COMPUTER-AIDED DESIGN & APPLICATIONS, 2017
VOL. 14, NO. 2, 140–147
http://dx.doi.org/10.1080/16864360.2016.1223425

An efficient parallel method for photo-realistic fluid animation

Guijuan Zhang [a,b], Jinyan Zhao [c], Weizhi Xu [b], Dianjie Lu [b], Yongjian Wang [d] and Xiangxu Meng [a]

[a]School of Computer Science and Technology, Shandong University, China; [b]School of Information Science and Engineering, Shandong Normal University, China; [c]Hunan Institute of Science and Technology, China; [d]Institute of Computing Technology, Chinese Academy of Sciences, China

ABSTRACT
Fluid animation often appears in applications such as games, films and cartoons. How to animate photo-realistic fluid motion efficiently is an important issue. We present an efficient parallel method for photo-realistic fluid animation in this paper. Our method is designed to generate fluid animation results with high efficiency on a cluster system. To do this, we categorize the computers in our cluster system into two classes, the server and the client. The server controls the process of the fluid animation while the clients are responsible for numerical computation. Given 3D virtual environment and fluid initial condition, we make pre-processing on the server so as to decompose the fluid animation task into several subtasks. Thus, the computation domain is divided into blocks and each client executes numerical computation for one block. The blocks of two adjacent clients are overlapped to keep the continuity of the solution across subdomain interface. We demonstrate the efficiency of our method by animating the motion of smoke and liquid. Results show that the proposed parallel algorithm can improve the computation speed of physically-based fluid animation significantly while getting interesting fluid details.

KEYWORDS
Fluid animation; photo-realistic; parallel algorithm; preconditioned conjugate gradient method

1. Introduction

Fluid animation is widely used when generating real-life phenomena in many applications such as films, cartoons and real-time computer games. How to animate fluid motion with high degree of visual realism efficiently is always a hot topic. To get animation results with sufficient fluid details, researchers propose the physically-based animation method [4,14] which can provide interesting animation results. However, it is rather difficult to compute the physical model efficiently since fluid often has complex behavior and rich visual details. The main reason is that the physically based methods consume a large amount of hardware memory which is too expensive to afford for common users. The long computation time caused by solving the partial equation and large-scale linear system is another important reason. Recently, the algorithms of parallelization has been proposed in many applications. In these methods, the powerful many-core processors can be utilized to speed-up computations while reducing the hardware cost. Therefore, exploring the parallel nature of fluid animation algorithms and implement it on parallel architecture is very important for improving efficiency.

Some researchers present parallel method especially GPU-based approaches [1,3,6,7,9,13,15] to improve the efficiency. In fluid animation world, physically-based animation methods are categorized into two kinds: Lagrangian methods and Eulerian methods. Most of current parallel methods for fluid animation focus on Lagrangian methods rather than Eulerian method. The main reason is that Eulerian method requires solving large-scale sparse linear system which is rather difficult to implement on multi-cores system. Recently, high performance computer, multi-core chip and cluster system grow quickly. To utilize the parallel structure of these systems is particularly applicable to speed up computation intensive tasks, and thus, many applications appeared on these systems such as video processing [13] etc. Making full use of parallel structure of modern computer systems for speeding up fluid computation algorithm is essential for improving the efficiency of animation production.

In this paper, we present a parallel mechanism for photo-realistic fluid animation on cluster system in this paper. Our method can animate liquid with rich details and high efficiency. Given 3D virtual environment and flu-id initial condition, we make pre-processing on the server computer so as to decompose the fluid animation task into subtasks. The computation domain is divided into blocks. Each client is responsible for one block. The blocks of two adjacent clients are overlapped to keep

CONTACT Guijuan Zhang guijuanzhang@gmail.com

the computation correctly. To reduce the cost of message transmission, we put fluid source file and voxelize files of 3D environment into a shared file system. Then, the shared system can be obtained in read-only mode by all clients. As for each client, when it receives message from the server program, it executes the computation process for the subdomain. Two clients that have adjacent computational domain may transmits information to keep the boundary condition correctly. Results show that our method can produce visual pleasing results while improves the animation efficiency significantly.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 present the overview of our main work. Sections 4 solves the physically-based fluid animation model to obtain interesting animation results which includes computing the fluid velocity and the motion of fluid substances. Section 5 gives a parallel implementation that consists of a server and several clients. Section 6 presents simulation results to evaluate the proposed algorithms, with conclusions following in Section 7.

2. Related work

Eulerian methods are popular in fluid animation world since it is good at capturing the motion of liquid surface [4]. The methods compute fluid velocity as well as liquid surface motion by creating grids that stores fluid quantities (e.g., density, velocity, temperature, etc.) and solves the partial differential equation. After that, the method has been extended to simulate other fluid phenomena include coupling between solid and fluid [1], multiphase flow [6], surface tension [2] and so on. Although Eulerian methods can capture surface details effectively, it suffers from strong numerical dissipation and requires high resolution grids to keep details. Thus, it is expensive to compute fluid equation on PC and it is necessary to find more effective way to animate fluids with Eulerian methods.

To speed up fluid animation, many researchers consider parallel implementation. For example, Kipfer et al. [7] animate fluid motion on terrains with parallel SPH method. Kurose et al. [9] simulate the two-way solid-fluid interaction according to parallel SPH method. The method adopts a set of particles to represent rigid-body and fluid, and formulates the interaction as a linear complementary problem which can be solved by Lemke's algorithm effectively. Zhang et al. [15] develop an adaptive parallel SPH method to speed up computation. One of the most challenging problems for parallel SPH methods is how to compute the neighbors of each particle effectively [3]. Most of the work [9,1] adopt uniform grid to compute particle neighbors but suffers from the disadvantage of growing grid size. To improve the efficiency of particle neighborhood queries, Ihmsen et al. [6] present two efficient mechanisms, spatial hashing and index sort. Goswami et al. [3] uses a parallel SPH simulation and rendering method on the GPU. The neighborhood search is implemented effectively according to Z-indexing and parallel sorting. However, most of the above methods are designed for Lagrangian method rather than Eulerian method. Xu et al. [5,11,12] proposed a framework of processing multimedia resources.

As the most popular scheme in fluid animation, Eulerian method is often used in the applications. But it requires global pressure correction and has poor scalability. Crane et al. [2] implement a method with Eulerian scheme to generate fluid effects such as smoke, water and fire. The method uses Jacobi iteration which counteracts the efficiency gain. We adopt preconditioned conjugate gradient (PCG) method for linear system solution which is more efficient than traditional Jacobi iteration method. Our method allows parallel fluid animation with Eulerian scheme on cluster system.

3. Overview

Given fluid initial and boundary condition, the goal of our method is to produce photo-realistic fluid effects in parallel system. The framework of our method is shown in Fig. 1. Our parallel fluid animation system includes three layers. The bottom layer is hardware system, which represents that our parallel algorithm can run on high-performance computer, multi-core chip and cluster system. The middle layer is our numerical computing platform. We process the given 3D model and obtain the boundary condition for fluid computation. It also includes the parallel numerical computation of

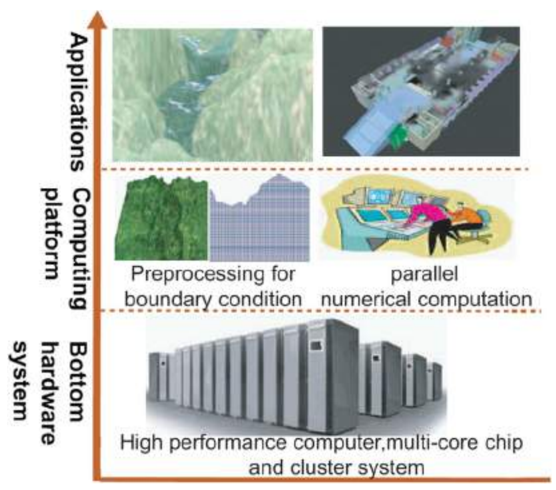
Applications
Computing platform
Bottom hardware system
Preprocessing for boundary condition
parallel numerical computation
High performance computer,multi-core chip and cluster system

Figure 1. The parallel framework of fluid animation model.

the fluid physical model. The top layer is application. In this layer, users input fluid source and the 3D scene that fluid flow, and the numerical results are returned from the middle layer. The numerical data are then processed for extracting the essential data (e.g., soot density, liquid mesh surface) that represents fluid. Finally, the Photo-realistic animation results are obtained from these numerical results using global illumination rendering method. Details will be described in the following sections.

4. Physically-based fluid animation model

We solve physically-based fluid animation model to obtain interesting animation results. It includes computing the fluid velocity as well as the motion of fluid substances.

4.1. Fluid velocity

To compute the fluid velocity u, we solve the following Navier-stokes equation

$$\nabla \cdot u = 0, \tag{4.1}$$

$$u_t = -u \cdot (\nabla u) + \nu \nabla \cdot \nabla u - \frac{1}{\rho} \nabla p + f, \tag{4.2}$$

where ν denotes the kinetic viscosity of the fluid, ρ is the density, p is the pressure, and f is the external force field. Since numerical computation may introduce diffusion effects to the results, we overlook the diffusion term $\nu \nabla \cdot \nabla u$ in Eqn. (4.2). Thus, the momentum equation in this paper is

$$u_t = -u \cdot (\nabla u) - \frac{1}{\rho} \nabla p + f. \tag{4.3}$$

We solve Eqn. (4.1) and Eqn. (4.3) with Eulerian scheme numerically. Therefore, the computational domain is discretized into a grid composed by computational cells. The values of each parameter (e.g., pressure p, velocity $\mathbf{u}(u, v, w)$, external force f, and so on) in the above equations are stored in the cells.

4.2. Smoke density computation

After obtaining the fluid velocity u, smoke will be advected under the velocity field

$$\frac{\partial \rho}{\partial t} = -u \cdot \nabla \rho + k_\alpha \nabla^2 + S_\alpha, \tag{4.4}$$

where ρ is smoke density, k_α is smoke diffusion constant and S_α is the source term that represents the injected smoke. We solve the advection term $u \cdot \nabla \rho$ with semi-Lagrange method [10] similar to the Eqn. (4.1) and Eqn. (4.3). Since the numerical computation also contains numerical diffusion intrinsically, we overlook the diffusion term in Eqn. (4.4).

4.3. Free surface computation

Most of liquid animation work focus on the complex behavior of its surface since it is the most interesting part. We use level set method [13] in this paper. So a signed distance function ϕ is defined and its zero iso-surface is used to represent water surface implicitly. The water surface motion can be described by the level set equation

$$\frac{\partial \phi}{\partial t} + u \cdot \nabla \phi = 0. \tag{4.5}$$

Obviously, Eqn. (4.5) is an advection equation, and we also solve it according to semi-Lagrange method. To make sure that the signed distance function ϕ is smooth (e.g., $\|\phi\| = 1$), we use Fast Marching algorithm [13] to re-initialize ϕ in each time step.

5. Parallel implementation of Eulerian method

We give a parallel implementation that consists of a server and several clients as shown in Fig. 2. Specifically, given a 3D virtual environment and the initial condition for fluid animation, the server first pre-processes the input so as to decompose the task into a number of subtasks and assign them to the clients. In each time step, the server collects the data from the shared file system and then renders the data with global rendering algorithm to get the animation results. As for each client, the fluid equations are solved in parallel when they receive the assigned subtask. The fluid computational domain is divided into blocks and each client is responsible for one block. To keep the continuity of the solution across subdomain interface, the blocks are overlapped. Messages about the overlapped domain are transmitted among adjacent clients. Details will be given in the following subsections.

5.1. System architecture

The architecture of the system is shown in Fig. 2. We use a client-server structure here. The server provides resource for start fluid computation and partitions the workload or tasks with load balance principle. The client executed the given partitioned task from the server. Clients and servers often communicate over a computer network on separate hardware, but both client and server may reside in the same system. Thus, it is a typical distributed application structure that can be executed on many parallel systems

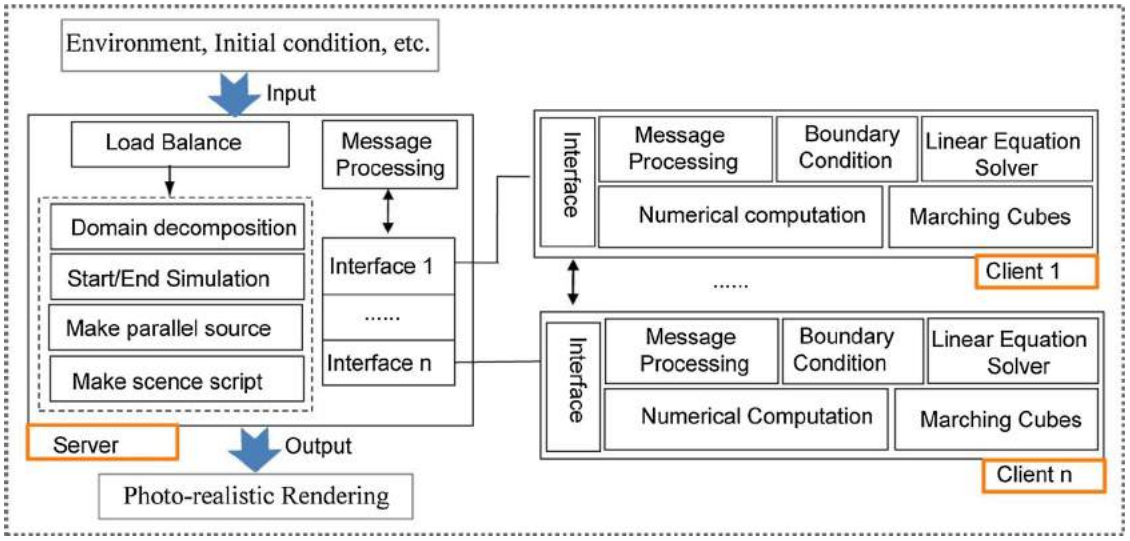

Figure 2. The parallel implementation of fluid animation model.

such as high performance computer, multi-core chip as well as cluster system.

When user input 3D environment for fluid flow and the initial condition, the server host runs a server program that process this information and send message to start the client program. Server program first voxelize the 3D environment into computable boundary conditions using a depth voxelization method [16]. Then, the fluid computation task will be decomposed into n subtasks. It is implemented by divide the fluid source into n parts. Details of the domain decomposition will be introduce in the next subsection. To reduce the cost of message transmission, we put fluid source file, voxelize files of 3D environment into a database that can be shared in read-only mode by all clients and server. Thus, processing of message may only requires to send the information about start and end position of the departed domain.

As for each client, when it receives message from the server program, it executes the computation process for its domain. Two clients that have adjacent computational domain may transmits information to keep the boundary condition correctly. Clients initiate communication sessions with the adjacent clients after each time step.

5.2. Domain decomposition

To solve the fluid equation in parallel, it is necessary to decompose the computational domain into subdomains. Specifically, domain decomposition methods split the above grids that stored fluid parameters into smaller subdomains. The problems of solving partial equation numerically on the subdomains are relatively independent which makes domain decomposition methods suitable for parallel computing. Each subdomain simulation is executed on a client and the solution between adjacent subdomains is coordinated to keep the numerical boundary condition correctly. In this paper, the server host coordinates the solution between the subdomains globally to get the combined animation results. In each time step, the numerical data is stored in the shared public file system and the server host analyzes these data to extract useful information (e.g., soot density for smoke animation, temperature for fire animation and surface mesh for liquid animation) for photo-realistic rendering.

We subdivide the computational domain into n blocks where n denotes the number of clients. In general, we divide the x, y and z coordinates into nx, ny and nz parts respectively and make sure that $n = nx \times ny \times nz$. Figure 3 shows three subdivision examples. In the left image of Fig. 3, the computational domain is subdivided into 8 cubic blocks where $n = 2 \times 2 \times 2$. Observe that there are many ways to break up 3D domain into n blocks. In the middle and right image of Fig. 3, other two subdivision results are given where $n = 4 \times 1 \times 2$ and $n = 1 \times 8 \times 1$. Users can subdivide in a way that they are used to but should consider the following subdivision principles.

5.2.1. Subdivision principles

When subdivide the computational domain, load balance principle should be considered for performance reasons. Load balancing requires distributing approximately equal amounts of work among computational tasks. This principle makes that all clients are kept busy all of the time and thus minimize the task idle time. If not, for example, some task will become the barrier synchronization point and the slowest one will dominate the overall computing time. It is intuitive to achieve load balance in our

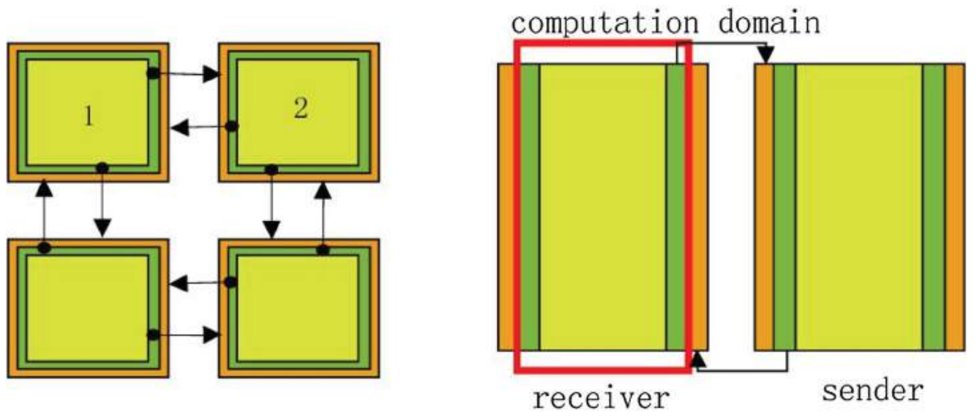

Figure 3. The parallel implementation of fluid animation model

method. We equally partition the task by subdivide the computational domain into n sub-regions evenly.

Another principle is to decrease the cost of communication as much as possible. In fluid simulation, the ratio of computation to communication is large. That is because the task is computational intensive. Compared to large cost of computation, the communication overhead is low and thus we only need to consider how to improve the efficiency of computation effectively.

5.2.2. *Map of neighbors*

Each block needs data from its neighbor blocks to keep the computation correctly. Thus, the parallel algorithm requires keeping a map of neighborhood blocks for each block. To do this, we index all blocks with the following order

$$b^{id}_{i,j,k} = i \times ny \times nz + j \times nz + ks, \tag{5.1}$$

where nx, ny and nz is the number of blocks on each dimension. We assign each block to a client whose number c is the index of the assigned block $b_{i,j,k}$, $c = b^{id}_{i,j,k}$. Thus, the six neighbors of client c are $c.left = b^{id}_{i-1,j,k}$, $c.right = b^{id}_{i+1,j,k}$, $c.top = b^{id}_{i,j-1,k}$, $c.bottom = b^{id}_{i,j+1,k}$, $c.front = b^{id}_{i,j,k-1}$ and $c.back = b^{id}_{i,j,k+1}$.

5.2.3. *Numerical computation on each node*

In each block, the fluid equations are solved with following steps. First, we construct the spatial grid for storing the scalar and vector parameters of fluid motion. Secondly, we solve Eqn. (4.1) and Eqn. (4.3). After obtaining fluid velocity, we drive the motion of fluid media, such as the soot density, temperature as well as fluid surface according to the obtained fluid velocity. Finally, the numerical data will be put to the public file sharing system. They will be used to extract visualization data so as to provide input for global illumination algorithm executed on server host. Solving the equations can be categorized into three operations: adding external force, advection and projection. According to the given initial condition, we set $\mathbf{u}_0 = \mathbf{u}_n = (u^n, v^n, w^n)$ in a block.

Add force. The first step is to add external force $\mathbf{f}$. Thus we get $\mathbf{u}_1(\mathbf{x}) = \mathbf{u}_0(\mathbf{x}) + \triangle t\mathbf{f}(\mathbf{x}, t)$.

Advection. In this step, we use Semi-Lagrange method to solve the advection term $\mathbf{u}_t = -\mathbf{u} \cdot (\nabla \mathbf{u})$. We regard each grid node $\mathbf{x}$ as a particle and then trace the particle back to get its position at the last time step $\mathbf{y}$ by the velocity of the fluid itself, $\mathbf{y} = \mathbf{x} - \mathbf{u}^{t+\triangle t}(\mathbf{x})\triangle t$. The scalar value at position $\mathbf{y}$ is calculated by linear interpolation and is finally assigned to the scalar value at grid node $\mathbf{x}$. Therefore, we can get each component of fluid velocity at a grid point $\mathbf{x}$ at the new time $t + \triangle t$ as $d^{t+\triangle t}(\mathbf{x}) = d^t(\mathbf{x} - \mathbf{u}^{t+\triangle t}(\mathbf{x})\triangle t)$. Similarly, the soot density ρ and signed distance function ϕ after advection can also be get according to the above formula.

Projection. After adding external force and advection step, there is only one term on the right side of the Eqn. (4.3), that is $-1/\rho\nabla p$. The final velocity is denoted as

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \frac{\triangle t}{\rho}\nabla p, \tag{5.2}$$

where $\mathbf{u}^*$ is the intermediate velocity obtained after the above two steps. So calculating $\mathbf{u}^{n+1}$ requires computing the pressure p firstly. So we apply divergence operator to both sides of equation and get

$$\nabla^2 p = \frac{\triangle t}{\rho}\nabla \cdot \mathbf{u}^*. \tag{5.3}$$

We use finite differences to approximate derivatives $\nabla^2 p$ and $\mathbf{u}^*$. Finally, Eqn. (5.3) becomes a sparse linear system $Ap = b$ where the pressure p is unknown.

Solving large sparse linear system. As described before, we are required to solve the linear system $Ap = b$. It should be solved globally because the coefficient matrix A is a global matrix. To construct A, we first get the index of each fluid cell in every client and send the number of fluid cells to the next client in order. The number is regarded as the offset of cell index. The cell index determines which column of matrix A is non-zero. Finally, we solve the sparse linear system $Ap = b$ with PCG method. Consider a sparse linear system

$$Ap = b, \tag{5.4}$$

where p is an unknown vector, b is a known vector, A is a known SPD matrix. According to PCG algorithm, Eqn. (5.4) can be written as

$$M^{-1}Ap = M^{-1}b, \tag{5.5}$$

where matrix M is a preconditioner [8]. We use a block diagonal preconditioner. The block diagonal preconditioner is formed from the incomplete Cholesky blocks on each client. After getting the preconditioner M, we set the start vector as $p = 0$. Given a maximum number of iterations k_{max} and an error tolerance ε, the PCG algorithm can enable us to compute the pressure p. In this algorithm, a set of α – orthogonal search directions $\alpha_1, \cdots, \alpha_n$ are constructed by the conjugation of the residues $r_1, \cdots, r_n$ respectively. Then in the kth iteration step, p_k takes exactly one step of the length h_k along the direction α_k. If the convergence conditions are met $err < \varepsilon$ or $k > k_{max}$, the iterative process is terminated and we get the final pressure p. After solving the linear system and obtaining the unknown p, we compute fluid velocity according to Eqn. (5.2).

5.2.4. Communication

To keep the computation correctly, the block on adjacent client should be overlapped. As shown in Fig. 3, yellow block and green block is computational domain on each client while the orange block receives information from the adjacent client. As we can see that the green block and orange block are the overlap blocks between the adjacent clients. In Fig. 3, the green block of client 1 is overlapped with the orange block of the client 2 and vice versa. They send and receive information from the adjacent regions to keep the continuity of the solution across subdomain interface. In addition, the indices of fluid cells are also required to communicate globally. To do this, the clients send the numbers of fluid cells to their neighbors sequentially.

6. Results

We have implemented our parallel algorithm in a cluster system which includes 17 computers. Each PC in the cluster has 2.6 GHz Intel Core 2 Duo CPU and 2GB memory. In this cluster system, we set one of the 17 computers as the server and 16 of them as clients. The performance of our parallel fluid animation algorithm can be evaluated by the high resolution smoke and liquid simulation examples which will be illustrated in the following parts.

6.1. Performance analysis

During fluid equation computation, one of the most time consuming parts is to solve the large sparse linear system for pressure p. We compare the convergence of our method with Jacobi iterative method in terms of rate per iteration. Tab. 1 shows the matrix A of $Ap = b$ used in this experiments. Since our block diagonal preconditioner is formed from the incomplete Cholesky blocks on each client, no communication is needed in this process. Furthermore, the number of iteration steps is also reduced by using the pre-conditioner. The final costs are given in Tab. 2. Observe that the performance of our method is more efficient than the traditional Jacobi iterative method.

Tab. 2 shows that there is significant performance increase for our PCG algorithm when solving the sparse linear system compared with Jacobi method. For example, given a matrix whose row number is $N = 304207$ and the non-zero elements is 1962311, our parallel PCG algorithm takes 35 iteration steps and consumes about 49 seconds for obtaining the unknowns. When solving the same linear system, parallel Jacobi methods takes about 80 seconds. We increase to 38% performance improvement in this example.

As shown in Fig. 4 (a), the fluid animation cost may fluctuate with different frames. That is because that different boundary conditions and initial conditions can result

Table 1. Matrix used for experiments

Matrix	N	Nonzero
M1	8087	49915
M2	22028	131118
M3	65043	394877
M4	140120	876518
M5	209908	1325066
M6	274949	1755263
M7	304207	1962311

Table 2. Matrix used for experiments.

Matrix	Jacobi steps	Timings for Jacobi method	PCG steps	Timings for PCG method
M1	84	4.46	17	3.54
M2	90	5.35	21	8.17
M3	123	9.92	22	18.56
M4	146	16.86	25	31.74
M5	192	55.26	30	33.13
M6	221	77.45	34	44.64
M7	234	79.63	35	48.94

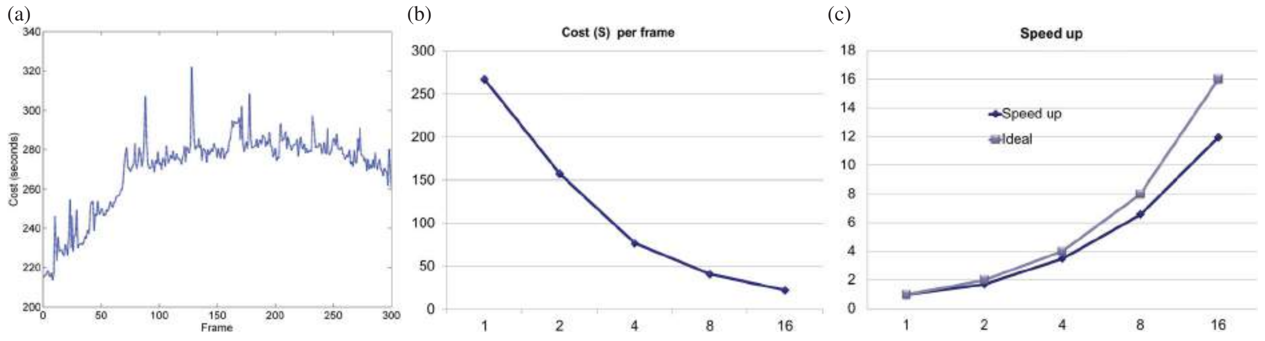

Figure 4. Efficiency of our parallel algorithm. (a): computational cost (seconds) of each frame. Note that the efficiency of fluid animation fluctuates due to many factors. (b): average cost (seconds) of each frame. Horizontal axis represents the number of PCs. The cost decreases significantly when the number of clients increases. (c): The speed up of our parallel algorithm. It is close to the ideal value because it is a computation intensive task in which the communication cost is quite small.

in quite different computational cost. Firstly, the computational times are varying with the number of fluid cells. Secondly, the condition number of the matrix for the sparse linear system can also lead to different cost. If the linear system in ill-conditioned, it will take more iteration steps to get a convergence value.

We also demonstrate the average computational cost of each frame and the speedup of our parallel algorithm (shown in Fig. 4 (b) and (c)). In this example, the liquid source is added to a 3D valley and we divide the computational task into different number of subtasks. Results show that the computational cost of each frame decreases significantly with the number of processors in the cluster system. We also show the speed up (see Fig. 4 (c)) of our parallel algorithm in this paper. It is very close to the ideal speed up value because our task is a computation intensive one in which the communication cost is quite small.

6.2. Animation results

Fig. 5 shows the animation results of the liquid valley example in this paper. We decompose the computational domain into 16 blocks and uses 16 clients to compute it. The efficiency of this example is illustrated in Fig. 4. The animation results are output from the server. It gathers numerical data from the clients in every time step and abstracts the water surface from the data. The global rendering method are used to get the final animation images on the server.

Our method also supports smoke animation. In Fig. 6, we demonstrate the smoke propagation in a building. The source of the smoke is set at a room in the center of the building. To compute the animation effectively, we decompose the domain into $8 \times 1 \times 2$ blocks. The blocks are assigned to 16 clients for numerical computation. In addition, the speed up is also close to 16 since the

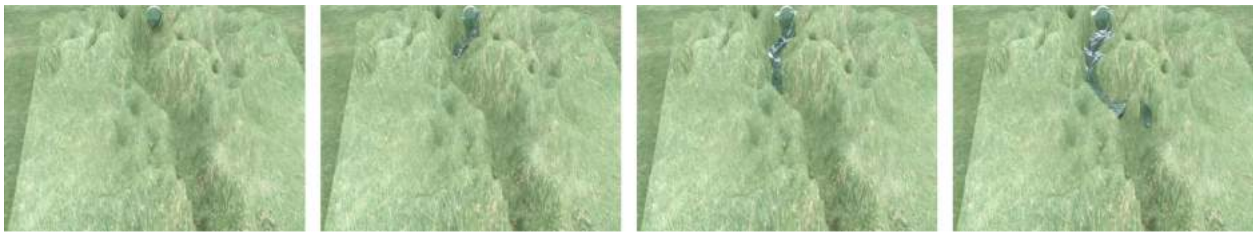

Figure 5. A sequence of liquid animation results in a valley.

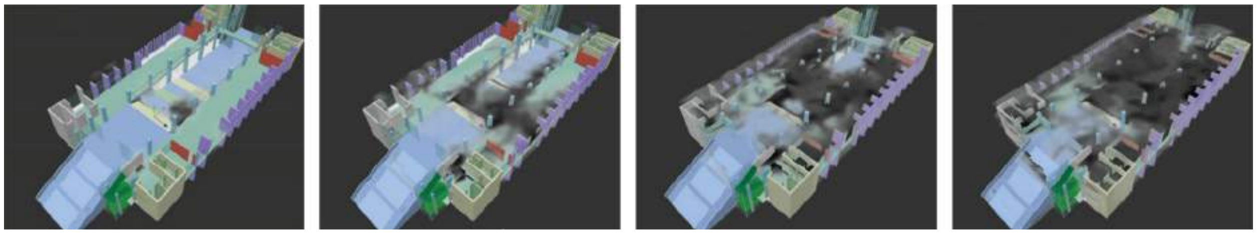

Figure 6. A sequence of smoke animation results in a building.

communication cost is small compared to the computation cost. The server collects density data and renders the data directly rather than abstracts surface mesh in liquid animation.

7. Conclusions

We have implemented a parallel method for fluid animation on cluster system. Our method can solve the fluid equation in parallel with high efficiency. To get interesting fluid details, we solve physically-based liquid animation model with the popular Eulerian scheme. During simulation, we control fluid animation on server while compute the numerical solution on clients. Results show that the proposed parallel fluid acceleration method can improve the efficiency of fluid animation significantly.

However, this paper only implements the most time-consuming step of fluid animation on GPU. In fact, physically-based fluid animation method is a very complex model that includes many equations and algorithms such as the Level Set equation, SPH model, Fast Marching algorithm and Marching Cubes et al. As for the future work, we would like to extend the above algorithms on GPU to further improve the efficiency. In addition, optimizing the method presented in this paper according to the hardware properties of GPU card is another meaningful work.

Acknowledgement

This work is supported the National Nature Science Foundation of China (Nos. 61202225, 61303157, 61303007, 61402270, 61572299), Shandong Provincial Natural Science Foundation, China (ZR2014FQ009, ZR2015FQ009), Research Fund for Excellent Young and Middle-aged Scientists of Shandong Province (BS2013DX044), Shandong Province Higher Educational Science and Technology Program (J13LN13).

ORCID

Guijuan Zhang http://orcid.org/0000-0002-9545-8668
Jinyan Zhao http://orcid.org/0000-0001-9836-0642
Weizhi Xu http://orcid.org/0000-0001-7549-4138
Dianjie Lu http://orcid.org/0000-0001-5435-5307
Yongjian Wang http://orcid.org/0000-0003-2351-3186
Xiangxu Meng http://orcid.org/0000-0001-7290-5659

References

[1] Bayraktar, S.; Gdkbay, U.; Ozgc B.: GPU-based neighbor-search algorithm for particle simulations, Journal Graphics GPU Game Tool, 14(1), 2009, 31–42.
[2] Crane, K.; Llamas, I.; Tariq S.: Real-time simulation and rendering of 3D fluids, GPU Gem 3, Chaper 30, 2007.
[3] Goswami, P.; Schlegel, P.; Solenthaler, B.; Pajarola R.: Interactive SPH simulation and rendering on the GPU. In Proceedings of the ACMSIGGRAPH/Eurographics Symposium on Computer Animation 2010, 55–64.
[4] He, X.; Wang, H.; Zhang, F. et al: Robust simulation of sparsely sampled thin features in SPH-based free surface flows, ACM Transactions on Graphics, 34(1), 2014, 1–9.
[5] Hu, C.; Xu, Z.; et al. Semantic Link Network based Model for Organizing Multimedia Big Data, IEEE Transactions on Emerging Topics in Computing, 2(3), 2014, 376–387.
[6] Ihmsen, M.; Akinci, N.; Becker, M.; Teschner M.: A parallel SPH implementation on multi-core CPUs, Computer Graphics Forum, 30(1), 2011, 99–112.
[7] Kipfer, P.; Westermann, R.: Realistic and interactive simulation of rivers, In Proceedings of Graphics Interface 2006, 41–48.
[8] Knyazev, A. V.; Lashuk, I.: Steepest descent and conjugate gradient methods with variable preconditioning, SIAM J. Matrix Analysis and Applications, 29(4), 2007, 1267–1280.
[9] Kurose, S.; Takahashi, S.: Constraint-based simulation of interactions between fluids and unconstrained rigid bodies, In Proceedings of Spring Conference on Computer Graphics 2009, 197–204.
[10] Stam, J.: Stable Fluids, In proceedings of SIGGRAPH 99, 1999, 121–128.
[11] Xu, Z.; et al. Semantic based representing and organizing surveillance big data using video structural description technology, The Journal of Systems and Software, 102, 2015, 17–225.
[12] Xu, Z.; et al. Semantic Enhanced Cloud Environment for Surveillance Data Management using Video Structural Description, Computing, 98(1–2), 2016, 35–54.
[13] Xu, W.; Yu, H.; Lu, D.; et al: Fast and scalable lock methods for video coding on many-core architecture, Journal of Visual Communication and Image Representation, 25(7), 2014, 1758–1762.
[14] Zhang, G.; Lu, D.; Zhu, D. et al: Rigid-motion-inspired liquid character animation, Computer Animation and Virtual Worlds, 24(3–4), 2013, 205–213.
[15] Zhang, Y.; Solenthaler, B.; Pajarola R.: Adaptive sampling and rendering of fluids on the GPU, In Proceedings of the Fifth Eurographics / IEEE VGTC conference on Point-Based Graphics 2008, 137–146.
[16] Zhang, G.; Zhu, D.; Qiu, X. et al: A Scene Processing Method for Fluid Simulation, Journal of Computer-Aided Design & Computer Graphics, 22(8), 2010, 1360–1365.