 CMES, vol.69, no.2, pp.143-165, 2010

An Alternated Grid Updating Parallel Algorithm for Material Point Method Using OpenMP

Yantao Zhang[1], Xiong Zhang[1, 2] and Yan Liu[1]

Abstract: Material point method(MPM) is a promising method in solving problems involving large deformations, especially explosion and penetration. In MPM, particles can move around the computing domain dynamically, which can result in load imbalance easily. In parallelizing MPM using OpenMP, data races will occur in the stage of grid node updating if we use loop-level parallelism for these loops. Huang et al. proposed a domain decomposition method to overcome data races [Huang, Zhang, Ma and Wang (2008)]. However, significant modifications of the original serial code are required. In this paper, we proposed a new alternated grid updating method to avoid data races in the stage of grid node updating, which only need small modifications of the original code and is much easier to achieve dynamic load balance. Test results show that our algorithm performs well.

Keywords: material point method, OpenMP, parallel, alternated grid updating, dynamic load balance

1 Introduction

MPM is an extension of particle-in-cell(PIC) method [Harlow (1964)] to solid mechanics by Sulsky et al. [Sulsky, Chen, Schreyer (1994); Sulsky, Zhou, Schreyer (1995)], and has been successfully used to various problems, such as Taylor bar impact [Sulsky, Zhou, Schreyer (1995)], metal forming problem [Sulsky, Schreyer (1996)], membrane [York, Sulsky, Schreyer (1999)], hyper-velocity impact [Zhang, Sze, Ma (2006)], granular media [Guilkey, Weiss (2003); Bardenhagen, Brackbill and Sulsky (2000)], multi-scale problem [Liu, Zhang, Sze and Wang (2007); Ma, Liu, Lu and Komanduri (2006)], dynamic fracture [Guo, Nairn (2006); Daphalapurkar, Lu, Coker and Komanduri (2007)], explosive process [Hu, Chen (2006); Guilkey, Harman and Banerjee (2007)], just to name a few. And all kinds of enhancements to MPM, such as GIMP shape function [Bardenhagen, Kober (2004)],

[1] School of Aerospace, Tsinghua University, Beijing 100084, China.
[2] Corresponding author: xzhang@tsinghua.edu.cn

adaptive algorithm [Ma, Zhang and Huang (2010)], contact algorithm [York, Sulsky, Schreyer (1999); Bardenhagen, Brackbill and Sulsky (2000); Bardenhagen, Guilkey and Roessig (2001)] have been widely used.

Material Point Method (MPM) is a promising method in solving problems involving large deformation and fracture. In solving engineering problems such as explosion in building, penetration through whipple shield plates, the problem scale goes well beyond common PC's capability. Therefore, development of parallel algorithm for MPM is desirable.

Danielson developed a parallel algorithm for Reproducing Kernel Particle Method (RKPM) using MPI [Danielson, Hao, Liu, Aziz Uras and Li (2000)], which is also a kind of meshless method. A graph-based decomposition tool, named Metis, was employed in the pre-analysis phase, in which integration points were uniquely defined on separate processors and all support particles for each point were defined locally on the corresponding processor, thus load balance was easy to reach.

Plimpton proposed "window" to solve the load imbalance of grid updating and particle updating independently in Particle-in-cell(PIC) method [Plimpton, Seidel, Pasik, Coats, and Montry (2003)], and tested this algorithm in two applications of extreme load imbalance. The result was good, but the communication scheme used was complicated.

Othmer made use of "one-side communication" in MPI-2 standard to realize "task-farm" strategy of dynamic load balancing [Othmer, Schüle (2002)]. This method saved the trouble of complex communication. A 1D electron-ion beam problem was simulated and the efficiency can be as up as 80% when using 128 processes.

Parker developed a parallel framework Uintah [Parker (2006)], where MPM is implemented. It mainly employs domain decomposition method. Using this infrastructure, an application of a fluid-structure interaction problem was carried out [Parker, Guilkey and Harman (2006)] and performance was quite good.

All the endeavors above are realized on distributed memory platform using MPI. In shared memory machines such as PC computers and notebooks with multi-core and hyper threading technologies, parallelization for MPM could be achieved using OpenMP, which uses thread based parallel model. As it supports incremental parallelization and may requires only small modifications of the serial code, it has been used in many applications and performs very well.

Jin parallelized a series of NAS benchmarks with OpenMP [Jin, Frumkin and Yan (1999)]. A few of them are finite difference method, in which large arrays are used commonly. In these problems load balance is automatically achieved when using OpenMP directive for loops. This article mainly introduces efficient use of arrays and cache friendliness. Most of these features of OpenMP have been summarized

in [Chapman, Jost and Pas (2007)].

Jay Hoeflinger used OpenMP to parallelize ROCFIRE, a program of fourth-order central difference scheme, and translated ROCFLO, a CFD code of finite volume method and parallelized using MPI, into OpenMP version [Hoeflinger, Alavilli, Jakson and Kuhn (2001)]. Although Jay achieved rather good scalability and efficiency, unknown factors of the SGI OpenMP compiler played a very important role.

Most of these applications do not need domain decomposition. OpenMP's loop-level parallelism are adequate for them to achieve good speedup and no data races occur at the same time. Thus no load imbalance exists in parallel finite difference method and finite volume method. But inhomogeneous distribution of particles is a inherent character of MPM, which can easily result in load imbalance. In parallelizing MPM, the top two difficulties are load imbalance and data race. In [Huang, Zhang, Ma and Wang (2008)], OpenMP is successfully used to parallelize MPM for the first time. Huang achieved a good scalability in the Taylor bar test. Stantchev proposed a Cell Based Particle Pull(CBPP) method [Stantchev, Dorland and Gumerovb (2008)] in parallel plasma simulation with PIC using GPU. Different from our program, which is based on "particle push", this method can prevent data races as long as OpenMP directives are used. However, significant modifications to the original serial code are required if it is applied to our program.

Huang used domain decomposition to update grid in parallel [Huang, Zhang, Ma and Wang (2008)]. He created an accessory array for each thread to prevent data races. In the end of parallel region, these arrays are assembled together to form the array used by the global grid. This method requires significant modifications of the original serial code. In this paper, we proposed a new method to update grid nodes in parallel, and then developed a dynamic load balance algorithm to improve efficiency.

This paper is organized as follows. The basic formulation of MPM and a C++ MPM code, MPM3D, are briefly reviewed in section 2, and parallel programming using OpenMP is introduced in section 3. MPM is parallelized using OpenMP based on the proposed alternated grid updating algorithm in section 4, and a dynamic load balance strategy is proposed in section 5 to further improve the performance. The proposed scheme is tested on a shared memory machine by several numerical examples in section 6, and some conclusions are drawn in section 7.

2 Material Point Method

MPM discretizes bodies by both grid and particles, as shown in Fig. 1. All the variables such as position, momentum, force, stress and strain lie on particles. Grid

is merely auxiliary, helping to solve the momentum equations. In each time step, particles variables are mapped to regular grid nodes, where the momentum equations are solved. Finally, new variables on grid nodes are mapped back to particles to update their state variables. In the next step, the deformed grid is deserted, and a new regular grid is rebuilt for solution.

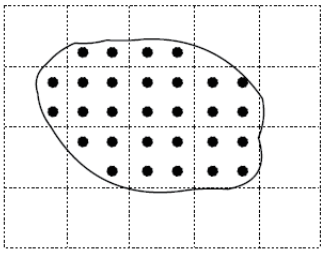

Figure 1: Discretization scheme of MPM, solid dots denote particles, dash lines denote background grid, solid line denotes body boundary

Let subscripts i and p denote variables associated to the grid node i and the particle p, respectively, and superscript k denote variables for kth time step. One cycle of the compute flow of the standard MPM will be generalized as the following [Sulsky, Zhou, Schreyer (1995)]:

a). Mass, momentum and force are mapped from particles to grid nodes using shape function as

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

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

where m_i^k and $\mathbf{p}_i^k$ denote the grid nodal mass and momentum respectively, and S_{ip}^k is the value of shape function of grid node i evaluated at the site of particle p.

b). Boundary condition is applied on grid nodes.

c). Grid nodal forces are updated by

$$\mathbf{f}_i^k = \mathbf{f}_i^{\text{int},k} + \mathbf{f}_i^{\text{ext},k} \tag{3}$$

where

$$\mathbf{f}_i^{\text{int},k} = -\sum_p \sigma_p^k \mathbf{G}_{ip}^k \frac{m_p}{\rho_p^k} \tag{4}$$

$$\mathbf{f}_i^{\text{ext},k} = \sum_p m_p S_{ip}^k \mathbf{b}_p^k + \int_{\Gamma_t} N_i^k \bar{\mathbf{t}}^k \mathrm{d}\Gamma \tag{5}$$

Here, $\mathbf{G}_{ip}^k$ is the derivative of shape function S_{ip}^k, $\mathbf{b}$ is the body force, and $\bar{\mathbf{t}}$ is the traction applied on the boundary.

d). Momentum on the grid nodes are updated by

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

where Δt is the time step size.

e). Velocity and acceleration are mapped back to particles to update their position $\mathbf{x}_p^{k+1}$ and velocity $\mathbf{v}_p^{k+1}$ as

$$\mathbf{x}_p^{k+1} = \mathbf{x}_p^k + \sum_i \frac{\mathbf{p}_i^{k+1}}{m_i^k} S_{ip}^k \Delta t \tag{7}$$

$$\mathbf{v}_p^{k+1} = \mathbf{v}_p^k + \sum_i \frac{\mathbf{f}_i^k}{m_i^k} S_{ip}^k \Delta t \tag{8}$$

f). For Modified Update Stress Last(MUSL) [Chen, Brannon (2002); Nairn (2003)] scheme, velocities of particles are mapped back to grid nodes as

$$\mathbf{v}_i^{k+1} = \frac{1}{m_i^k} \sum_p m_p \mathbf{v}_p^{k+1} S_{ip}^k \tag{9}$$

g). The essential boundary conditions are applied again.

h). Incremental strain and vorticity of particles are updated,

$$\Delta \varepsilon_p = \frac{\Delta t}{2} \sum_i \left[\left(\mathbf{G}_{ip}^k \mathbf{v}_i^{k+1} \right)^T + \mathbf{G}_{ip}^k \mathbf{v}_i^{k+1} \right] \tag{10}$$

$$\Delta \omega_p = \frac{\Delta t}{2} \sum_i \left[\left(\mathbf{G}_{ip}^k \mathbf{v}_i^{k+1} \right)^T - \mathbf{G}_{ip}^k \mathbf{v}_i^{k+1} \right] \tag{11}$$

Finally, a constitutive model is called to update the stresses of particles.

MPM3D [Ma, Zhang and Huang (2010)] is a three dimensional MPM code developed by the authors using C++ programming language, in which steps a), c), e), f) and h) are implemented by looping over all particles, while steps b), d) and g) are by looping over all grid nodes. Details of MPM3D can be found in [Ma, Zhang and Huang (2010)].

3 Parallel Programming OpenMP

OpenMP is not a new programming language but a standard definition of a collection of directives, runtime library routines and environment variables that can specify shared-memory parallelism in C, C++ and Fortran programs. The programming model realized by OpenMP is suitable for shared-memory (or shared-address space) machines. All threads in the parallel region can access some or all of the memory when being executed.

Parallelization of an program using OpenMP can usually be implemented incrementally because OpenMP realizes its parallelism through fork/join mode, see Fig. 2. OpenMP program starts its execution with only one thread called master thread. When encountering a parallel region, e.g. a loop, the master thread will fork into a set of threads called slave thread, each executing part of computing. At the end of this parallel region, all threads join together and only master thread (usually thread numbered 0) continues to run. As we can see in Fig. 2, there may be more than one parallel regions all over the program, so we can conveniently parallelize a serial program block after block, or incrementally. We can also examine the parallel efficiency of each block, providing a way to improve the total efficiency incrementally. But fork and join will cause overload inevitably, so we should decrease the number of parallel regions as much as possible.

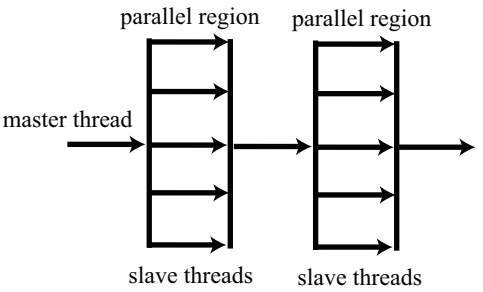

Figure 2: fork/join model of OpenMP parallelism

In C++ language, a loop will be parallelized easily by putting

```
#pragma omp parallel for
```

directive before the loop body. This is called loop-level parallelism, in which all loop iterations are decomposed and distributed among threads. This is also the most common usage of realizing parallelism using OpenMP, and can only be used when different iterations of the loop are independent.

A much better way of parallelization is creating a parallel region using

```
#pragma omp parallel
```

directive. We can distribute work among all threads manually in this parallel region. Besides loops, this method can be applied to many other situations and is much more flexible.

The most important issue in parallelizing a program using OpenMP is preventing data races. As the name shows, data races happen when different threads try to change one variable at the same time. The OpenMP standard have provided many synchronization mechanisms to prevent this, such as "barrier", "atomic", "critical", and "ordered" directives. Using "critical" directive is in fact letting threads execute code in the critical region sequentially, thus reducing efficiency greatly. "atomic" and "ordered" are similar to "critical". "barrier" will create an explicit point where those threads which have arrived will not continue until all threads have arrived. If load is not balanced, threads which arrive early will wait for those which arrive late and thus waste a lot of time. In this paper, a new method to prevent data races in parallelizing MPM is proposed.

4 Parallelizing MPM using OpenMP

The most time consuming parts in MPM are the grid node updating and particle updating. In the grid node updating, state variables on grid node such as mass, momentum and force are updated, while in the particle updating, the position, velocity, strain and stress of particles are updated.

4.1 Grid node updating

In the first stage of MPM, mass and momentum on all particles are mapped to nearby grid nodes through shape function. In MPM3D, this stage is realized by looping over all particles, whose pseudo codes are:

```
for(int i=0;i<nb_point;i++){
    Particle = particle_list[i];
    <map variables on Particle to all grid nodes
     of the cell containing the particle>
}
```

where `nb_point` is the total number of particles.

As the grid is shared by all threads, if one simply adds

```
#pragma omp parallel for
```

directive before this stage, two or more threads may update the same grid node thus result in data races. Huang's method [Huang, Zhang, Ma and Wang (2008)]

decomposes the computing grid into several subgrids and creates accessory array for each subgrid to store its variables. Map between global ID and local ID of grid node is also created, which will be used in the step of assembling these accessory arrays into the array used by the global grid. This step successfully prevents data races that might happen on the boundary between threads, but results in waste of memory and the implementation is also pretty complicated.

Here we propose a new method to update state variables on grid nodes in parallel which avoide the complex manipulation of subgrid and waste of memory. Take the 2D problem shown in Fig. 3 as an example and suppose only 2 threads are used. The computing domain is first divided into two parts in horizontal direction, and each thread takes one part. In each part, particles are further divided into two groups, L group and R group, as denoted by solid dots and hollow dots in Fig. 3, respectively. To achieve good load balance, the number of particles in each group should be as close as possible.

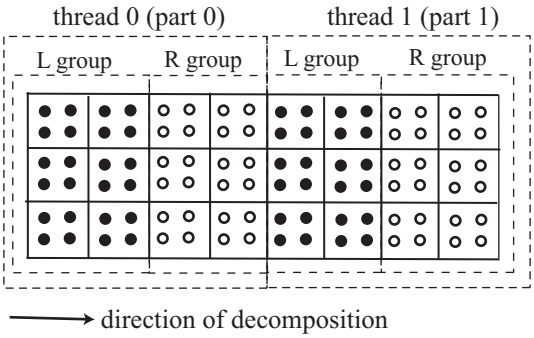

Figure 3: Decomposition of the computing grid. Solid dots denote particles in L group, and hollow dots denote particles in R group

Because the final result of grid updating does not depend on the order that the particles are processed, we can let all threads update their own L group first, and then update their own R group, namely, the L and R groups are updated alternately. An OpenMP barrier is added between L group update and R group update. Before the barrier, all threads update their L groups in parallel. As L groups of part 0 and part 1 do not overlap each other, no data races emerge. After the barrier, all threads update their R groups in parallel. As R groups of part 0 and part1 do not overlap each other, no data races emerge too. In this way, data races are successfully avoided. After this operation, all particles in the whole computing domain are mapped to grid, and the result of parallel MPM is guaranteed to be correct.

To make this process clear, we rewrite this grid updating scheme into pseudo code as the following.

```
#pragma omp parallel num_threads(nthread)
{
   int ithread=omp_get_thread_num();
   for(int i=0;i<nb_point;i++){
      Particle = particle_list[i];
        if(Particle belongs to L group of ithread)
           <map variables on Particle to all grid nodes
            of the cell containing the particle>
   }

   #pragma omp barrier

   for(int i=0;i<nb_point;i++){
      Particle = particle_list[i];
      if(Particle belongs to R group of ithread)
         <map variables on Particle to all grid nodes
          of the cell containing the particle>
   }
}
```

As can be seen from the computing flow given in section 2, MPM updates grid for 3 times. It first updates mass and momentum, then force, finally momentum again in MUSL scheme. Thus the code above will waste too much time, because before and after the barrier, all threads will loop over all particles and check whether this particle belongs to this subdomain. In order to save run time, we create an index of particle denoting the particle ID in each subdomain, and in the loops above, each thread will loop over its own index, greatly reducing the number of iterations. This process can be done in parallel as:

```
int *LIndex, *RIndex;
<allocate memory for LIndex and RIndex>
#pragma omp parallel num_threads(nthread)
{
   int ithread=omp_get_thread_num();
   int m,n;
   for(int i=0;i<nb_point;i++){
      Particle = particle_list[i];
      if(Particle belongs to L group of ithread)
```

```
            LIndex[m++]=i;
          elseif(Particle belongs to R group of ithread)
            RIndex[n++]=i;
      }
  }
```

The advantages of the proposed alternated grid updating method are obvious. The amount of memory usage remains almost equal to the serial version of MPM. The only increase of memory comes from the index of each subdomain. Also, this algorithm is much easier to implement compared to [Huang, Zhang, Ma and Wang (2008)]. We only need to rewrite the loop over particles and make them loop over different indexes. No complex manipulation of local grid is needed.

4.2 Particle updating

This stage includes particle stress update and position update, which mainly loops over all particles, threads will not interfere with each other, so that we can simply add

```
#pragma omp parallel for
```

directive before those loops.

Some special attention should be paid to C++ class data structure. To declare private variables which are inside classes and appear in a parallel region of any function of the same class to be private using "private" or "threadprivate" directive is not supported by OpenMP standard. The only way to solve this problem is to move temporary variables that have been declared in class body into specific functions.

Take the following pseudo code as an example. If *nthread*>=2 , it may happen that two threads are executing `func()` at the same time, resulting in data races. If *a* and *b* are only temporary variables, we should move them into `func()`. Otherwise, this code block is inherently serial and should not be parallelized.

```
class example{
  public:
    void a, b;

    void func(){
    #pragma omp parallel num_threads(nthread)
      <change a and b>
    }
}
```

5 Dynamic load balance

The most important issue to achieve high efficiency for a parallel program is load balance. Amount of computation should be decomposed evenly among all threads. In the stage of updating particle variables, as we directly put

```
#pragma omp parallel for
```

before "for" loop, OpenMP decomposes all iterations among threads, keeping an automatic load balance. Thus load imbalance mainly emerges when updating grid node variables. In MPM, particles move dynamically all over the computing domain, and this may result in significant load imbalance. Grid node updating is realized by looping over particles. Hence, load balance will be achieved if the number of particles in each thread is kept balanced.

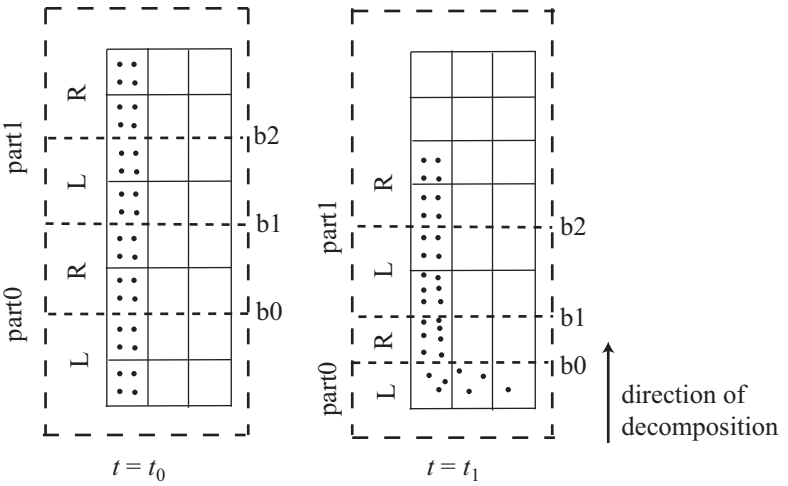

Figure 4: Dynamic domain decomposition of Taylor bar impact

For the algorithm of updating grid node described in section 4.1, dynamic load balance algorithm is easy to implement. We only need to change the boundary of each subdomain and rebuild the index array of the particles in each subdomain such that each subdomain accommodates roughly the same number of particle. Fig. 4 is an example of Taylor bar impact and shows the process of dynamic decomposition, where b1 is the boundary between two threads, b0 and b2 are boundary between L and R parts of each thread repectively.

To achieve a good performance, we rebuild the index array every n time steps, where n is determined experimentally. Fig. 5 plots the bound position along the direction of decomposition in the example of Fig. 4, which shows that particles al-

most stay static near the end of computing and boundaries also do not change. At this time redecomposition is not required at all. To avoid such a waste of unnecessary redecomposition, a criterion denoting the extent of imbalance is evaluated, which is defined as the ratio of the maximum over the average number of particles each thread holds. If it is bigger than a specific value, redecomposition is required.

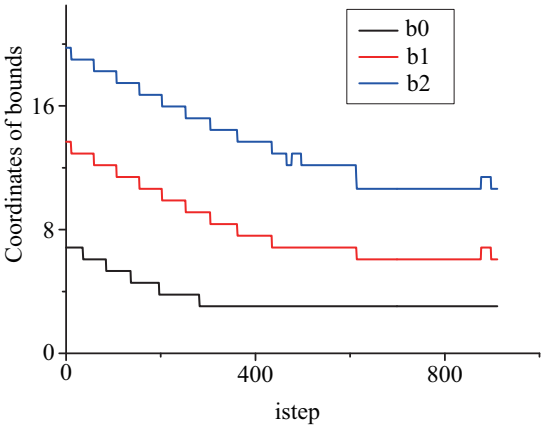

Figure 5: Boundary of each subdomain

In the example Taylor bar impact, the total number of particles is 5239 and 2 threads are used. Hence, the average number of particles in each subdomain is about 1300. Fig. 6 and Fig. 7 show the difference in the number of particles between two threads, where the left figure shows the difference between L parts of the 2 threads, while the right figure shows the difference between R parts of the two threads. We can see that when balance algorithm is used, the difference was restricted to a relatively low level. Without balance algorithm, the workload quickly becomes extremely imbalanced. The workload balance also deteriorates a little near the end of simulation, as shown in the right figure of Fig. 6. At this time, the particles are distributed very unevenly.

Decomposition along one dimension has many advantages. Usually, the number of threads equals the number of subdomains, so that each thread is assigned a subdomain. When the decomposition is realized by cutting the computing domain along two or more dimensions, the subdomains are usually regular blocks. Also take the Taylor bar impact as an example, rigorous load balance can be achieved over the whole process of simulation if the whole bar is decomposed into 2 or 4 parts, as shown in Fig. 8. However, it is hard to decompose the computing domain into 6 blocks and keep good load balance at the same time if 6 threads is used. It is also very hard to decompose a 1/4 Taylor bar to achieve rigorous load balance, no matter

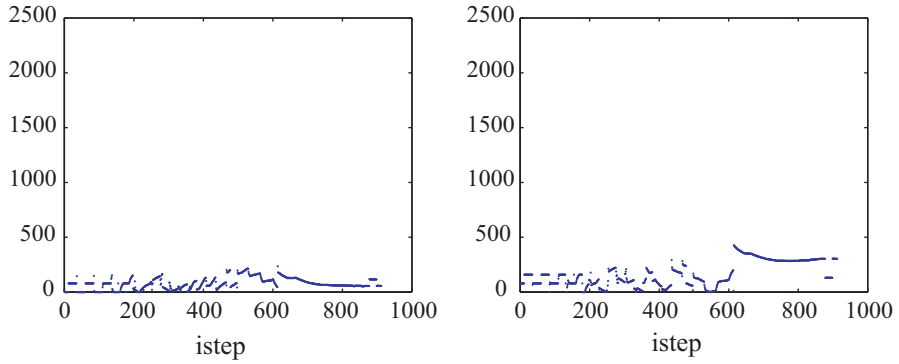

Figure 6: Difference in the number of particles between threads when dynamic domain decomposition is used

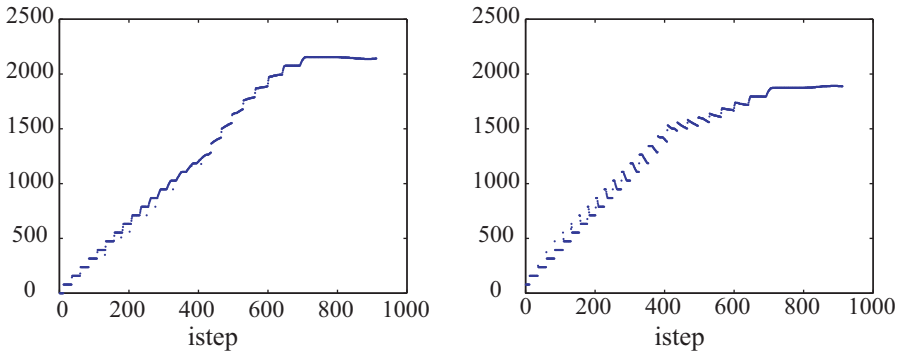

Figure 7: Difference in the number of particles between threads without using dynamic domain decomposition

how many threads is used. In this case, applications of domain decomposition over more than one dimensions are very limited.

One exception is when task pool strategy is used. In this case, the domain can be cut into a large number of subdomains, each as a task. Load balance is achieved by scheduling tasks. But this strategy is seldom used in OpenMP programming. As a result, our method of decomposition along one dimension can be widely used. The load balance is also very good, which can be seen in the section of numerical tests.

Besides the measures described above, we adopted other well-known strategies to enhance the performance. For example, we enlarge the parallel region as much as possible, parallelized almost all loops to reduce the effect of Amdahl's law and changed the order of loop variable to make full use of cache. Unnecessary implicit

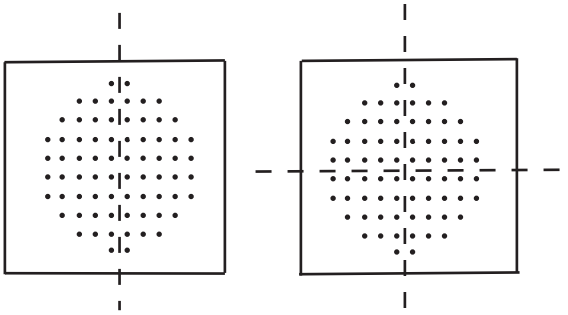

Figure 8: Possible decomposition method for a full Taylor bar problem (top view), dash line denotes the boundary between threads

barriers are also removed by directive `#pragma omp nowait`. All these strategies can be seen in [Chapman, Jost and Pas (2007)].

6 Numerical test

All the following examples are tested on a shared memory machine with two Quad-Core Xeon 5520 CPUs and 12GB memory. The operating system is CentOS and gcc 4.2.1 compiler is used.

6.1 Taylor bar impact

Taylor bar test, see Fig. 4, has been very popular in testing parameters of materials at high strain rate. The bar is made of copper, and the Johnson Cook constitutive model

$$\sigma_y = (A + B\varepsilon^{p^n})(1 + C\ln\dot{\varepsilon}^*)(1 - T^{*m}) \tag{12}$$

and Mie-Grüneisen equation of states (EOS) are employed to update the stress and pressure of particles. In Eq.(12), A, B, n and m are the material constants, ε^p is the effective plastic strain, $\dot{\varepsilon}^*$ denotes the dimensionless effective plastic strain rate and defined as $\dot{\varepsilon}^* = \dot{\varepsilon}^p/\dot{\varepsilon}_0$ for $\dot{\varepsilon}_0 = 1\text{s}^{-1}$, and T^* is the homologous temperature, which is defined as $T^* = (T - T_{\text{room}})/(T_{\text{melt}} - T_{\text{room}})$ with T_{melt} the melt temperature and T_{room} the room temperature. The material parameters are listed in Tab. 1 and Tab. 2. The background grid covers the space domain of $(-11.4 : 11.4, -11.4 : 11.4, 0 : 26.6)$ mm. To test the dependency of parallel efficiency on problem scale, three cases with different cell sizes are investigated, whose cell sizes are equal to 0.76, 0.38 and 0.19, respectively. There are 34596 grid nodes and 21172 particles in case 1, 264191 grid nodes and 169376 particles in case 2, 2064381 grid nodes and

Table 1: Parameters of strength model of copper

ρ_0(g/mm^3)	E(MPa)	ν	A(MPa)	B(MPa)	N
8.9×10^{-3}	117.0×10^{3}	0.3	98	368	0.7
C	M	T_{room}	T_{melt}	c_v	$\dot{\varepsilon}_0$
0.025	1.09	293K	1900K	385	1.0×10^{-3}
D_1	D_2	D_3	D_4	D_5	
0.5	4.5	-3.03	0.014	1.12	

Table 2: Parameters of EOS

c_0 (mm/ms)	s	γ_0
3940	1.49	1.96

1346432 particles in case 3, respectively. We decompose computing domains along the length of the Taylor bar. The overall efficiency, grid node updating efficiency and particle updating efficiency for these three cases are shown in Fig. 9, Fig. 10 and Fig. 11, respectively. All the results were obtained by using release version of MPM3D with -O3 optimization.

When the number of particles increases and the cell size decreases, one thread can hold more layers of grid and thus better load balance can be achieved in the stage of grid node updating. Therefore, the parallel efficiency will usually increase with increasing problem scale.

In contrast, when the number of threads increases, one thread will hold less layers of grid, so that the parallel efficiency of grid updating decreases. We decompose particles according to their layer ID into $2*nthread$ subdomains. Therefore, the "granular" of this algorithm is much larger, compared with loop–level parallelism. With increasing number of threads, the number of layers of grid each thread can accommodate decreases. It may happen that one layer of grid owns a large number of particles, while others owns only a small number of particles, which easily results in load imbalance. However, it can be found from Fig. 9 that a good efficiency of up to 90% can be achieved when using 4 threads. This disadvantage of alternated grid updating method can be compensated by using hybrid mode of parallelism, each process with no more than 4 threads.

Although the efficiency of particle updating does not show steep decrease compared with the efficiency of grid node updating, it does decrease. This phenomenon is more obvious in the example of shaped charge given later.

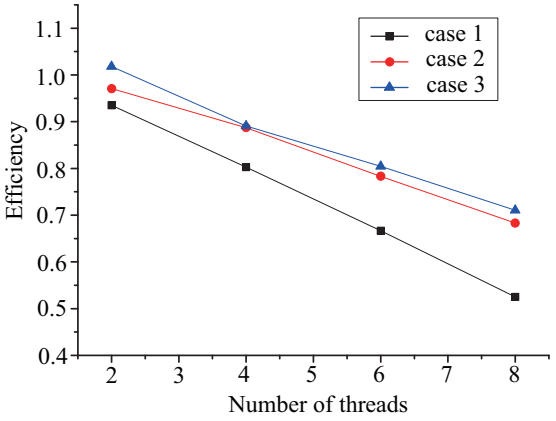

Figure 9: Overall efficiency of Taylor bar impact

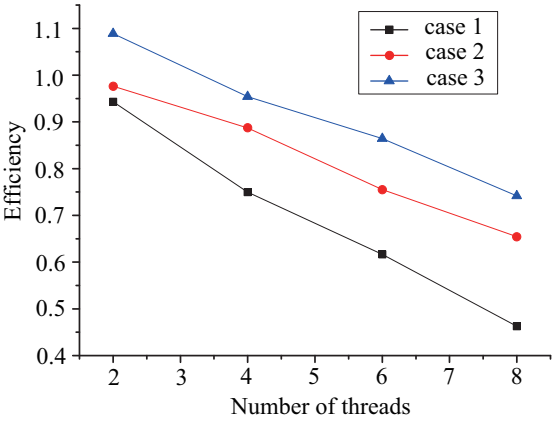

Figure 10: Efficiency of grid node updating in Taylor bar impact

6.2 2D TNT gas explosion

The initial discretization configuration of a 2D TNT is shown in Fig. 12. Jones–Wilkins–Lee (JWL) EOS is employed to update the pressure of TNT, whose parameters are listed in Tab. 3.

In this simulation, adaptive particle splitting algorithm is used to prevent numerical fracture, so that the number of particles is increased from 5027 to 94866. This scheme will add particles to the end of particle list, which will manipulate memory directly. In OpenMP standard, such manipulations are inherently serial and cannot be parallelized. This might decrease the efficiency slightly.

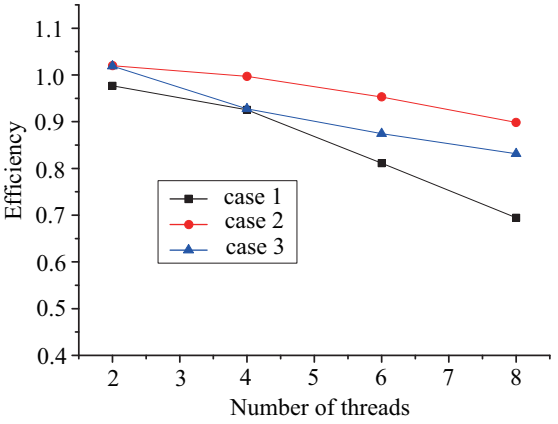

Figure 11: Efficiency of particle updating in Taylor bar impact

Due to the symmetry, a 1/4 model is used, and the distribution of particles are very uneven, see Fig. 12. If we cut the computing domain along two dimensions, and assign one subdomain to each thread, it can be predicted that the load balance will be very poor. In this simulation, the grid is decomposed along one dimension and the dynamic decomposition is applied. The efficiency is plotted in Fig. 13, which shows that the efficiency in this simulation is almost as good as that in Taylor bar test.

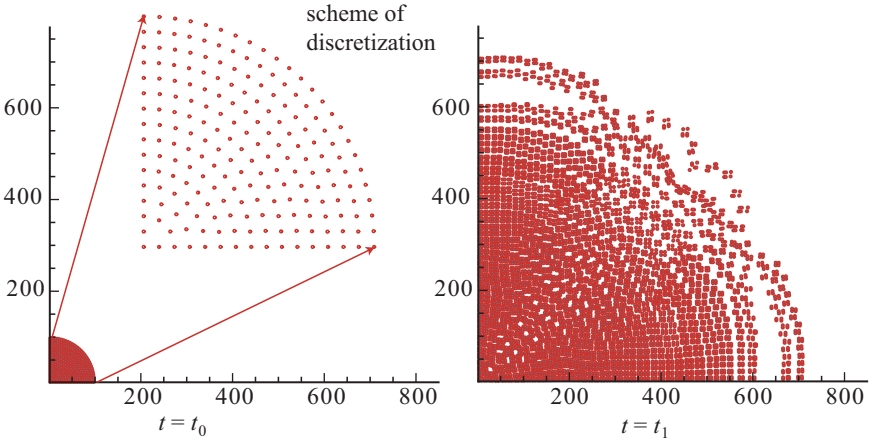

Figure 12: Configuration of 2D TNT gas explosion

Table 3: Parameters of TNT

ρ(g/mm^3)	Detonation velocity (mm/ms)	Shear viscosity		
1.63×10^{-3}	6930	1.0×10^4		
A (MPa)	B (MPa)	R_1	R_2	ω
3.7377×10^5	3.7471×10^3	4.15	0.90	0.35

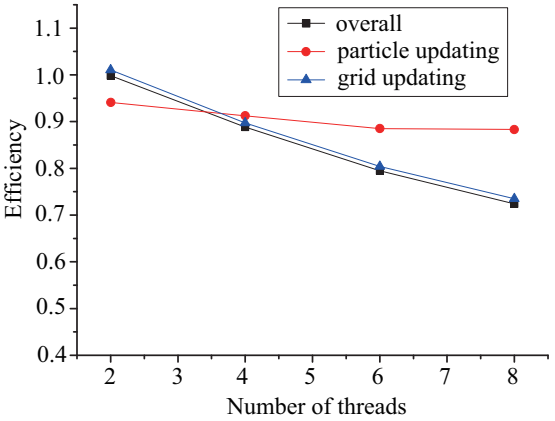

Figure 13: Efficiency of 2D TNT gas explosion

6.3 Shaped charge

A shaped charge is an explosive charge shaped to focus the effect of the explosive's energy, which can generate jet with high velocity and high energy density. It can be used to penetrate the armor of tanks, and in the oil and gas industry.

A 3D shaped charge with apex angle of $\alpha = 38^\circ$ is simulated. The outer shell and the liner consist of copper, which are simulated using Johnson-Cook strength model and Grüneisen EOS. The dynamite is TNT whose pressure is updated using JWL equation of state.

To prevent numerical fracture, the adaptive particle splitting scheme is also used for the liner, so that the number of particles of the liner is increased from 66912 at $0\mu s$ to 112996 at $20\mu s$. The total number of particles is 1187884 at the beginning. The profile of the liner at $20\mu s$ can be seen in Fig. 16 and the efficiency is plotted in Fig. 15, which shows the performance in this case is a little worse than the previous two tests. The efficiency of particle updating also reduces to about 74% when 8 threads is used, which is much lower than previous cases. This can be attributed to possible cache miss. The particles in this test move too fast. In the end of simulation, the particles are distributed all around the computing domain.

The position of particles in the particle index does not match their spacial position. As variables of the cell where the particle is located have to be fetched from main memory to complete particle updating, this irregular dynamic behavior of particles is not good for cache reuse. To overcome this deficiency, data structures should be adapted to cell based. But this will need reasonable work, and will be considered later.

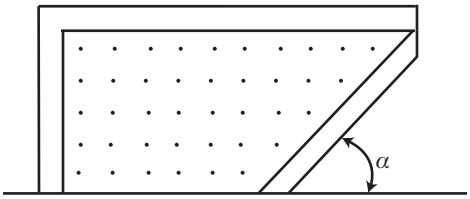

Figure 14: Initial configuration of shaped charge

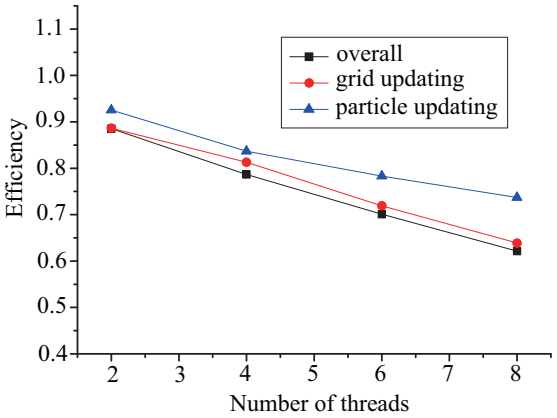

Figure 15: Efficiency of shaped charge

7 Discussion and conclusion

In this paper, a new method for updating grid node in parallel using OpenMP is proposed, and a dynamic decomposition scheme is also successfully implemented. The test results show that the performance of the proposed method is good, and the speedup can be up to 5.8 when using 8 threads.

Not like decomposition method along more than one dimensions, this dynamic decomposition algorithm can be applied to many situations. It also performs very well on load balance.

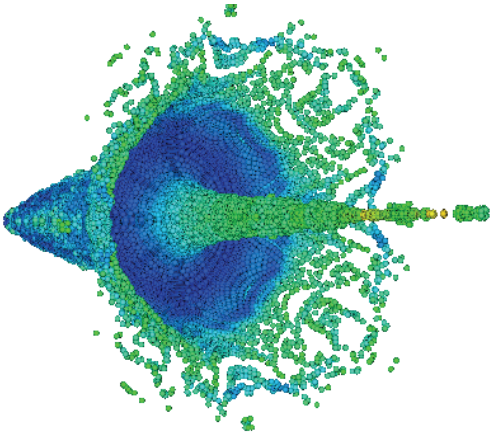

Figure 16: Profile of the liner at $t = 20\mu s$

If the problem scale is fixed, then the number of layers of grid along the direction of decomposition is fixed. When the number of threads increases, the number of layers of grid each thread can hold decreases and load balance also deteriorates. We should make sure each thread can hold a number of layers of grid, so that the number of particles in each subdomain adds up to roughly the average number.

The stage of particle updating is rigorously well balanced. But as can be seen from Fig. 15, the efficiency still decreases slightly as the number of threads increases. This might come from cache miss. Therefore, further optimization of the original serial program is needed.

Acknowledgement: This work was supported by the National Basic Research Program of China (2010CB832701) and National Natural Science Foundation of China (10872107).

References

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

Bardenhagen, S. G.; Guilkey, J. E.; Roessig, K. M. (2001): An improved contact algorithm for the material point method and application to stress propagation in granular material. *CMES: Computer Modeling in Engineering and Science*, 2(4), 509–522

Bardenhagen, S. G.; Kober E. M. (2004): The generalized interpolation material point method. *CMES: Computer Modeling in Engineering and Science*, 5(6), 477–495

Chapman, B.; Jost, G.; Ruud van der Pas (2007): *Using OpenMP: portable shared memory parallel programming*. The MPI Press

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

Dagum, L.; Menon, R. (1998): OpenMP: An Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*

Danielson, K. T.; Hao, S.; Liu, W. K.; Aziz Uras, R.; Li, S. F (2000): Parallel computation of meshless methods for explicit dynamic analysis. *Int. J. Numer. Meth. Engng*, 47, 1323–1341

Daphalapurkar, N. P.; Lu, H.; Coker, D.; Komanduri, R.(2007): Simulation of dynamic crack growth using the generalized interpolation material point (gimp) method. *International Journal of Fracture*, vol. 143, no. 1, pp. 79–102.

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*, 57(9), 1323–1338

Guo, Y. J.; Nairn, J. A. (2006): Three-dimensional dynamic fracture analysis using the material point method. *CMES: Computer Modeling in Engineering & Sciences*, vol. 16, no. 3, pp. 141–155.

Harlow, F. H. (1964): The particle-in-cell computing method for fluid dynamics. *Methods for Computational Physics*, vol. 3, pp. 319–343

Hoeflinger, J.; Alavilli, P.; Jakson, T.; Kuhn, B.(2001): Producing scalable performance with OpenMP: Experiments with two CFD applications. *Parallel computing*, 27, 391–413

Hu, W. Q.; 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, no. 12, pp. 2066–2096.

Huang, P.; Zhang, X.; Ma, S.; Wang, H.K. (2008): Shared Memory OpenMP Parallelization of Explicit MPM and Its Application to Hypervelocity Impact. *CMES: Computer Modeling in Engineering & Science*, vol. 38, no. 2, pp. 119–148

Jin, H.;Frumkin, M.; Yan, J. (1999): The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *NAS Technical Report NAS-99-011*

Liu, Y.; Zhang, X.; Sze, K. Y.; Wang, M. (2007): Smoothed molecular dynamics for large step time integration. *CMES-Computer Modeling in Engineering & Sciences*, vol. 20, no. 3, pp. 177–191

Ma, J.; Liu, Y.; Lu, H. B.; Komanduri, R. (2006): Multiscale simulation of nanoindentation using the generalized interpolation material point (GIMP) method, dislocation dynamics (dd) and molecular dynamics (md). *CMES: Computer Modeling in Engineering & Sciences*, vol. 16, no. 1, pp. 41–55

Ma, Z. T.; Zhang, X.; Huang, P. (2010): An object-oriented MPM framework for simulation of large deformation and contact of numerous grains. *CMES: Computer Modeling in Engineering & Sciences*, 55(1), 61–88

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

OpenMP Architecture Review Board (2008): OpenMP Application Program Interface. *http://www.openmp.org*

Othmer, C.; Schüle, J. (2002): Dynamic load balancing of plasma particle-in-cell simulations: The taskfarm alternative. *Computer Physics Communications*, 147, 741–744

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 comonent-based parallel infrastructure for the simulation of fluid-structure interaction. *Engineering with Computers*, vol. 22, pp. 277–292

Plimpton, S. J.; Seidel, D. B.; Pasik, M. F.; Coats, R. S.; Montry, G. R. (2003): A load-balancing algorithm for a parallel electromagnetic particle-in-cell code. *Computer Physics Communications*, 152, 227–241

Stantchev, G.; Dorland, W.; Gumerovb, N. (2008): Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. *J. Parallel Distrib. Comput*, 68, 1339–1349

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

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 and Engineering*, 139(1–4), 409–429

Sulsky, D.; Zhou, S. J.; Shreyer, H. L. (1995): Application of a Particle-in-cell Method to Solid Mechanics. *Computer Physics Communications*, 87(1–2), 236–252

York, A. R; Sulsky, D.; Schreyer, H. L. (1999): The material point method for simulation of thin membranes. *International Journal for Numerical Methods in Engineering*, 44(10), 1429–1456

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