International Journal of Rock Mechanics & Mining Sciences 133 (2020) 104361

Contents lists available at ScienceDirect

International Journal of Rock Mechanics and Mining Sciences

journal homepage: http://www.elsevier.com/locate/ijrmms

Parallel implementation of the four-dimensional lattice spring model on heterogeneous CPU-GPU systems

Gao-Feng Zhao [*], Fuxin Rui, Hua Chen, Qin Li [**]

State Key Laboratory of Hydraulic Engineering Simulation and Safety, School of Civil Engineering, Tianjin University, Tianjin, 300072, China

A R T I C L E I N F O

Keywords:
Lattice spring model
Parallelization
Heterogeneous computing
Fracturing

A B S T R A C T

As a newly developed computational method, the four-dimensional lattice spring model (4D-LSM) is computationally intensive due to the introduction of extra-dimensional interactions. In this work, the 4D-LSM is parallelized to fully utilize the available computational resources of modern computers, namely, the multi-core CPU and the GPU. To utilize computing power of the multi-core CPU, OpenMP with a fork-join scheme is used to assign computational tasks to different CPU threads, whereas CUDA, with a granular computing scheme, is adopted to assign computations to thousands of GPU threads. A domain decomposition with a data communication scheme is proposed to utilize both the multi-core CPU and the GPU. The influence of digital precision and hardware on the parallel computing performance of the 4D-LSM are investigated through a number of numerical examples including elastic deformation, elastic bulking and dynamic fracturing. Finally, the multi-core CPU 4D-LSM is used to solve a crack propagation problem and is compared with existing experimental and numerical results.

1. Introduction

The four-dimensional lattice spring model (4D-LSM)[1] was developed to solve the Poisson's ratio limitation of the classical lattice spring model[2] by introducing fourth dimensional interactions. Its main advantage is the feasibility of dealing with nonlinear dynamic responses of solids including fracturing. In the 4D-LSM, fracturing of the solid is represented as a sequence of spring bond breakages between particles. Conceptually, it is similar to the treatment adopted in the damage-based model,[3] the phase field model,[4] and the cohesive zone model (CZM).[5] For CZM-based numerical approaches,[6] the base numerical method, e.g., the finite element method (FEM),[6] the numerical manifold method (NMM),[7] the scaled boundary finite element method (SBFEM),[8] is used to address mechanical responses of the bulk material, whereas the fracturing is handled by CZM elements. However, "artificial compliance" spurious elastic behaviour due to the insertion of CZM interfacial elements might arise for both the "extrinsic"- and "intrinsic"-based CZMs.[9] The LSM can be viewed as a computational method that is made from the purely intrinsic-based CZM-like interactions, where the bulk response is also represented by CZM-like interactions. Because there is no introduction of zero thickness interface elements, the LSM is free from the "artificial compliance" problem. However, to have a realistic simulation of solid fracturing with the LSM usually requires a large number of numerical elements, which is computationally demanding. Parallelization is the only feasible solution.

With the popularization and maturity of modern computer technology, such as the 64-bit operating system, the multi-core central processing unit (CPU) and the graphics processing unit (GPU), it is possible to conduct large-scale scientific computations with personal computers. Some researchers even believe that *a new era* is coming for scientific computing.[10] Due to technical or economic constraints, the computing performance (the number of instructions processed per second) of a single processor is currently close to the limit. A multi-core CPU is a straightforward solution to increase the performance from integrating more cores in a single CPU chip. A modern multi-core CPU has 2 to 28 cores,[11] which could run up to 56 CPU threads using the hyper-threading technique. A GPU was specially designed for image processing. Due to its highly parallel structure, the GPU has an advantage on the processing of massive simple mathematical operations simultaneously, which was recently used in scientific computing. The basic concept of the GPU is close to the multi-core CPU, however, it integrated much less sophisticated processing units, e.g., there are 1280 CUDA cores on the GeForce

* Corresponding author.
** Corresponding author.
E-mail addresses: gaofeng.zhao@tju.edu.cn (G.-F. Zhao), lqin223@126.com (Q. Li).

https://doi.org/10.1016/j.ijrmms.2020.104361
Received 11 April 2020; Received in revised form 30 April 2020; Accepted 30 April 2020
Available online 30 June 2020

GTX1060 card.[12] In a modern computer, in most cases, it will be equipped with both a multi-core CPU and a GPU. However, to obtain the performance gain of the multi-core CPU and the GPU for a specific numerical method, parallel implementation of the numerical method is required. Currently, parallelization techniques such as Open Multi-Processing (OpenMP)[13] and Compute Unified Device Architecture (CUDA)[12] are available to parallelize a serial code. Many numerical methods have been parallelized with these techniques, e.g., the boundary element method (BEM),[14] the FEM,[15] the material point method (MPM),[16] the discrete element model (DEM),[17–19] the smoothed particle hydrodynamics (SPH) method,[20] discontinuous deformation analysis (DDA),[21] and the distinct lattice spring model (DLSM),[22] finite-discrete element method.[23,24] Typical GPU speedups were reported from 10 × to 100 × . However, there are some drawbacks in these studies. First, they were more focused on the speed up of the parallel computing, but lacked a deep investigation on the precision in solving various mechanical problems under the CPU-GPU heterogeneous computing environment. Second, most of these works did not make full use of both the multi-core CPU and GPU. The most computationally intensive part is handled by either the multi-core CPU or GPU alone, rather than being shared evenly between them.

In this work, we parallelize the 4D-LSM to fully utilize the multi-core CPU and GPU equipped in modern computers. First, a brief introduction to the concept and calculation procedures involved in 4D-LSM is presented. Then, the parallel implementation of 4D-LSM for the multi-core CPU and GPU environments is presented, which emphasizes parallel strategies on the multi-core CPU, GPU, and the CPU-GPU three schemes. Following this, the influences of digital precision and computing hardware (the multi-core CPU and GPU) are investigated through performing a number of numerical examples covering a wide range of mechanical problems. The parallel performance of multi-core CPU 4D-LSM and GPU 4D-LSM is tested using computers with different configurations.

2. Parallelization of 4D-LSM

2.1. 4D-LSM

In 4D-LSM, the 3D world is assumed to be a membrane in four dimensions, which is further presented through a group of discrete 4D particles through springs. Details of description and proof of 4D-LSM can be found in Ref. 1. Here, we will only focus on equations, which are essential for the parallel implementation. First, for a 4D particle, its spatial position and motion are represented as:

$$\mathbf{x}_i = (x_i y_i z_i \vartheta_i)^T \tag{1}$$

$$\dot{\mathbf{x}}_i = (\dot{x}_i \dot{y}_i \dot{z}_i \dot{\vartheta}_i)^T \tag{2}$$

$$\ddot{\mathbf{x}}_i = (\ddot{x}_i \ddot{y}_i \ddot{z}_i \ddot{\vartheta}_i)^T \tag{3}$$

where i refers to the ith particle; x, y, z and ϑ are the corresponding four coordinates of the particle; $\dot{x}$, $\dot{y}$, $\dot{z}$ and $\dot{\vartheta}$ are the particle velocity in four directions; and $\ddot{x}$, $\ddot{y}$, $\ddot{z}$ and $\ddot{\vartheta}$ are the particle acceleration in four directions. Using the time central differential integration, the position of the 4D particle is updated as:

$$\mathbf{x}_i^{t+\Delta t} = \mathbf{x}_i^t + \dot{\mathbf{x}}_i^{t+\frac{\Delta t}{2}} \Delta t \tag{4}$$

in which t is the time and Δt is the time increment. The particle's velocity can be obtained in a similar way as:

$$\dot{\mathbf{x}}_i^{t+\frac{\Delta t}{2}} = \dot{\mathbf{x}}_i^{t-\frac{\Delta t}{2}} + \ddot{\mathbf{x}}_i^t \Delta t \tag{5}$$

The 4D distance between particle i and j is calculated as:

$$l_{ij}^t = \sqrt{\left(x_j^t - x_i^t\right)^2 + \left(y_j^t - y_i^t\right)^2 + \left(z_j^t - z_i^t\right)^2 + \left(\vartheta_j^t - \vartheta_i^t\right)^2} \tag{6}$$

If these particles are linked through a spring with stiffness of k, then the spring force induced from particle j to particle i can be given as:

$$\mathbf{f}_{ij} = \left(f_{ij}^x, f_{ij}^y, f_{ij}^z, f_{ij}^\vartheta\right)^T = \frac{\mathbf{x}_j^t - \mathbf{x}_j^t}{l_{ij}^t} k \left(l_{ij}^0 - l_{ij}^t\right) \tag{7}$$

where l_{ij}^0 and l_{ij}^t are the initial and current spring length calculated from Equation (6). In 4D-LSM, the body force along the fourth dimension is assumed to be zero, and the current particle force of a given particle with m neighbour particles are calculated as:

$$\mathbf{f}_i = \sum_{j=1}^{m} \mathbf{f}_{ij}^z + \mathbf{g} m_i \tag{8}$$

in which m_i is the particle mass, and $\mathbf{g} = (g_x, g_y, g_z, 0)^T$ (g_x, g_y, and g_z are the gravity accelerations). In 4D-LSM, the Newton's second law is assumed to also be applicable in the fourth dimension, and the particle accelerations in four directions are given as:

$$\ddot{\mathbf{x}}_i^t = \frac{\mathbf{f}_i^t}{m_i} \tag{9}$$

Equations (1)–(9) are all of the essential calculations involved in the parallel implementation of 4D-LSM, which are all natural extensions of the classical 3D LSM[25] to the four-dimensional space.

In 4D-LSM,[1] a parallel world concept is used to construct the 4D lattice model through its 3D counterpart (see Fig. 1). The parallel copy of the 3D model is formed from an offset operation along the fourth dimension. The original 3D lattice model and its parallel version make up a 4D membrane. Then, the original particles and parallel particles are connected through 4D springs. For the cubic lattice, there are three types of 4D springs (see Fig. 1); their stiffness variables are k_α, k_β, and k_γ. To represent the isotropic elasticity, their stiffness values must satisfy the following equation[1]:

$$k_\alpha = k_\beta = 4/3 k_\gamma = \lambda^{4D} k \tag{10}$$

where λ^{4D} is the 4D stiffness ratio and k is the stiffness of the 3D spring, which can be further given as:

$$k = \frac{6VE}{\eta \sum l_i^2} \tag{11}$$

in which V is the represented volume of the 3D lattice model, E is the elastic modulus, l_i is the initial spring length of the 3D lattice model, and η is a scale parameter, which can be further obtained as[1]:

$$\eta = -0.0078506\lambda_{4D}^2 + 0.41613615\lambda_{4D} + 1.00369223 \tag{12}$$

The 4D stiffness ratio can be further obtained from the Poisson's ratio as:

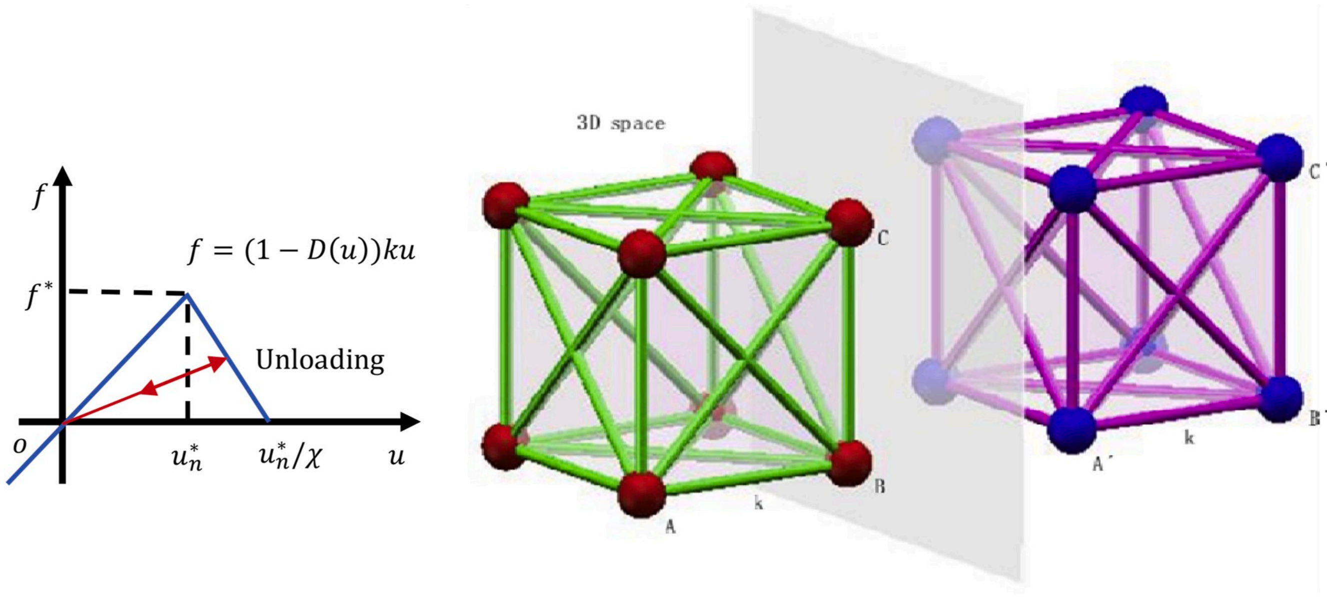
$$\lambda_{4D} = -211.13493779v^3 + 162.84655851v^2 - 55.42449719v + 6.92902211 \tag{13}$$

where v is the Poisson's ratio. With Equations (10)–(13), the corresponding mechanical parameters (spring stiffness) can be determined for Equation (7). These equations are all pre-calculated and will not be involved in the calculation cycle of 4D-LSM. Therefore, there is no need for parallelization. More details and mathematical proof of these equations and the meaning of 4D interaction can be found in the original work of 4D-LSM.[1]

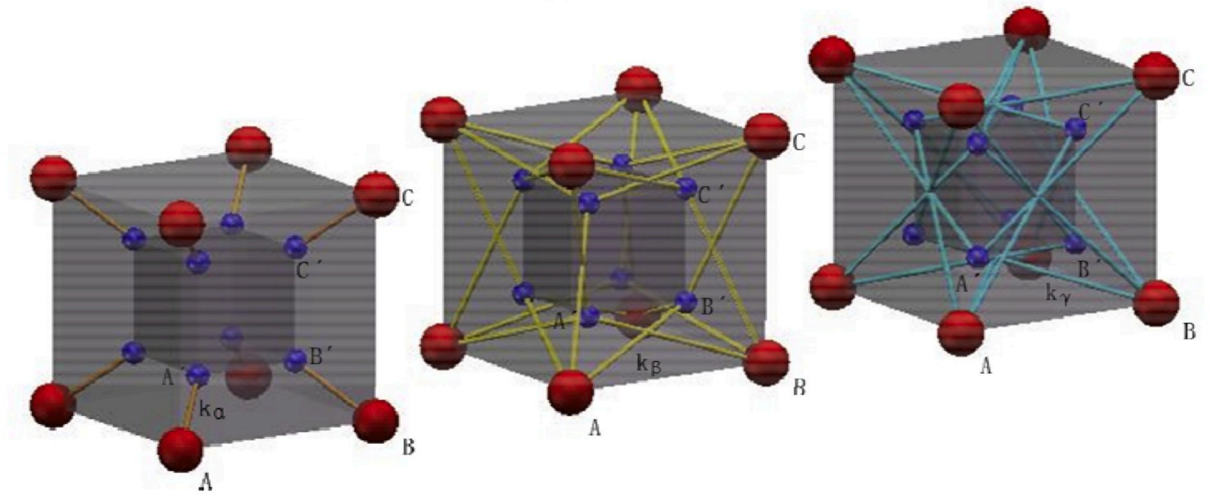
In this work, fracturing of the solid is described by a cohesive zone like model[26] which can be written as

$$f = (1 - D)ku \tag{14}$$

where D repents the amount of damage to the spring. For example, the a linear softening damage function is used for the 3D spring as



(a)



(b)

Fig. 1. Basic concept and components of the 4D-LSM.[1]

$$D(u^{3D}) = \begin{cases} 0 & , \quad u_{3D} < u_{3D}^* \\ 1 - \dfrac{u_{3D}^* - \chi \cdot u^{3D}}{(1-\chi)u_{3D}^*} & , \quad u_{3D}^* \le u_{3D} \le u_{3D}^*/\chi \\ 1 & , \quad u_{3D} > u_{3D}^*/\chi \end{cases} \tag{15}$$

where u_{3D}^* is the peak deformation of the spring bond, and χ is the ratio of the peak deformation to its ultimate deformation (see Fig. 1a). In 4D-LSM, damage of the four types of spring stiffnesses were assumed as

$$D^\alpha \equiv 0, D^\beta = D^\gamma = D^{3D} \tag{16}$$

More details on failure model of 4D-LSM and parameters selection can found in Ref. 26.

2.2. Parallelization strategy

A modern computer, a personal computer or workstation usually contains a multi-core CPU and a GPU. To fully utilize these computing resources, it is essential to understand the data flow of numerical modelling on the computer system. As shown in Fig. 2, for most cases, the computational model is stored in the hard drive (HD). For numerical modelling with multi-core CPU, the data are first read from the HD to the memory (DRAM) then read from DRAM to the cache of the CPU, and finally the data will be processed by the CPU. For multi-core parallelization, each CPU core will handle a small portion of the data. No data communication is required because the CPU cores can access all of the data on the DRAM. For GPU computing, the data needs to be transferred from DRAM to the graphical DRAM of the GPU, which is instructed by the CPU. In GPU computing, the most intensive computing will be handled by the GPU. The GPU threads can also access all of the GPU memory. When we want the CPU and GPU to handle different portions of the computational model, the computer system should be treated as a distributed memory system in this condition. In this case, we have to perform domain decomposition and communication. In this work, we consider multi-core CPU parallelization, GPU parallelization and CPU-GPU parallelization. For the first two cases, we use the shared memory parallel strategy, while in the third case we adopt the distributed memory parallel strategy. Based on these strategies, the corresponding parallel implementation is explained in the following section.

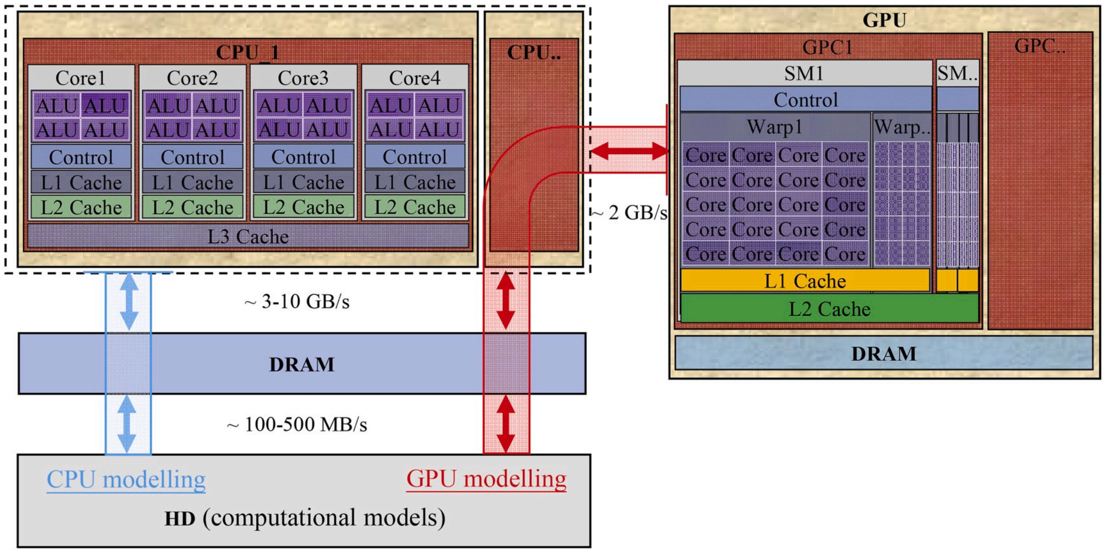


Fig. 2. Typical data flow of numerical modelling on a modern computer with a multi-core CPU and GPU.

2.3. *Serial implementation*

The calculation cycle of the 4D-LSM is shown in Fig. 3. The particle position update refers to Equation (4); the corresponding C pseudocode for a serial CPU 4D-LSM code was given Appendix A including the particle position update, the particle force calculation in Fig. 3, the particle acceleration calculation, and the motion update step. These pieces of pseudocode are the most computational intensive part of 4D-LSM.

2.4. *Multi-core CPU parallel implementation*

In this work, we used OpenMP to parallelize these pieces of code for a multi-core CPU. Only a few lines are required to be added to the serial

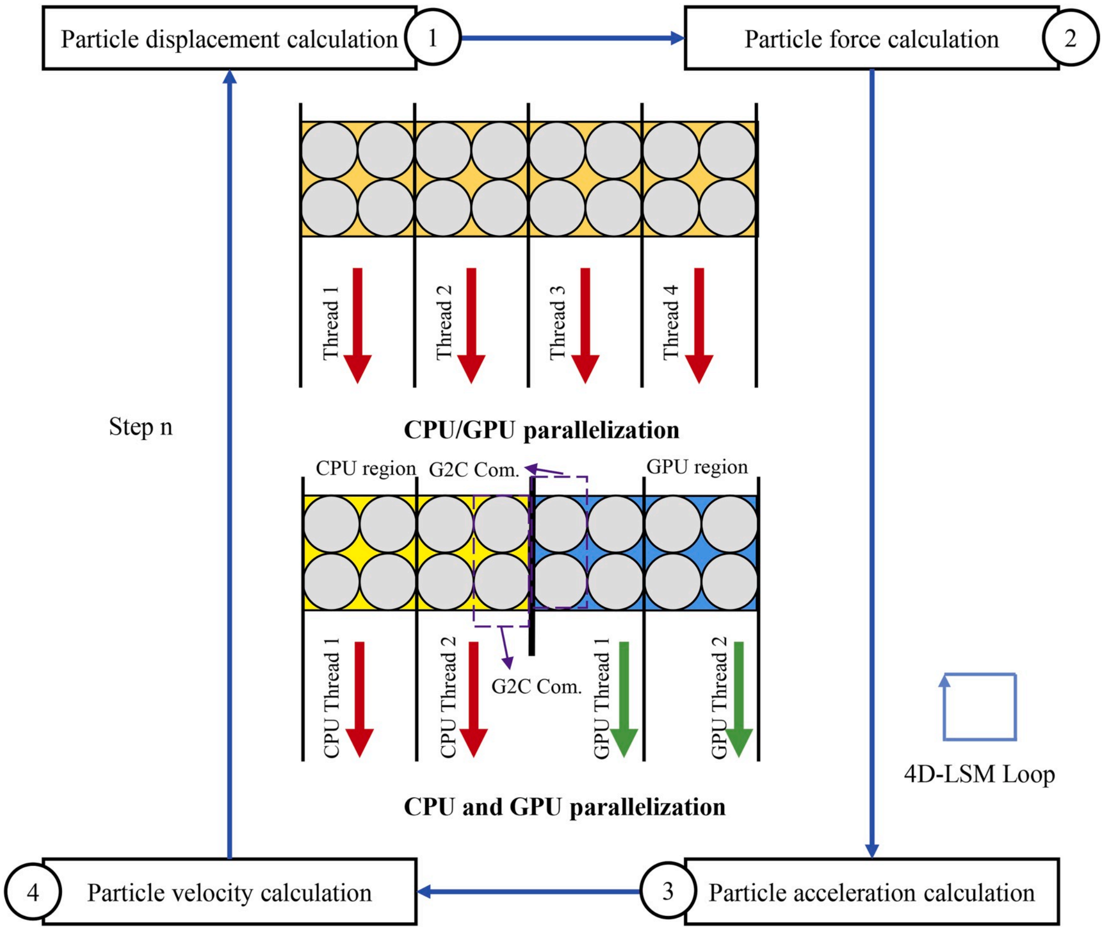


Fig. 3. Calculation cycle of the 4D-LSM and parallelization strategies for a modern computer with a multi-core CPU and GPU.

version. The multi-core CPU parallel pseudocodes of particle position update, the particle force calculation, the particle acceleration calculation and the motion update step were listed in Appendix B.

2.5. GPU parallel implementation

The GPU implementation is more complex. There are two main parts. The first part is the communication between CPU and GPU, which can be completed by using the CUDA function *cudaMemcpy*. The data communication is only conducted at the beginning and the end. The entire calculation cycle is performed within the GPU. In CUDA, the function run in the GPU is called the kernel. Pseudocodes of the main framework and the particle force calculation of GPU 4D-LSM were given in Appendix C. The function (kernel) will be executed by thousands of GPU threads simultaneously. To complete this, we should call the kernel through the following format:

ckParticleForce$<<<$*nGPUBlocks*, *nGPUThreads*$\gg>$ (*NP*)

where *nGPUBlocks* is the number of blocks allocated for the kernel, *nGPUThreads* is the number of threads assigned per block, and *NP* refers to the number of particles assigned to GPU. The total number of threads executing the kernel are *nGPUBlocks* $\times$ *nGPUThreads*. For particle position, velocity and acceleration updates, the corresponding GPU kernel can be modified in a similar way. To further optimize the GPU code and shared memory, the fast hardware math library and memory coherence can be adopted.[21] However, because the particle force computation has to access the memories of the corresponding neighbouring particles, the memory operation of the 4D-LSM is complex. Therefore, the memory coherence optimization is not adopted in this work. The GPU 4D-LSM with CUDA could utilize multiple GPU cards for proper hardware configuration.[23]

2.6. CPU-GPU parallel implementation

When both the CPU and GPU are used for the calculation cycle, the total number of particles are divided into to two parts: $[1\cdots N_{gpu}]$ and $[N_{gpu}+1\cdots N]$. Here, the domain decomposition is based on the index of the particles, and the entire computational model will be divided into two parts. The particle position, velocity and acceleration can be computed without accessing the corresponding neighbour particles. As an example, the corresponding pseudocode for the particle position calculation was given in Appendix D. The particle force update can be worked out by using a similar manner. The only difference is that a communication between the CPU and GPU is required before the execution of the particle force updates. Those particles, belonging to the CPU domain and neighbouring with particles of CPU domain, are put into a *GPU2CPUList*, and before the CPU particle force calculation, the corresponding particle position are transmitted from the GPU to the CPU. A *CPU2GPUList* is also formed for the GPU computing. Before the computing of particle force, the particle position of these particles in the GPU memory will also be updated from the CPU to the GPU.

3. Numerical examples

3.1. Deformation of a table under pressure

The computational differences between the parallel DLSM[25] and the parallel 4D-LSM are studied by a static elastic deformation problem. Fig. 4a shows the 4D-LSM model and the corresponding boundary conditions. The table is made up of 116,000 particles with a diameter of 1 mm. The material parameters are E = 10 GPa, $\nu = 0.2$, $\rho = 2.0\times 10^3$ kg/m^3. Under a surface pressure loading of 1 MPa, the square table was supposed to deform symmetrically. Without loss of generality, the vertical displacement along the table's surface middle line was selected as our target. Fig. 4b shows the displacement curves predicted by the DLSM and 4D-LSM using different computing hardware and digital precisions. These results calculated with the double precision are very close, including multi-core CPU DLSM, GPU DLSM, multi-core CPU 4D-LSM and GPU 4D-LSM. When using the float precision, GPU DLSM can give qualitative results, but the specific deformation is not consistent, while the result of the GPU 4D-LSM is obviously wrong because the curve is almost horizontal, and the specific deformation is totally different from the others. More intuitively, Fig. 5 shows the vertical deformation contour predicted by the GPU DLSM and GPU 4D-LSM with different digital precisions. Fig. 5a, (b) and (d) are predicted by float precision GPU DLSM, double precision GPU DLSM and double precision GPU 4D-LSM, respectively, which can give a qualitative description of the problem — sunken in the middle and four legs upward. For double precision GPU DLSM (Fig. 5b) and double precision GPU 4D-LSM (Fig. 5c), the differences in upward and downward maximum displacements are 3% and 1%, respectively. Although the GPU DLSM with float precision has a similar contour (Fig. 5a), compared with the GPU DLSM with double precision (Fig. 5b), the differences in upward and downward maximum displacements reach 10% and 9%, respectively. However, the contour of float precision GPU 4D-LSM (Fig. 5c) is totally different from the other three cases, where almost all of the desktop has a downward displacement. In summary, for GPU 4D-LSM, double precision is strongly required.

3.2. Elastic buckling of a bar

This example is about the dynamic elastic instability of a bar under vertical loading. The purpose is to analyse the differences of the 4D-LSM in solving nonlinear elastic problems using the CPU and GPU. As shown

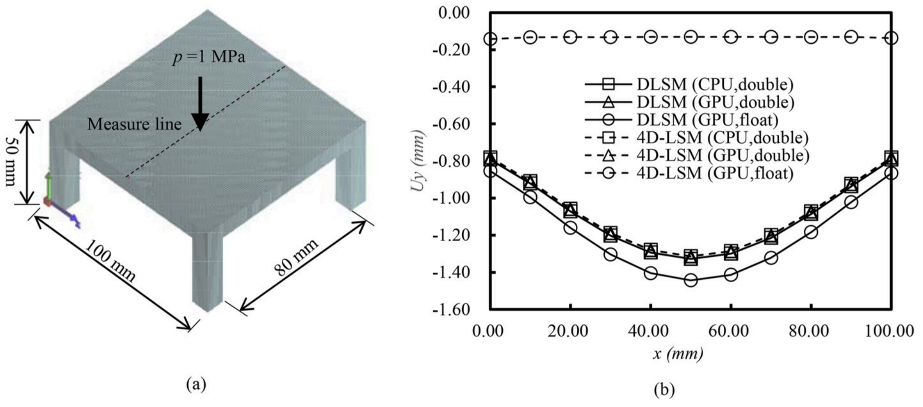


Fig. 4. Elastic deformation of a table under pressure: (a) computational model and (b) deformation along the measure line predicted by the DLSM and 4D-LSM using a multi-core CPU and GPU with different digital precisions.

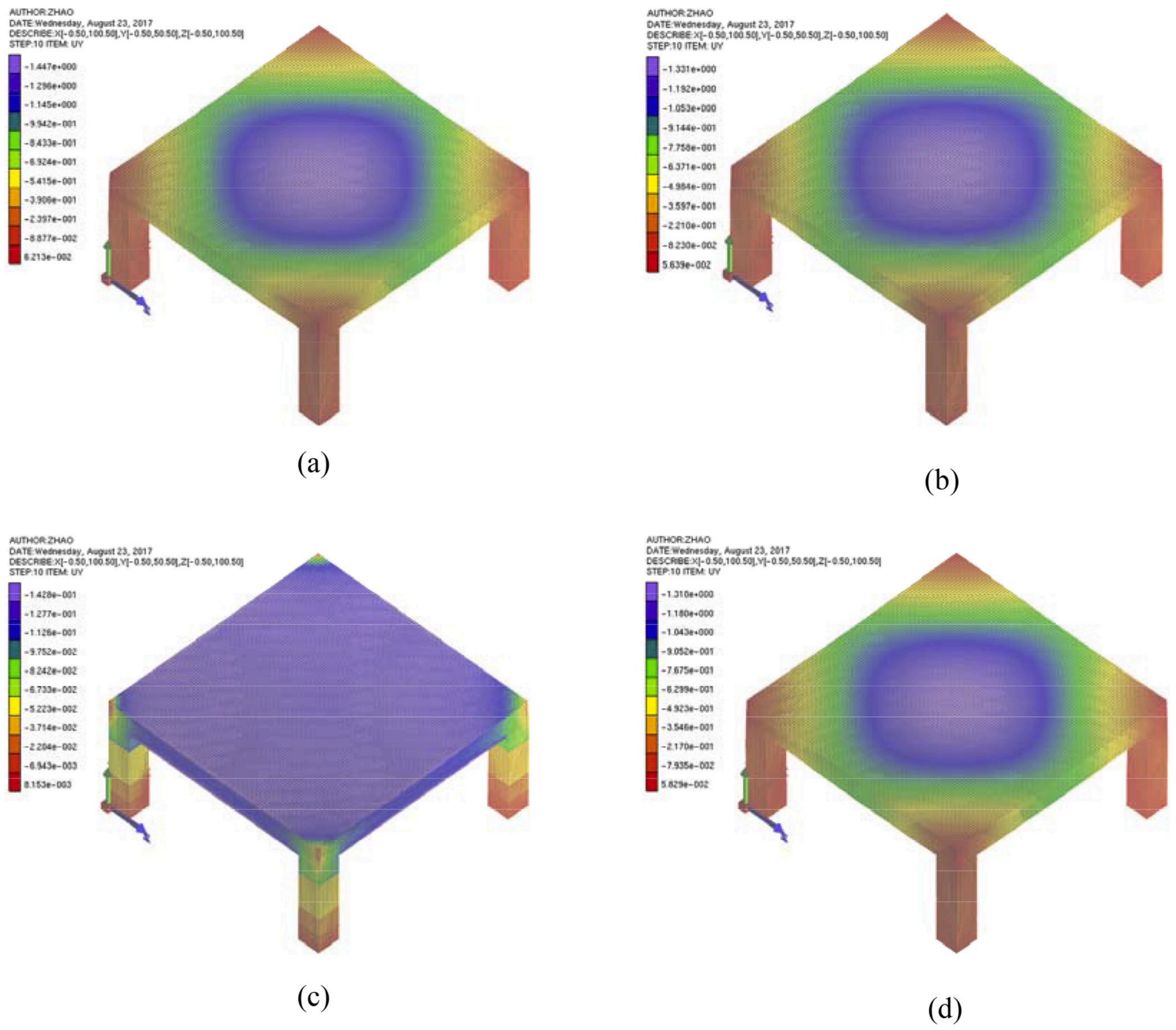


Fig. 5. Deformation contour of the table predicted by the GPU DLSM and GPU 4D-LSM with different digital precisions: (a) the GPU DLSM with float precision, (b) the GPU DLSM with double precision, (c) the GPU 4D-LSM with float precision, and (d) the GPU 4D-LSM with double precision.

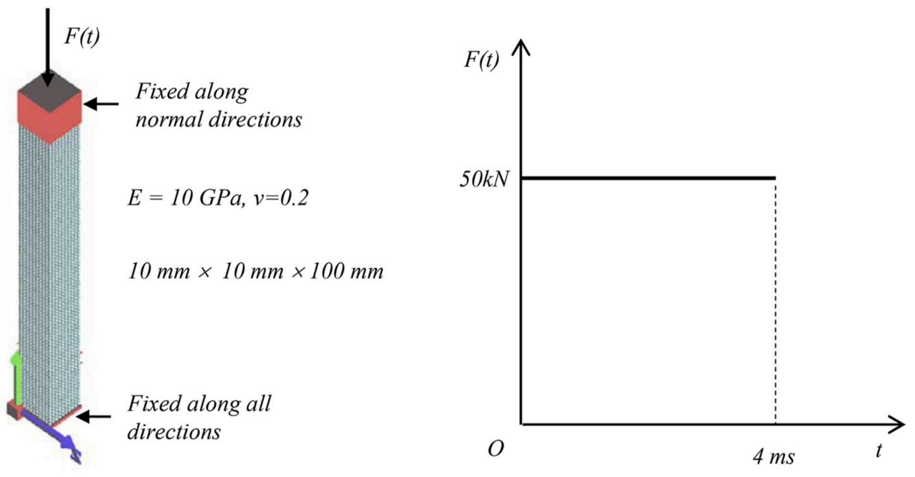


Fig. 6. Computational model and applied loading of an elastic bar buckling problem.

in Fig. 6, the computational model is a 3D bar with dimension of 10 mm × 10 mm × 100 mm. The model consists of 10,000 particles with a diameter of 1 mm. The model boundary condition is that the bottom of the bar is hinged to limit the displacement in all directions. We set the sleeve piece on the top of the pressure bar (as shown in Fig. 6 red area); the height of the sleeve piece is 10 mm to constrain the displacement in the horizontal direction (*XZ* direction) of the top end of the bar. An axial compressive load $F = -50$ kN with the action line coinciding with the *Y* axis is applied to the bar (see Fig. 6). The material properties are $E = 10$ GPa, $\nu = 0.2$ and $\rho = 2.5 \times 10^3$ kg/m^3. As shown in Fig. 7, there is a vertical vibration state during the calculation process. From 1000 steps to 56000 steps, the pressure bars calculated by the CPU and GPU always vibrate within the elastic range with the same amplitude. As shown in Fig. 7, there is no significant difference between the results predicted by the GPU 4D-LSM and the CPU 4D-LSM at 1000 steps and 20000 steps. Here, we refer to the top surface deformation along *y* direction of the bar as *S* (*mm*). When the bar enters the unstable state, the difference of the displacement values predicted by the CPU 4D-LSM and the GPU 4D-LSM increases gradually. For example, at 60000 steps, $S^{(CPU)} = 1.77$ mm, $S^{(GPU)} = 3.21$ mm, and $\Delta S = 1.44$ mm; at 64000 steps, $S^{(CPU)} = 11.78$ mm, $S^{(GPU)} = 13.26$ mm, and $\Delta S = 1.48$ mm. This example reveals that even if we are using double digital precision, for solving highly nonlinear large deformation problems, there will be apparent differences between multi-core CPU 4D-LSM and GPU 4D-LSM, especially at the post failure stage.

3.3. *Surface cracking of a notched sphere*

The dynamic fracturing of solids is a geometric nonlinear problem where the topology of the computational model is dynamically changing during the calculation. In this example, the GPU 4D-LSM and CPU 4D-LSM will be used to solve a 3D surface fracturing problem of a hollow sphere under blasting. Fig. 8 shows the computational model: the diameter of the entire sphere is 100 mm and the thickness of the wall is 30 mm. The dimension of the initial crack is 30 mm × 10 mm × 2 mm. There are 489,828 particles with a diameter of 1 mm. The material properties are $E = 35$ GPa, $\nu = 0.21$, and $\rho = 2.45 \times 10^3$ kg/m^3, and the ultimate deformation for spring breakage is $u_n{}^* = 0.003$ mm. A uniformly distributed blasting load is applied on the inner surface. The blast loading is represented as a triangle pressure load that has a peak pressure of 1598 MPa; the rise time and total duration time are of 15 μs and 100 μs, respectively. Fig. 9 shows the fracturing process of the hollow

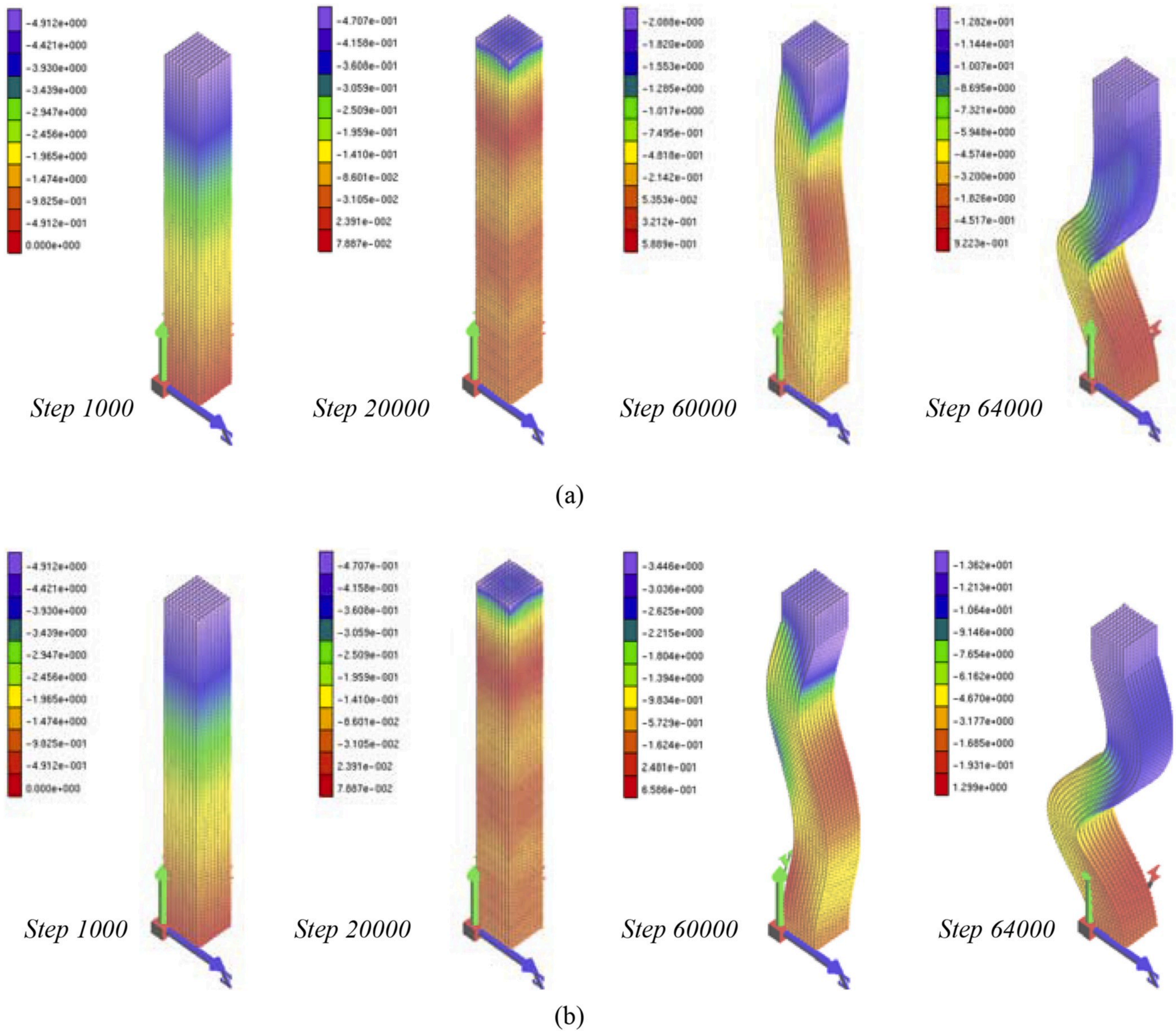


Fig. 7. Bulking process of the bar predicted by the multi-core CPU 4D-LSM and GPU 4D-LSM: (a) multi-core CPU 4D-LSM and (b) GPU 4D-LSM.

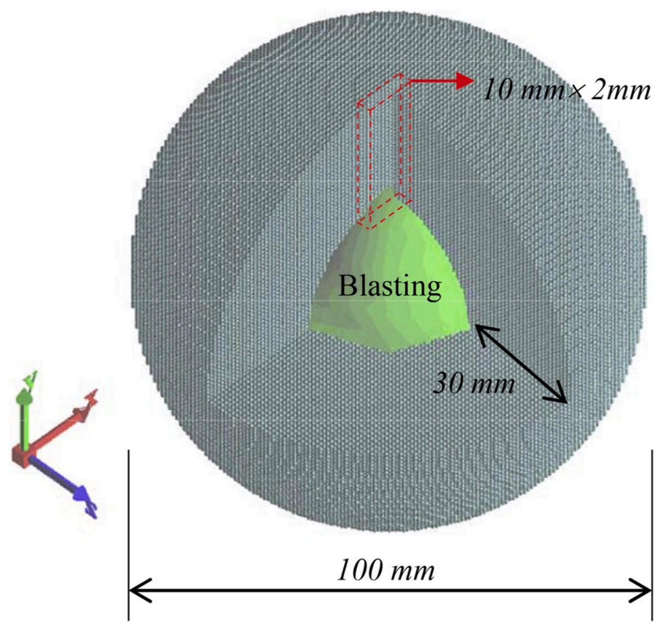


Fig. 8. Computational model of a sphere with surface pre-crack under blasting.

sphere predicted by the multi-core CPU 4DLSM and GPU 4D-LSM. Generally, the results of the multi-core CPU 4D-LSM (Fig. 9a) and GPU 4D-LSM (Fig. 9b) are similar, especially the first three images. The surface cracks extend symmetrically along the short edges of the original crack in an arc. However, as the calculation continues, differences between the multi-core CPU 4D-LSM and the GPU 4D-LSM begin to appear, and the final form of surface cracks correspondingly shows the difference in detail.

3.4. Deformation of a cube under gravity

In this example, the dynamic deformation process of a cube under gravity is solved by the fully coupled CPU-GPU 4D-LSM. As is shown in Fig. 10, the cube has a dimension of 50 mm × 50 mm × 50 mm; its bottom is fixed along the y direction, and under gravity, the cube will deform gradually. There are 125,000 particles with a diameter of 1 mm. The material properties are $E = 10$ GPa, $\nu = 0.2$, and $\rho = 2.45 \times 10^3$ kg/m^3. The domain decomposition is adopted in this example, and the model is divided into CPU and GPU parts. The y-direction velocity contour of the cube at 50 μs is predicted by the 4D-LSM using 100% of the CPU domain, 100% of the GPU domain, and 50% of the CPU and 50% GPU domains are given in Fig. 10 (b), (c) and (d), respectively. Good spatial symmetry is presented in the results solved with 100% CPU domain (Figs. 10b) and 100% GPU domain (Fig. 10c), and the maximum particle velocities between them only differs approximately 0.5%. However, when the 50% CPU and 50% GPU domains are used, the symmetry breaks down (Fig. 10 d), and the maximum particle velocity differs by 8% compared with the pure CPU or GPU computing. This example indicates that the full CPU-GPU heterogeneous parallelization for 4D-LSM is still unsuitable due to the different computing precision of the CPU and GPU. A further study on eliminating the difference of GPU and CPU computing is required.

3.5. Performance analysis using cubes with different sizes

The cube deformation problem in the previous section is further adopted to study the parallel computing performance of the multi-core CPU 4D-LSM and GPU 4D-LSM. To consider influence of the number of particles, as shown in Fig. 11, three cubes are built, i.e., 50 mm × 50 mm × 50 mm, 100 mm × 100 mm × 100 mm and 150 mm × 150 mm × 150 mm. To study the influence of different computational configurations, we used three computers. The first one was a PC equipped with an Intel Core i5-6400 CPU, a memory of 8G, and a GTX1060 GPU card. The second computer was a workstation equipped with two Intel Xeon E5-2630 v4 CPUs, memory of 64G, and a Quadro M5000 GPU. The third computer was a workstation equipped with two Intel Xeon E5-2660 v3 CPUs, memory of 32G, and a Quadro K5200 GPU. The cube deformation problems were solved with the first two computers using the multi-core CPU 4D-LSM and GPU 4D-LSM. The corresponding computational times are listed in Table 1. To compare the parallel efficiency of the multi-core CPU 4D-LSM and GPU 4D-LSM using different computers, we calculated the nominal speedup ratio based on the corresponding computational time of the PC with a single CPU thread. The corresponding results are plotted in Fig. 12. For multi-core CPU 4D-LSM, the PC using four CPU threads could obtain almost 3 times the speed up, while the workstation

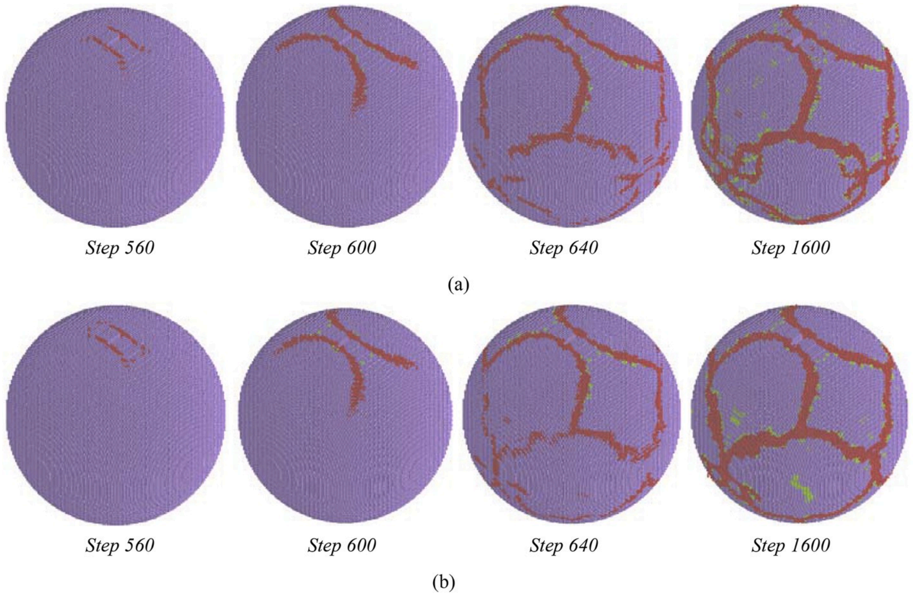


Fig. 9. Surface fracturing process of the sphere under blasting predicted by the multi-core CPU 4D-LSM and GPU 4D-LSM: (a) multi-core CPU 4D-LSM and (b) GPU 4D-LSM.

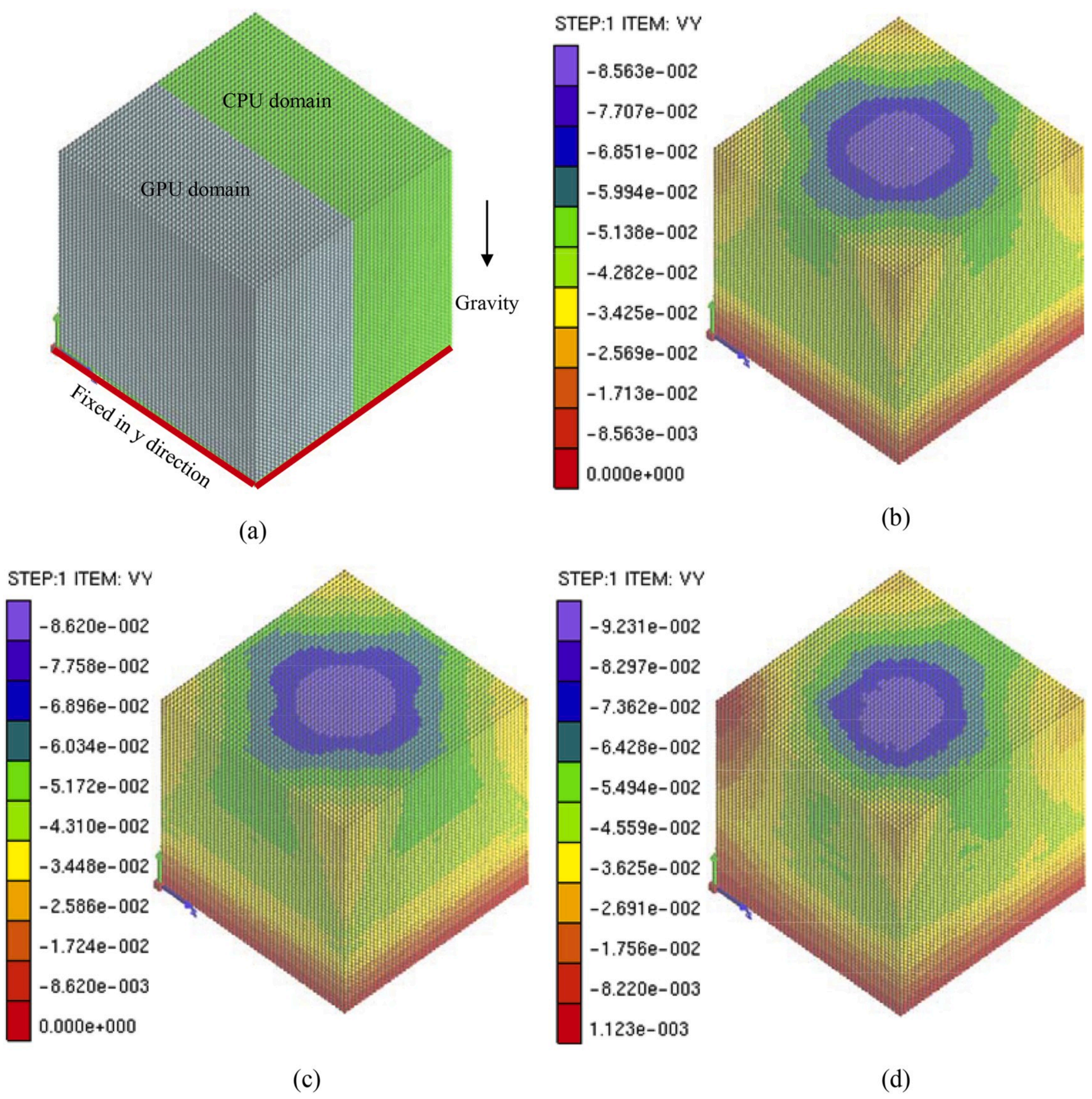


Fig. 10. Deformation of a cube under gravity: (a) computational model with domain decomposition, (b) velocity contour map in the *y* direction predicted by the CPU-GPU 4D-LSM with 100% CPU domain, (c) velocity contour map in the *y* direction predicted by the CPU-GPU 4D-LSM with 100% GPU domain, and (d) velocity contour map in the *y* direction solved by the CPU-GPU 4D-LSM with 50% CPU domain and 50% GPU domain.

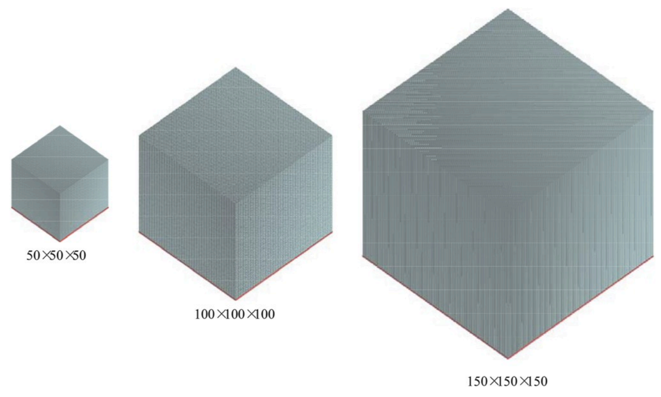


Fig. 11. Computational models of the cube under the gravity problem using a different number of particles.

Table 1
Computational time of the multi-core CPU 4D-LSM and GPU 4D-LSM using different computers (seconds).

Solvers	Computers	Cube 50	Cube 100	Cube 150
Multi-core CPU 4D-LSM	PC (CPU threads = 1)	75.209	639.045	2153.960
Multi-core CPU 4D-LSM	PC (CPU threads = 4)	27.417	210.664	691.070
Multi-core CPU 4D-LSM	WS (CPU threads = 40)	9.703	88.100	307.273
GPU 4D-LSM	PC (GPU threads = 3200)	6.411	51.046	184.821

with 40 CPU threads could obtain 8 times speedup. Using a GTX 1060 GPU, the GPU 4D-LSM could obtain approximately 12 times speedup. Compared with the investment of the PC equipped with a GTX 1060 and that of the workstation with two Intel Xeon CPUs, the GPU computing was economically promising. The second computer had 20 CPU cores and could provide 40 CPU threads, which was used to test the parallel computing performance of the multi-core CPU 4D-LSM. Fig. 13 shows speed up of the multi-core CPU 4D-LSM for the cube deformation problem using different numbers of CPU threads. The speed up is calculated based on the time that the workstation used one CPU thread.

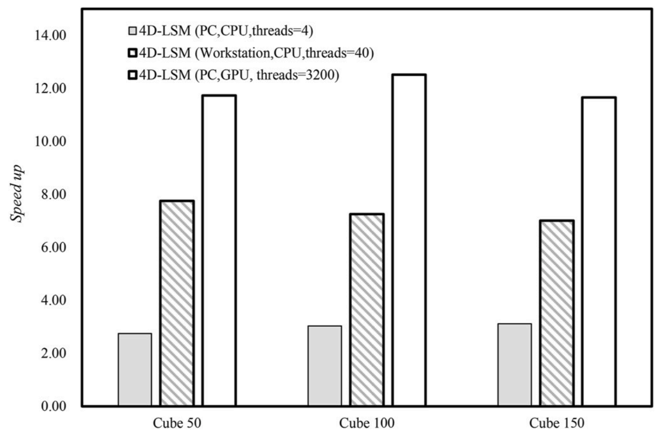


Fig. 12. Speed up comparison between the multi-core CPU 4D-LSM and GPU 4D-LSM.

As shown in these speed-up curves, computational efficiency is affected by the number of particles. Nevertheless, the overall tendency is still the same. When the number of CPU threads is under 10, speed up of the CPU 4D-LSM has a good scaling efficiency, while when the number of CPU threads is over 10, the increase of speed up obviously declines. This phenomenon is related to the transmission and reading mechanism of the multi-core CPU's memory and cache. When the CPU threads increase too much, the system consumes a large number of resources to coordinate each CPU thread to read and transfer the model data. Therefore, parallel efficiency decreases. It is a disadvantage of the parallel algorithm based on shared memory configuration.

In GPU computation, the number of blocks and threads per block required to be set are related to the research of the creation of the most efficient parallel computing. Table 2 lists the computational time of the GPU 4D-LSM using the PC with the GTX 1060 GPU through setting different GPU blocks and threads configurations. To have a more effective comparison, the computational time of the GPU 4D-LSM was scaled with the corresponding computational time of the CPU 4D-LSM using a single CPU thread (shown in Fig. 14). There is a peak value in the speed up of each cube, which is almost 2.5 times of total CUDA cores. To further verify this setting, two different graphics cards (Quadro K5200 and Quadro M5000) were used to calculate the 50 mm × 50 mm × 50 mm cube. The corresponding computational time versus the total number of GPU threads are shown in Fig. 15. According to the valley location of computational time, the GPU threads configuration of 2.5 times total CUDA cores is still applicable to the Quadro K5200 and Quadro M5000 GPUs. Therefore, for the GPU 4D-LSM, we suggest the

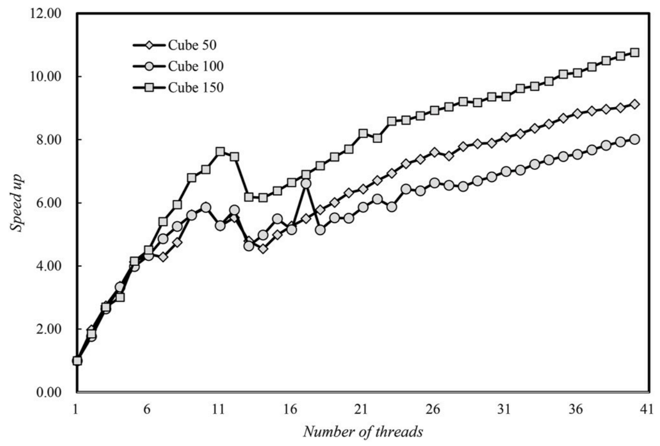


Fig. 13. Speed up of the multi-core CPU 4D-LSM for the cube deformation problem using a different number of CPU threads.

Table 2
Computational time of the GPU 4D-LSM using the PC with GTX 1060 GPU.

Blocks	Threads per block	Total threads	Computational time (s)		
			Cube 50	Cube 100	Cube 150
10	32	320	26.659	213.779	737.379
10	64	640	15.232	119.756	417.024
10	128	1280	9.088	69.976	244.733
10	160	1600	8.205	65.775	229.351
10	200	2000	7.463	58.698	207.27
10	300	3000	6.664	52.545	187.919
50	32	1600	8.243	65.931	228.863
50	64	3200	6.411	51.046	184.821
50	128	6400	6.233	54.247	198.544
50	160	8000	6.739	57.299	201.01
50	200	10000	6.839	63.093	226.262
50	300	15000	7.078	57.933	206.35
100	32	3200	6.42	51.313	183.862
100	64	6400	6.244	54.07	197.592
100	128	12800	7.138	57.103	208.803
100	160	16000	6.727	56.944	201.188
100	200	20000	6.669	57.124	201.749
100	300	30000	6.883	59.511	206.82

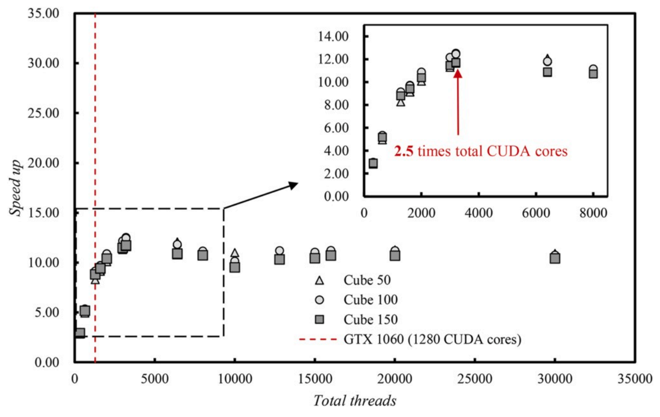


Fig. 14. Speed up of the GPU 4D-LSM for the cube deformation problem using a different number of GPU threads.

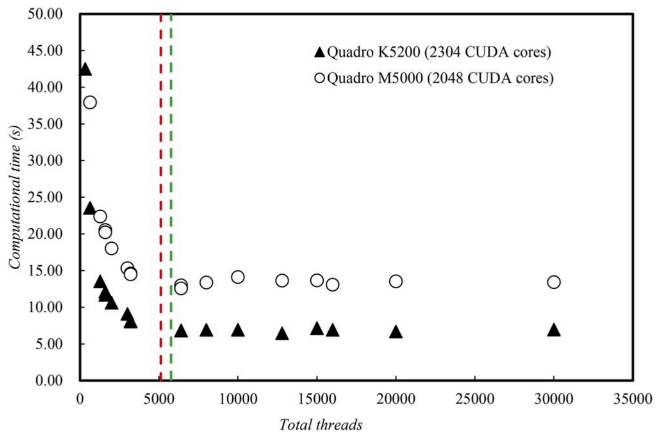


Fig. 15. The optimal GPU threads of the GPU 4D-LSM for the cube deformation problem on Quadro K5200 and Quadro M5000 GPUs (two dash lines refers to 2.5 times of the total CUDA cores of two GPUs, respectively).

number of total GPU threads as being 2.5 times the total CUDA cores in the corresponding GPU.

3.6. Crack propagation of a notched plate with hole under tension

In this example, the multi-core CPU 4D-LSM is further used to solve a crack propagation test of a notched plate under tension, which was previous solved by FEM with phase field models.[27] Fig. 16 shows the geometric parameters of the notched plate with hole and the corresponding 3D computational model adopted in this work. The overall size of the plate is 120 mm × 65 mm × 15 mm. The radii of the two small holes for boundary conditions are both 5 mm, one of the centres is 20 mm from the left boundary and 20 mm from the top boundary, and the other centre is 20 mm from the left boundary and 20 mm from the bottom boundary. An upward velocity loading is applied to the upper hole, while the lowest hole is fixed in the vertical direction. The radius of the large hole is 10 mm, and its centre is 28.5 mm from the right boundary and 51 mm from the bottom boundary. The original notch runs through the front surface and the rear surface, with 10 mm as the length and 1 mm as the thickness, and its centre line is 65 mm from the bottom boundary. There are 109,710 particles with a diameter of 1 mm. The material properties are $E = 5.98$ GPa, $\nu = 0.22$, $\rho = 2.00 \times 10^3$ kg/m^3, and the ultimate deformation for the spring breakage is $u_n^* = 3.34 \times 10^{-3}$ mm. Fig. 17a and (b) show the displacement of the upper hole versus the reaction force curve and the crack propagation process of the notched plate under tension. With the expansion of the crack, the bearing capacity decreases rapidly. The descending section in the curve of the second crack is not as smooth as the first one. The first crack is longer but costs less time, therefore, it has a faster crack speed. This observation is in agreement with the numerical observation using the FEM with a phase field model.[27] Fig. 18 further shows the final form of the cracked pattern of the specimen, including experimental data, FEM simulations (anisotropic field, hybrid field), the DLSM, and the 4D-LSM simulation. Both the FEM and 4D-LSM are generally consistent with the experimental data. There are some differences, e.g., the first crack of the numerical simulation with the FEM and 4D-LSM has a horizontal section or even moves upward, while that of the experimental simulation develops in a bit of a downward direction. For the secondary crack, the experimental observation is a roughly horizontal crack beginning at the upper part of the large hole (see Fig. 18a). The FEM and DLSM results are oblique, or the horizontal crack is at the middle of the large hole, whereas the initial position of the 4D-LSM is closer to the experimental observation. The reason for this might be that the 4D-LSM has automatically considered the nonlinear dynamic during the test. However, a full investigation of the ability of 4D-LSM on modelling crack propagation[28] is still needed, which will be considered in future studies.

4. Conclusions

In this work, 4D-LSM is parallelized to fully utilize the multi-core CPU and GPU heterogeneous computing resources of modern computers using OpenMP and CUDA parallel technology. Details of the parallelization implementation are presented. From a number of numerical examples including elastic deformation, nonlinear buckling and dynamic fracturing, the influences of digital precision on parallel computing performance of 4D-LSM using multi-core CPU and GPU are fully investigated. It was found that both multi-core CPU 4D-LSM and GPU 4D-LSM can provide a speed up ratio of approximately 10. An order parallel speedup is very promising for long time numerical simulation using the 4D-LSM, therefore, the parallel implementation of the 4D-LSM in this work is successful. For the numerical simulation, we found that the optimal number of CPU threads for the multi-core CPU 4D-LSM is approximately 10 in a workstation with 40 available CPU threads. In this sense, the CPU 4D-LSM still cannot fully utilize the computational resources provided by the modern multi-core CPU in a workstation. The optimal GPU threads for the GPU 4D-LSM was found to be 2.5 times the total CUDA cores available in the GPU. In summary, the GPU 4D-LSM can solve a problem faster than its CPU counterpart but still falls in the same order. Because computational investment of GPU computing is much lower than that of multi-core CPU computing, the GPU computing for 4D-LSM is still promising. Moreover, based on our numerical results, we found that a double digital precision is highly necessary for the 4D-

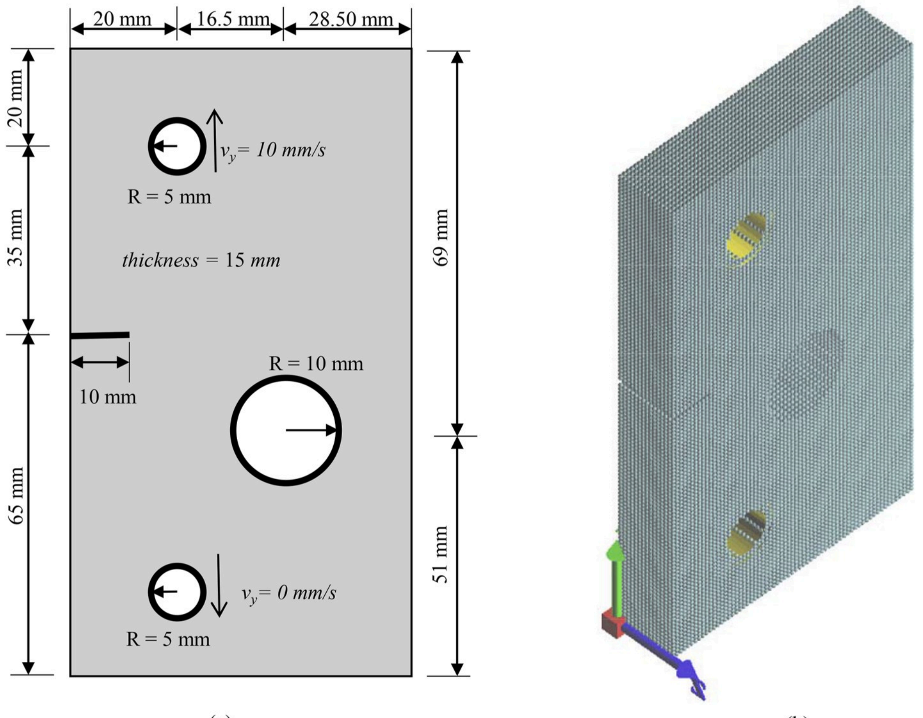
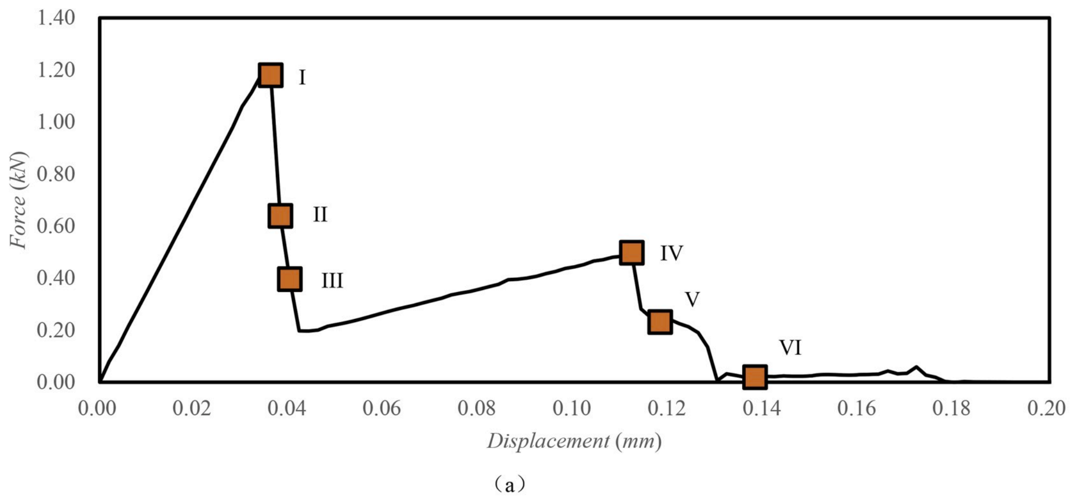


Fig. 16. Geometry and computational model of a notched plate with hole: (a) geometry model and (b) computational model of 4D-LSM.



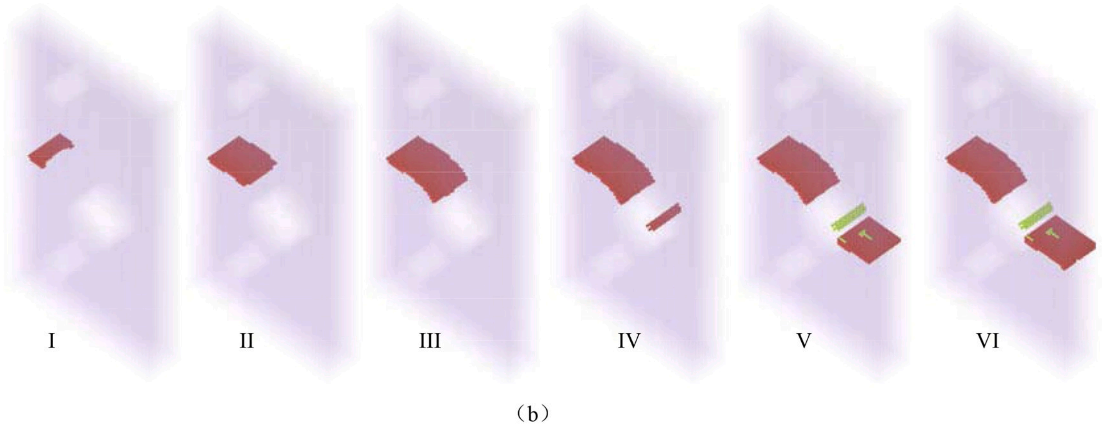


Fig. 17. Results of the notched plate with hole problem predicted by the multi-core CPU 4D-LSM: (a) displacement versus reaction force and (b) crack patterns at different stages (red refers to broken particles in tension and green refers broken particles in compression). (For interpretation of the references to colour in this figure legend, the reader is referred to the Web version of this article.)

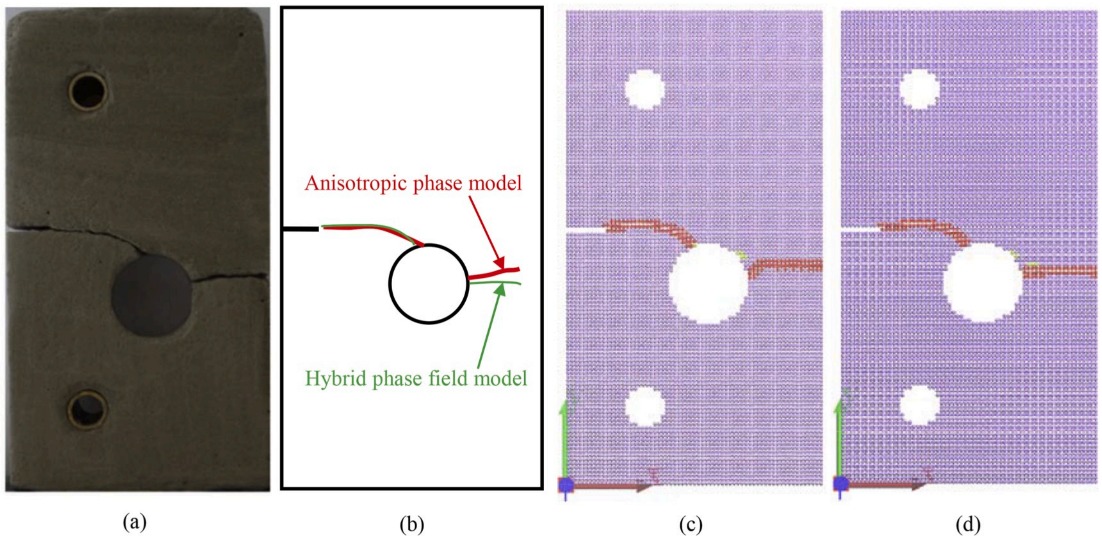


Fig. 18. Comparison between the experimental and numerical results (a) experimental result,[27] (b) numerical result predicted by anisotropic phase models,[27] (c) numerical result predicted by the multi-core CPU DLSM, and (d) numerical result predicted by the multi-core CPU 4D-LSM.

LSM. Even with a double digital precision, the GPU 4D-LSM and multi-core CPU 4D-LSM still produce different numerical results for nonlinear problems. Thus, GPU computing is still suggested to be suitable for a numerical simulation of problems at the linear elastic stage. GPU 4D-LSM can also be used as a fast numerical tool to obtain the approximation of material parameters for nonlinear problems. However, computational difference between the multi-core CPU 4D-LSM and GPU 4D-LSM handicaps the full CPU and GPU heterogeneous parallelization of the 4D-LSM. Future work to eliminate this digital difference is required to fully utilize the computational resources provided in modern workstations, where both multi-core CPU and GPU are computationally powerful.

Declaration of competing interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

This research is financially supported by the National Key R&D Program of China (Grant No. 2018YFC0406804) and National Natural Science Foundation of China (Grant No. 11772221).

Appendix E. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.ijrmms.2020.104361.

Appendix A. Pseudocode of the CPU 4D-LSM

The particle position update:

```
for (int i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (4)
  Particle_x1[i]= Particle_x1[i]+Particle_v1[i]*dt;
  Particle_x2[i]= Particle_x2[i]+Particle_v2[i]*dt;
  Particle_x3[i]= Particle_x3[i]+Particle_v3[i]*dt;
  Particle_x4[i]= Particle_x4[i]+Particle_v4[i]*dt;
}
```

The particle force calculation step in Fig. 3:

```
for (int i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (8)
  Particle_f1[i]= Particle_m[i]*gx;
  Particle_f2[i]= Particle_m[i]*gy;
  Particle_f3[i]= Particle_m[i]*gz;
  Particle_f4[i]= 0.0;
  for(int j=0;j<m;j++)
  {
    //Equation (7)
    Particle_f1[i]+= SpringForce_1(i,j);
    Particle_f2[i]+= SpringForce_2(i,j);
    Particle_f3[i]+= SpringForce_3(i,j);
    Particle_f4[i]+= SpringForce_4(i,j);
  }
}
```

The particle acceleration calculation:

```
for (int i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (9)
  Particle_a1[i]= Particle_f1[i]/Particle_m[i];
  Particle_a2[i]= Particle_f2[i]/Particle_m[i];
  Particle_a3[i]= Particle_f3[i] /Particle_m[i];
  Particle_a4[i]= Particle_f4[i] /Particle_m[i];
}
```

The motion update step:

```
for (int i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (5)
  Particle_v1[i]= Particle_v1[i]+Particle_a1[i]*dt;
  Particle_v2[i]= Particle_v2[i]+Particle_a2[i]*dt;
  Particle_v3[i]= Particle_v3[i]+Particle_a3[i]*dt;
  Particle_v4[i]= Particle_v4[i]+Particle_a4[i]*dt;
}
```

Appendix B. Pseudocode of the multi-core CPU 4D-LSM

The particle position update:

```
int i;
#pragma omp parallel for
for (i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (4)
  Particle_x1[i]= Particle_x1[i]+Particle_v1[i]*dt;
  Particle_x2[i]= Particle_x2[i]+Particle_v2[i]*dt;
  Particle_x3[i]= Particle_x3[i]+Particle_v3[i]*dt;
  Particle_x4[i]= Particle_x4[i]+Particle_v4[i]*dt;
}
```

The particle force calculation:

```
int i;
#pragma omp parallel for
for (i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (8)
  Particle_f1[i]= Particle_m[i]*gx;
  Particle_f2[i]= Particle_m[i]*gy;
  Particle_f3[i]= Particle_m[i]*gz;
  Particle_f4[i]= 0.0;
  for(int j=0;j<m;j++)
  {
    //Equation (7)
    Particle_f1[i]+= SpringForce_1(i,j);
    Particle_f2[i]+= SpringForce_2(i,j);
    Particle_f3[i]+= SpringForce_3(i,j);
    Particle_f4[i]+= SpringForce_4(i,j);
  }
}
```

The particle acceleration calculation:

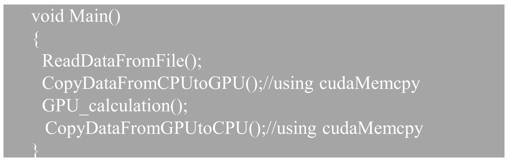
```
int i;
#pragma omp parallel for
for (i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (9)
  Particle_a1[i]= Particle_f1[i]/Particle_m[i];
  Particle_a2[i]= Particle_f2[i]/Particle_m[i];
  Particle_a3[i]= Particle_f3[i] /Particle_m[i];
  Particle_a4[i]= Particle_f4[i] /Particle_m[i];
}
```

The motion update step:

```
int i;
#pragma omp parallel for
for (i = 0; i < N; i++) // N is the total number of particles
{
  //Equation (5)
  Particle_v1[i]= Particle_v1[i]+Particle_a1[i]*dt;
  Particle_v2[i]= Particle_v2[i]+Particle_a2[i]*dt;
  Particle_v3[i]= Particle_v3[i]+Particle_a3[i]*dt;
  Particle_v4[i]= Particle_v4[i]+Particle_a4[i]*dt;
}
```

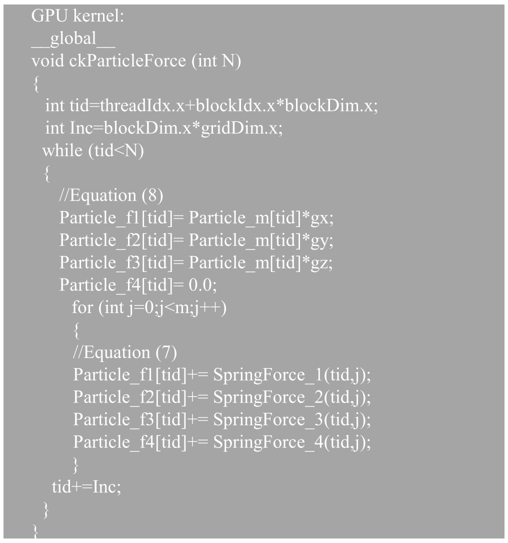
Appendix C. Pseudocode of the GPU 4D-LSM

The main framework:



```
void Main()
{
  ReadDataFromFile();
  CopyDataFromCPUtoGPU();//using cudaMemcpy
  GPU_calculation();
   CopyDataFromGPUtoCPU();//using cudaMemcpy
}
```

The particle force calculation:



```
GPU kernel:
__global__
void ckParticleForce (int N)
{
   int tid=threadIdx.x+blockIdx.x*blockDim.x;
   int Inc=blockDim.x*gridDim.x;
  while (tid<N)
  {
      //Equation (8)
      Particle_f1[tid]= Particle_m[tid]*gx;
      Particle_f2[tid]= Particle_m[tid]*gy;
      Particle_f3[tid]= Particle_m[tid]*gz;
      Particle_f4[tid]= 0.0;
         for (int j=0;j<m;j++)
         {
         //Equation (7)
         Particle_f1[tid]+= SpringForce_1(tid,j);
         Particle_f2[tid]+= SpringForce_2(tid,j);
         Particle_f3[tid]+= SpringForce_3(tid,j);
         Particle_f4[tid]+= SpringForce_4(tid,j);
         }
    tid+=Inc;
  }
}
```

Appendix D. Pseudocode of the CPU and GPU 4D-LSM

The particle position update:

```
GPU kernel:
__global__
void ckParticlePosition (int Ngpu)
{
  int tid=threadIdx.x+blockIdx.x*blockDim.x;
  int Inc=blockDim.x*gridDim.x;
 while (tid<Ngpu)
  {
     //Equation (8)
     Particle_f1[tid]= Particle_m[tid]*gx;
     Particle_f2[tid]= Particle_m[tid]*gy;
     Particle_f3[tid]= Particle_m[tid]*gz;
     Particle_f4[tid]= 0.0;
       for (int j=0;j<m;j++)
      {
     //Equation (7)
       Particle_f1[tid]+= SpringForce_1(tid,j);
       Particle_f2[tid]+= SpringForce_2(tid,j);
       Particle_f3[tid]+= SpringForce_3(tid,j);
       Particle_f4[tid]+= SpringForce_4(tid,j);
      }
      tid+=Inc;
  }
}

CPU function:
void cpuParticlePosition (int Ngpu, int N)
{
 int i;
 #pragma omp parallel for
 for (i = Ngpu+1; i < N; i++) // N is the total number of
particles
  {
      //Equation (4)
      Particle_x1[i]= Particle_x1[i]+Particle_v1[i]*dt;
      Particle_x2[i]= Particle_x2[i]+Particle_v2[i]*dt;
      Particle_x3[i]= Particle_x3[i]+Particle_v3[i]*dt;
      Particle_x4[i]= Particle_x4[i]+Particle_v4[i]*dt;
  }
}
```

References

1 Zhao GF. Developing a four-dimensional lattice spring model for mechanical responses of solids. *Comput Methods Appl Mech Eng*. 2017;315:881–895.
2 Hrennikoff A. Solution of problems of elasticity by the framework method. *J Appl Mech*. 1941;8:A169–A175.
3 Tang C. Numerical simulation of progressive rock failure and associated seismicity. *Int J Rock Mech Min*. 1997;34(2):249–261.
4 Kakouris EG, Triantafyllou SP. Phase-field material point method for brittle fracture. *Int J Numer Methods Eng*. 2017;112(12):1750–1776.
5 Espinha R, Park K, Paulino GH, Celes W. Scalable parallel dynamic fracture simulation using an extrinsic cohesive zone model. *Comput Methods Appl Mech Eng*. 2013;266(11):144–161.
6 Xu XP, Needleman A. Numerical simulations of fast crack growth in brittle solids. *J Mech Phys Solid*. 1994;42(9):1397–1434.
7 Wu Z, Fan L, Liu Q, Ma G. Micro-mechanical modeling of the macro-mechanical response and fracture behavior of rock using the numerical manifold method. *Eng Geol*. 2017;225(20):49–60.
8 Ooi ET, Yang ZJ, Guo ZY. Dynamic cohesive crack propagation modelling using the scaled boundary finite element method. *Fatigue Fract Eng M*. 2012;35(8):786–800.
9 Radovitzky R, Seagraves A, Tupek M, Noels L. A scalable 3d fracture and fragmentation algorithm based on a hybrid, discontinuous galerkin, cohesive element method. *Comput Methods Appl Mech Eng*. 2011;200(1–4):326–344.
10 Papadrakakis M, Stavroulakis G, Karatarakis A. A new era in scientific computing: domain decomposition methods in hybrid cpu-gpu architectures. *Comput Methods Appl Mech Eng*. 2011;200(13–16):1490–1508.
11 https://www.intel.com/content/www/us/en/products/processors.html. accessed on 12/04/2017.
12 https://developer.nvidia.com/cuda-zone. accessed on 12/04/2017.
13 http://www.openmp.org/. accessed on 12/04/2017.
14 Takahashi T, Hamada T. Gpu-accelerated boundary element method for helmholtz' equation in three dimensions. *Int J Numer Methods Eng*. 2009;80(10):1295–1321.
15 Ta T, Choo K, Tan E, Jang B, Choi E. Accelerating dynearthsol3d on tightly coupled cpu-gpu heterogeneous processors. *Comput Geosci*. 2015;79(C):27–37.
16 Dong Y, Wang D, Randolph MF. A gpu parallel computing strategy for the material point method. *Comput Geotech*. 2015;66:31–38.
17 Hazeghian M, Soroush A. Dem simulation of reverse faulting through sands with the aid of gpu computing. *Comput Geotech*. 2015;66(52):253–263.
18 Yue X, Zhang H, Ke C, et al. A gpu-based discrete element modeling code and its application in die filling. *Comput Fluids*. 2015;110:235–244.
19 Zhao G-F. *Discrete Element Model and High Performance Computing*. ISTE & Elsevier; 2015. ISBN 978-1-78548-031-7.
20 Cercos-Pita JL. Aquagpusph, a new free 3d sph solver accelerated with opencl. *Comput Phys Commun*. 2015;192:295–312.
21 Fu X, Sheng Q, Zhang Y, Chen J. Investigation of highly efficient algorithms for solving linear equations in the discontinuous deformation analysis method. *Int J Numer Anal Methods GeoMech*. 2015;40(4):469–486.
22 Zhao G-F, Fang J, Zhao J. A 3D distinct lattice spring model for elasticity and dynamic failure. *Int J Numer Anal Methods GeoMech*. 2011;35:859–885.
23 Liu Q, Wang W, Ma H. Parallelized combined finite-discrete element (fdem) procedure using multi-gpu with cuda. *Int J Numer Anal Methods GeoMech*. 2020;44(2):208–238.
24 Lisjak A, Mahabadi OK, He L, et al. Acceleration of a 2d/3d finite-discrete element code for geomechanical simulations using general purpose gpu computing. *Comput Geotech*. 2018;100:84–96.

25 Zhao G-F, Fang J, Sun L, Zhao J. Parallelization of the distinct lattice spring model. *Int J Numer Anal Methods GeoMech*. 2013;37(1):51–74.
26 Zhao G-F, Deng Z-Q, Zhang B. Multibody failure criterion for the four-dimensional lattice spring model. *Int J Rock Mech Min Sci*. 2019;123:104126.
27 Ambati M, Gerasimov T, Lorenzis L. A review on phase-field models of brittle fracture and a new fast hybrid formulation. *Comput Mech*. 2015;55(2):383–405.
28 Jiang C, Zhao G-F, Khalili N. On crack propagation in brittle material using the Distinct Lattice Spring Model. *Int J Solid Struct*. 2017;118–119:41–57.