Computational Physics*, vol. 212, pp. 318-337.

Nairn, J. A. (2003): Material Point Method Calculations with Explicit Cracks. *CMES: Computer Modeling in Engineering and Sciences*, vol. 4, pp. 649-663.

Pan, X. F.; Xu, A. G.; Zhang, G. C.; Zhu, J. S.; Ma, S.; **Zhang, X.** (2008): Three-dimensional multi-mesh material point method for solving collision problems. *Communications in Theoretical Physics*, vol. 49, pp. 1129-1138.

Pantalé, O. (2005): Parallelization of an object-oriented FEM dynamics code: influence of the strategies on the Speedup. *Advances Engineering Software*, vol. 36, pp. 361-373.

Park, Y. K.; Fahrenthold, E. P. (2005): A kernel free particle-finite element method for hypervelocity impact simulation. *International Journal for Numerical Methods in Engineering*, vol. 63, pp. 737-759.

Parker, S. G. (2006): A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Computer Systems*, vol. 22, pp. 204-216.

Parker, S. G.; Guilkey, J.; Harman, T. (2006): A component-based parallel infrastructure for the simulation of fluid-structure interaction. *Engineering with Computers*, vol. 22, pp. 277-292.

Quinn, M. J. (2004): Parallel Programming in c with MPI and OpenMP. McGraw-Hill Companies, New York.

Reynders, J. V. W.; Hinker, P. J.; Cummings, J. C.; Atlas, S. R.; Banerjee, S.; Humphrey, W. F.; Karmesin, S. R.; Keahey, K.; Srikant, M.; Tholburn, M. (1996): POOMA: A framework for scientific simulations on parallel architectures, In: G.V. Wilson and P. lu (ed). *Parallel Programming using C++*, MIT Press, pp. 553-594.

Schreyer, H. L.; Sulsky, D. L.; Zhou, S. J. (2002): Modeling delamination as a strong discontinuity with the material point method. *Computer Methods in Applied Mechanics Engineering*, vol. 191, pp. 2483-2507.

Shen, L.; Chen, Z. (2005): A silent boundary scheme with the material point method for dynamic analyses. *CMES: Computer Modeling in Engineering and Sciences,* vol. 7, no. 3, pp. 305-320.

Sosa, C. P.; Scalmani, G.; Gomperts, R.; Frisch, M. J. (2000): Ab initio quantum chemistry on a ccNUMA architecture using openMP. III. *Parallel Computing*, vol. 26, pp. 843-856.

Steffen, M.; Wallstedt, P. C.; Guilkey, J. E.; Kirby, R. M.; Berzins, M. (2008):

Examination and analysis of implementation choices within the material point method (MPM). *CMES: Computer Modeling in Engineering and Sciences*, vol. 31, no. 2, pp. 107-127.

Sulsky, D.; Chen, Z.; Schreyer, H. (1994): A particle method for history-dependent materials. *Computer Methods in Applied Mechanics Engineering*, vol. 118, pp. 179-196.

Sulsky, D.; Kaul, A. (2004): Implicit dynamics in the material-point method. *Computer Methods in Applied Mechanics and Engineering*, vol. 193, no. 12-14, pp. 1137-1170.

Sulsky, D.; Schreyer, H. (2004): MPM Simulation of Dynamic Material Failure with a Decohesive Constitutive Model. *European Journal of Mechanics A/Solids*, vol. 23, pp: 423-445.

Sulsky, D.; Schreyer, H. L. (1996): Axisymmetric form of the material point method with applications to upsetting and Taylor impact problems. *Computer Methods in Applied Mechanics Engineering*, vol. 139, pp. 409-429.

Sulsky, D.; Zhou, S.; Schreyer, H. (1995): Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, vol. 87, pp. 236-252.

Tan, H.; Nairn, J. A. (2002): Hierarchical Adaptive Material Point Method in Dynamic Energy Release Rate Calculations. *Computer Methods in Applied Mechanics and Engineering*, vol. 191, pp. 2095-2109.

Thacker, R. J.; Couchman, H. M. P. (2006): A parallel adaptive P3M code with hierarchical particle reordering. *Computer Physics Communication*, vol. 174, pp. 540-554.

York, A. R. II.; Sulsky, D.; Schreyer, H. L. (1999): The material point method for simulation of thin membranes. *International Journal for Numerical Methods in Engineering*, vol. 44, pp. 1429−1456.

Zhang, D. Z.; Zou, Q.; VanderHeyden, W. B.; Ma, X. (2008): Material point method applied to multiphase flows. *Journal of Computational Physics,* vol. 227, pp. 3159-3173.

Zhang, X.; Sze, K. Y.; Ma, S. (2006): An explicit material point finite element method for hyper-velocity impact. *International Journal for Numerical Methods in Engineering*, vol. 66, pp. 689-706.