Numerical simulation of tsunamis generated by landslides on multiple GPUs

M. de la Asunción[a,*], M. J. Castro[a], J. M. Mantas[b], S. Ortega[a]

[a] *Dpto. Análisis Matemático, Universidad de Málaga, Spain*
[b] *Dpto. Lenguajes y Sistemas Informáticos, Universidad de Granada, Spain*

Abstract

In this work we propose a two-layer Savage-Hutter type model that is the natural extension of the 1D system proposed by E. D. Fernández-Nieto et al in 2008 to simulate tsunamis generated by landslides. We describe a single GPU and a multi-GPU implementation of this model using MPI and the CUDA framework over structured meshes. The distributed implementation is tested for several artificial and realistic problems using up to 24 GPUs. We also propose a static and a dynamic load balancing algorithm in order to deal with the unbalanced computational load due to different amount of wet and dry areas among the subdomains. The validity of the model is tested by simulating the tsunami occurred in Lituya Bay, Alaska, in 1958. Numerical experiments show the efficiency of the multi-GPU solver, the usefulness of the load balancing algorithms and the validity of the model to simulate real tsunamis generated by landslides.

Keywords: Tsunamis generated by landslides, Finite volume schemes, Structured meshes, CUDA, MPI, Load balancing

1. Introduction

Giant landslides, underwater or not, are one of the main causes of tsunami generation and can have a dramatic impact on life, property and infrastructures [1].

*Corresponding author. Tel.: +34 952 131898; fax: +34 952 131894.
Email address: marcah@uma.es (M. de la Asunción)

In [2] a two-layer Savage-Hutter 1D model that can be used to simulate tsunamis generated by underwater landslides was presented. Here the natural extension of this model to 2D domains is considered to simulate landslides and the generated tsunami on realistic bathymetries. The application of this model requires a great computational demand due to the big sizes of the spatial and temporal domain. For this reason, very efficient parallel solvers are required to perform these simulations in real domains in order to obtain results in reasonable times.

In recent years, the Graphics Processing Units (GPUs) have proved to be a powerful accelerator for intensive scientific simulations. The high memory bandwidth and massive parallelism of these platforms make it possible to achieve dramatic speedups over a standard CPU in many applications [3, 4], and several programming toolkits and interfaces, such as NVIDIA CUDA [5] and Open Computing Language (OpenCL) [6], have shown a high effectiveness in the mapping of data parallel applications to GPUs [3, 7].

Currently most of the proposals to simulate shallow flows on a single GPU are based on the CUDA programming model. There are several proposals of finite volume CUDA solvers to simulate one-layer shallow water flows over structured regular meshes [8, 9] and for the two-layer shallow water system [10, 11]. In this work, we are interested in the efficient acceleration, using GPU-enabled hardware, of a sophisticated first order numerical model to simulate realistic tsunamis generated by landslides.

Realistic tsunami simulations involve huge meshes, many time steps and possibly real time accurate predictions. These characteristics suggest to use a cluster of GPU-enhanced computers (hereafter, a GPU cluster) in order to scale the runtime reduction and overcome the memory limitations of a GPU-enhanced node by suitably distributing the data among the nodes, enabling us to simulate significantly larger realistic models.

Most of the proposals to exploit GPU clusters in computational fluid dynamics (CFD) simulations use CUDA to program each GPU, and MPI [12] to implement interprocess communication, and they use non-blocking communication MPI functions to overlap the remote transfers with GPU computation [13, 14, 15, 16, 17, 18].

There are several studies related with the multi-GPU solution of shallow water systems. A one-layer shallow water solver is implemented on a GPU cluster for tsunami simulation in [13], achieving good performance results in the range 1-32 GPUs. The multi-GPU implementation of a Lattice-Boltzmann solver of the one-layer shallow water system is tackled in [19] for a small GPU cluster. In [20], the performance of a MPI-CUDA finite volume solver of one-layer shallow flows is evaluated on a GPU cluster with 32 nodes. In [16], an efficient MPI-CUDA implementation of a structured mesh finite volume solver for the one-layer shallow water system coupled with a pollutant transport equation is described, and the impact of using the ghost cell expansion technique [21] in InfiniBand GPU clusters is analyzed.

An important issue with the distributed simulation of these kind of problems arises when there is a high variation in the amount of wet and dry zones among the subdomains. Since a wet zone requires more numerical processing than a dry zone, special attention should be paid so that the computational load of all the subdomains is as equal as possible. Therefore, load balancing methods are required in order to overcome this issue. It have been published several works which analyze the influence of load balancing algorithms in the obtained runtimes on shallow water problems. See, for example, [22, 23, 24]. A review of some partitioning and load balancing methods for the numerical resolution of partial differential equations can be found in [25].

In this work we propose a state-of-the-art finite volume numerical scheme to simulate tsunamis generated by landslides. We have developed an efficient MPI-CUDA implementation of this numerical scheme over structured meshes for GPU clusters which follows a row-block domain decomposition and incorporates several techniques to improve the efficiency, such as the overlapping of computation and communication, and load balancing algorithms. Specifically, we have developed a static (SLB) and a dynamic (DLB) load balancing algorithm to deal with the variations of wet and dry zones among the submeshes. We show that it is possible to obtain scalable multi-GPU implementations of the model which works on big realistic domains. The influence on the scalability (weak and strong scalability) of the load balancing algorithms has been studied using up to 24 GPUs by performing several experiments with artificial and real simulation problems on an InfiniBand GPU cluster.

The numerical scheme described in this paper is also validated by simulating the tsunami occurred on July 9, 1958, in Lituya Bay, Alaska. This tsunami was caused by a subaerial landslide and it reached the highest run-up ever measured for a tsunami: 524 meters. A comparison of real and model data is presented.

The rest of this paper is organized as follows. Section 2 introduces the two-layer Savage-Hutter type system used to simulate tsunamis generated by landslides and Section 3 describes the finite volume numerical scheme proposed to solve the equations of this model. Section 4 introduces the SLB and DLB algorithms. Section 5 describes the main details of the CUDA implementation of this solver and its extension for GPU clusters using MPI and CUDA. The efficiency of the developed implementations is analyzed in Section 6 by using artificial and realistic problems, and a validation of the model is performed by simulating the Lituya Bay tsunami. Finally, in Section 7 we collect the main conclusions of the work.

2. A two-layer Savage-Hutter type model for simulating landslides generated tsunamis

In [2] a new model for the simulation of tsunamis generated by submarine landslides was presented for 1D geometries. In this work, we consider the natural extension for 2D domains, so that real problems, like those considered in Section 6, could be simulated.

We consider a stratified media composed by a non viscous and homogeneous fluid with constant density ρ_1 (water) and a fluidized granular material with density ρ_s and porosity ψ_0. We suppose that the fluid and the granular material are immiscible and that the mean density of the granular material is given by: $\rho_2 = (1-\psi_0)\rho_s + \psi_0\rho_1$. The following 2D system is derived under the assumption of shallow-flows and could be used to simulate the interaction of a granular landslide with the ambient water (see [2] for details about its derivation in 1D problems):

$$\begin{aligned}\frac{\partial W}{\partial t} + \frac{\partial F_1}{\partial x}(W) + \frac{\partial F_2}{\partial y}(W) = B_1(W)\frac{\partial W}{\partial x} + B_2(W)\frac{\partial W}{\partial y} \\ + \; S_1(W)\frac{\partial H}{\partial x} + S_2(W)\frac{\partial H}{\partial y} + S_F(W) \qquad (1)\end{aligned}$$

being

$$W = \begin{pmatrix} h_1 \\ q_{1,x} \\ q_{1,y} \\ h_2 \\ q_{2,x} \\ q_{2,y} \end{pmatrix}, \quad F_1(W) = \begin{pmatrix} q_{1,x} \\ \dfrac{q_{1,x}^2}{h_1} + \dfrac{1}{2} g h_1^2 \\ \dfrac{q_{1,x} q_{1,y}}{h_1} \\ q_{2,x} \\ \dfrac{q_{2,x}^2}{h_2} + \dfrac{1}{2} g h_2^2 \\ \dfrac{q_{2,x} q_{2,y}}{h_2} \end{pmatrix}, \quad F_2(W) = \begin{pmatrix} q_{1,y} \\ \dfrac{q_{1,x} q_{1,y}}{h_1} \\ \dfrac{q_{1,y}^2}{h_1} + \dfrac{1}{2} g h_1^2 \\ q_{2,y} \\ \dfrac{q_{2,x} q_{2,y}}{h_2} \\ \dfrac{q_{2,y}^2}{h_2} + \dfrac{1}{2} g h_2^2 \end{pmatrix},$$

$$B_1(W) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -gh_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -rgh_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad S_1(W) = \begin{pmatrix} 0 \\ gh_1 \\ 0 \\ 0 \\ gh_2 \\ 0 \end{pmatrix},$$

$$B_2(W) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -gh_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -rgh_2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad S_2(W) = \begin{pmatrix} 0 \\ 0 \\ gh_1 \\ 0 \\ 0 \\ gh_2 \end{pmatrix},$$

$$S_F(W) = \begin{pmatrix} 0 & S_{f_1}(W) & S_{f_2}(W) & 0 & S_{f_3}(W) + \tau_x & S_{f_4}(W) + \tau_y \end{pmatrix}^T.$$

In the previous system, $h_l(x, y, t)$, $l = 1, 2$ denotes the thickness of the water layer ($l = 1$) and the granular material ($l = 2$), respectively, at point $(x, y) \in D \subset \mathbb{R}^2$ at time t, being D the horizontal projection of the 3D domain where the landslide and tsunami take place. $H(x, y)$ is the depth

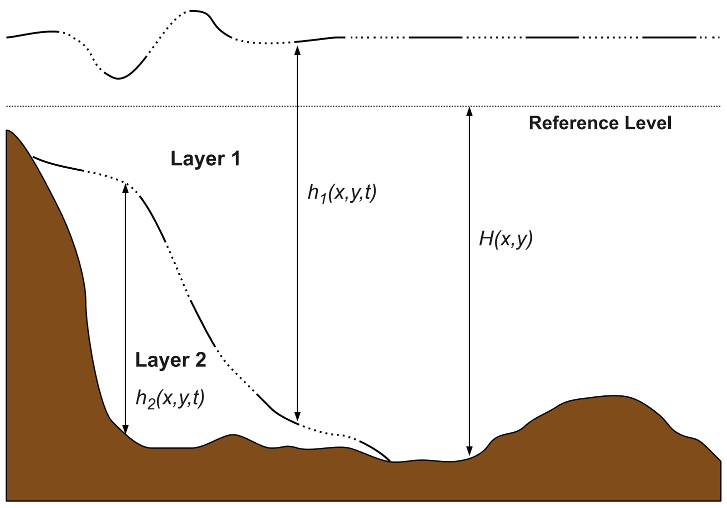


Figure 1: Scheme of the model. Relation between h_1, h_2 and H.

of the bottom at point (x, y) measured from a fixed level of reference. Let us also define the function $\eta_1(x, t, t) = h_1(x, y, t) + h_2(x, y, t) - H(x, y)$ that corresponds to the free surface of the fluid, and $\eta_2(x, y, t) = h_2(x, y, t) - H(x, y)$, the interface between the granular layer and the fluid. Let us denote by $\boldsymbol{q}_l(x, y, t) = (q_{l,x}(x, y, t),\, q_{l,y}(x, y, t))$ the mass-flow of the l-layer at point (x, y) at time t. The mass-flow is related to the height-averaged velocity $\boldsymbol{u}_l(x, y, t)$ by means of the expression: $\boldsymbol{q}_l(x, y, t) = h_l(x, y, t)\, \boldsymbol{u}_l(x, y, t)$, $l = 1, 2$. $r = \rho_1/\rho_2$ is the ratio of the constant densities of the layers ($\rho_1 < \rho_2$). Figure 1 shows graphically the relation between h_1, h_2 and H.

The terms $S_{f_k}(W)$, $k = 1, \ldots, 4$, model the different friction effects, while $\boldsymbol{\tau} = (\tau_x,\, \tau_y)$ is the Coulomb friction law. $S_{f_k}(W)$, $k = 1, \ldots, 4$, are given by:

$$\begin{aligned} S_{f_1}(W) &= S_{c_x}(W) + S_{1_x}(W) \qquad & S_{f_3}(W) &= -r\, S_{c_x}(W) + S_{2_x}(W) \\ S_{f_2}(W) &= S_{c_y}(W) + S_{1_y}(W) \qquad & S_{f_4}(W) &= -r\, S_{c_y}(W) + S_{2_y}(W) \end{aligned}$$

$S_c(W) = \big(S_{c_x}(W),\, S_{c_y}(W)\big)$ parameterizes the friction between the two layers, and is defined as:

$$\begin{cases} S_{c_x}(W) = m_f \dfrac{h_1 h_2}{h_2 + r h_1} (u_{2,x} - u_{1,x})\, \|\boldsymbol{u}_2 - \boldsymbol{u}_1\| \\[2ex] S_{c_y}(W) = m_f \dfrac{h_1 h_2}{h_2 + r h_1} (u_{2,y} - u_{1,y})\, \|\boldsymbol{u}_2 - \boldsymbol{u}_1\| \end{cases}$$

where m_f is a positive constant.

$S_l(W) = \big(S_{l_x}(W),\, S_{l_y}(W)\big)$, $l = 1,2$ parameterizes the friction between the fluid and the non-erodible bottom ($l = 1$) and between the granular material and the non-erodible bottom ($l = 2$), and both are given by a Manning law

$$\left\{\begin{array}{l} S_{l_x}(W) = -gh_l \, \dfrac{n_l^2}{h_l^{4/3}} \, u_{l,x} \, \|\boldsymbol{u}_l\| \\[2ex] S_{l_y}(W) = -gh_l \, \dfrac{n_l^2}{h_l^{4/3}} \, u_{l,y} \, \|\boldsymbol{u}_l\| \end{array}\right. , \quad l = 1,2$$

where $n_l > 0$ ($l = 1,2$) is the Manning coefficient. Note that $S_1(W)$ is only defined where $h_2(x,y,t) = 0$. In this case, $m_f = 0$ and $n_2 = 0$. Similarly, if $h_1(x,y,t) = 0$ then $m_f = 0$ and $n_1 = 0$.

Finally, the Coulomb friction term $\boldsymbol{\tau} = (\tau_x,\, \tau_y)$ controls the stopping mechanism of the landslide and it is defined as follows:

$$\text{If } \|\boldsymbol{\tau}\| \geq \sigma^c \Rightarrow \left\{\begin{array}{l} \tau_x = -g(1-r)h_2 \, \dfrac{q_{2,x}}{\|\boldsymbol{q}_2\|} \, \tan(\alpha) \\[2ex] \tau_y = -g(1-r)h_2 \, \dfrac{q_{2,y}}{\|\boldsymbol{q}_2\|} \, \tan(\alpha) \end{array}\right.$$

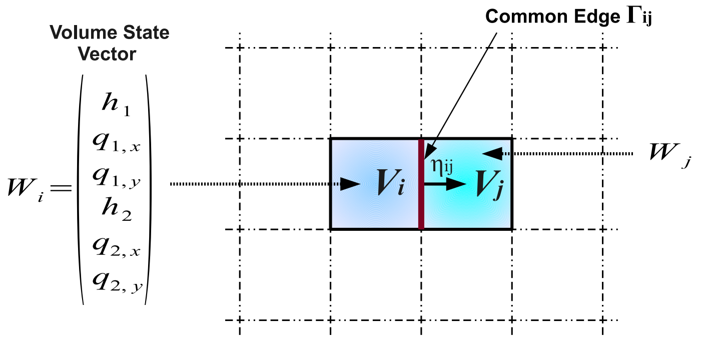
$$\text{If } \|\boldsymbol{\tau}\| < \sigma^c \Rightarrow q_{2,x} = 0, \quad q_{2,y} = 0$$

where $\sigma^c = g(1-r)h_2 \tan(\alpha)$, being α the Coulomb friction angle. Let us remark that r is set to zero in σ^c and $\boldsymbol{\tau}$, if $h_1(x,y,t) = 0$, that is, if it is an aerial landslide.

Note that the previous model reduces to the usual one-layer shallow-water system if $h_2 = 0$ and to the Savage-Hutter model if $h_1 = 0$.

3. Numerical Scheme

In this section, we present the finite volume method that we use to discretize the system (1). To discretize system (1), the domain D is divided into L cells or finite volumes $V_i \subset \mathbb{R}^2$, $i = 1, \ldots, L$, which we assume they are squares or rectangles with edges parallel to the cartesian axes. Given a finite volume V_i, $N_i \subset \mathbb{R}^2$ is the center of V_i, $\aleph_i$ is the set of indices j such that V_j is a neighbour of V_i; Γ_{ij} is the common edge of two neighbouring volumes V_i



Volume State Vector
Common Edge Γ_{ij}
$W_i = (h_1, q_{1,x}, q_{1,y}, h_2, q_{2,x}, q_{2,y})$
V_i
η_{ij}
V_j
W_j

Figure 2: Finite volumes.

and V_j, and $|\Gamma_{ij}|$ is its length; $\boldsymbol{\eta}_{ij} = (\eta_{ij,x},\, \eta_{ij,y})$ is the unit vector which is normal to the edge Γ_{ij} and points towards the volume V_j (see Figure 2).

We denote by W_i^n an approximation of the average of the solution in the volume V_i at time t^n:

$$W_i^n \cong \frac{1}{|V_i|} \int_{V_i} W\big(x, y, t^n\big)\, dx\, dy$$

where $|V_i|$ is the area of the volume and $t^n = t^{n-1} + \Delta t$, being Δt the time step.

The source terms corresponding to friction terms are discretized semi-implicitly following the ideas described in [2]. Thus, in a first step, friction terms are neglected and only physical flux, source and nonconservative terms are considered. In this first step, a 2D extension of IFCP scheme (see [26]) is used to approximate the non-conservative and source terms. Here, only the first step is detailed. The discretization of the friction terms are performed following [2].

In order to extend to 2D problems, the 1D IFCP Riemann solver, a family of projected Riemman problems in the normal direction to each edge of the mesh is considered. These projected Riemann problems can be easily defined as system (1) verifies the property of invariance by rotations (see [17] for more details). Thus, if we define

$$T_{\boldsymbol{\eta}_{ij}} = \begin{pmatrix} R_{\boldsymbol{\eta}_{ij}} & \mathbf{0} \\ \mathbf{0} & R_{\boldsymbol{\eta}_{ij}} \end{pmatrix}, \quad R_{\boldsymbol{\eta}_{ij}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \eta_{ij,x} & \eta_{ij,y} \\ 0 & -\eta_{ij,y} & \eta_{ij,x} \end{pmatrix}$$

the 2D extension of the IFCP scheme could be written as follows:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|} \sum_{j \in \aleph_i} |\Gamma_{ij}| \, F_{ij}^-(W_i^n, W_j^n, H_i, H_j) \tag{2}$$

where

$$F_{ij}^{\pm}(W_i^n, W_j^n, H_i, H_j) = T_{\boldsymbol{\eta}_{ij}}^{-1} G_{ij}^{\pm} \tag{3}$$

with

$$G_{ij}^{\pm} = \left[\left(\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}\right)_{[1]} \quad \left(\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}\right)_{[2]} \quad \left(\Phi_{\boldsymbol{\eta}_{ij}^{\perp}}^{\pm}\right)_{[1]} \quad \left(\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}\right)_{[3]} \quad \left(\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}\right)_{[4]} \quad \left(\Phi_{\boldsymbol{\eta}_{ij}^{\perp}}^{\pm}\right)_{[2]} \right]^T .$$

Here $\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}$ and $\Phi_{\boldsymbol{\eta}_{ij}^{\perp}}^{\pm}$ are 1D numerical fluxes corresponding to the 1D projected Riemann problem associated to edge Γ_{ij} (see [17]) and $\left(\Phi_{\boldsymbol{\eta}_{ij}}\right)_{[l]}$ is the component l of vector $\Phi_{\boldsymbol{\eta}_{ij}}$ (similar notation is used with vector $\Phi_{\boldsymbol{\eta}_{ij}^{\perp}}^{\pm}$). The description of the 1D numerical fluxes $\Phi_{\boldsymbol{\eta}_{ij}}^{\pm}$ and $\Phi_{\boldsymbol{\eta}_{ij}^{\perp}}^{\pm}$ is given in Appendix A.

The resulting numerical scheme is explicit, therefore, it is necessary to impose a CFL (Courant-Friedrichs-Lewy) condition to ensure linear stability of the scheme. In practice, this condition implies a restriction on the time step given by:

$$\Delta t^n = \min_{i=1,\dots,L} \left(\frac{\sum_{j \in \aleph_i} |\Gamma_{ij}| \, \max |\lambda_{ij}|}{2\gamma |V_i|} \right)^{-1} \tag{4}$$

where $0 < \gamma \leq 1$ and λ_{ij} are the eigenvalues of matrix $\mathcal{A}_{ij}$ (see Appendix A).

In order to deal with wet-dry transitions, we have used the modification of the numerical scheme described in [27] for the one dimensional case and we have extended it to 2D.

The resulting scheme is exactly well-balanced for the stationary water at rest solution ($\boldsymbol{q}_1 = \boldsymbol{q}_2 = \boldsymbol{0}$ and μ_1 and μ_2 constant), that is, it solves exactly this stationary solution. Moreover, the scheme is able to approximate accurately the stationary solutions corresponding to $\boldsymbol{q}_1 = \boldsymbol{q}_2 = \boldsymbol{0}$, μ_1 constant and $\partial_x \mu_2 < \tan(\alpha)$ and $\partial_y \mu_2 < \tan(\alpha)$, that is an stationary water at rest solution for which the Coulomb friction term balances the pressure term in the granular material. In fact, this last property is critical to simulate landslides accurately.

4. Load Balancing Algorithms

The row-block domain decomposition is the partitioning method which we have used in this work because of its simplicity and its straightforward implementation. However, as stated in Section 1, if there is a high variation in the amount of wet and dry zones among the subdomains, it may affect the obtained efficiency due to the unbalanced computational load among the different subdomains. Next, we describe two load balancing algorithms which we have developed to overcome this issue.

4.1. Static Load Balancing

The SLB algorithm may be useful in simulations with little variations of wet and dry zones during the simulation, since the domain partition is performed only once in a preprocessing step.

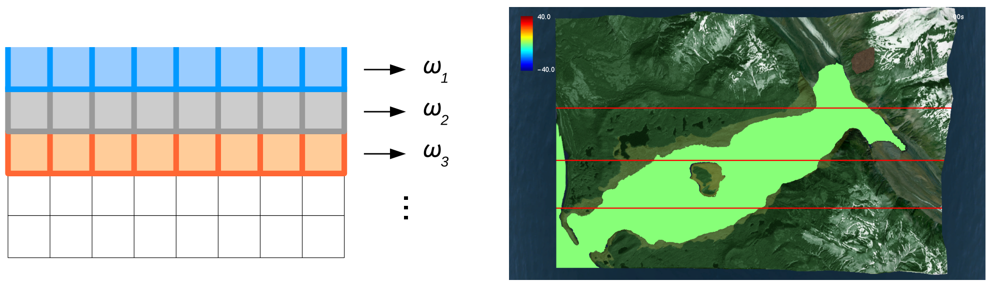
In this algorithm, firstly we assign four weights corresponding to a wet or dry edge, and a wet or dry cell. An edge (similarly for cells) is wet if $h_1 > 0$ or $h_2 > 0$ for at least one of its adjacent volumes. The values of these weights depend on the graphics card, and they are independent of the simulated problem. These weights are obtained experimentally based on the computational demand of the processing of the edges and volumes. Specifically, we run a particular test problem with a fixed time step, and we measure the runtimes of the processing of the edges and volumes. Next, we set both layers to zero and repeated the same procedure. Finally, the relative weights of the processing of edges and volumes are assigned.

Once the previous weights are determined, using the initial state we get the total weight ω_i for each row i of the spatial mesh, where ω_i is computed by adding the weights of all the vertical edges, the volumes and the lower horizontal edges of the row i (see Figure 3a). Finally, the domain decomposition is performed by adding consecutive rows to a particular submesh until the ideal weight of a submesh is reached.

Note that, if the submeshes are big enough, the influence of the upper horizontal edges of the first row of every submesh, which are not taken into account, is negligible. Figure 3b shows an example of application of the SLB algorithm to a spatial domain, where it can be seen that the upper submesh is bigger than the others because it has a very small wet area.

4.2. Dynamic Load Balancing

The DLB algorithm is an extension of the SLB algorithm in order to be useful in problems with high variations of wet and dry zones during the sim-



(a) (b)

Figure 3: SLB algorithm: (a) Computation of weight ω_i for each row. (b) Example of application of the SLB algorithm to a spatial domain.

ulation. Basically, in this case we check a balancing condition every certain number of iterations and, if this condition fulfills, we perform a redistribution of the rows among the submeshes in the same way as in the SLB algorithm. We define the balancing condition as the existence of at least one submesh s that satisfies:

$$\left| 1 - \frac{\Omega_s}{\Omega_I} \right| > d \tag{5}$$

being Ω_s the weight of the submesh s, Ω_I its ideal weight, and d a given threshold. That is, if the deviation of the weight of a submesh with respect to its ideal weight is greater than d, a redistribution of the rows among the submeshes is performed. Section 5.2 gives some details about the implementation of the load balancing algorithms.

The number of iterations between two consecutive checks of the balancing condition and the value of the d threshold should be chosen carefully, and their appropriate values depend on the features of the particular problem. For example, if the dry areas become flooded in a fast way, you may want to check the balancing condition more often than if the inundation advanced slowly.

5. Implementation Details

5.1. One GPU Implementation

In this section we describe the most important aspects of the implementation of the numerical scheme presented in Section 3 that we have carried

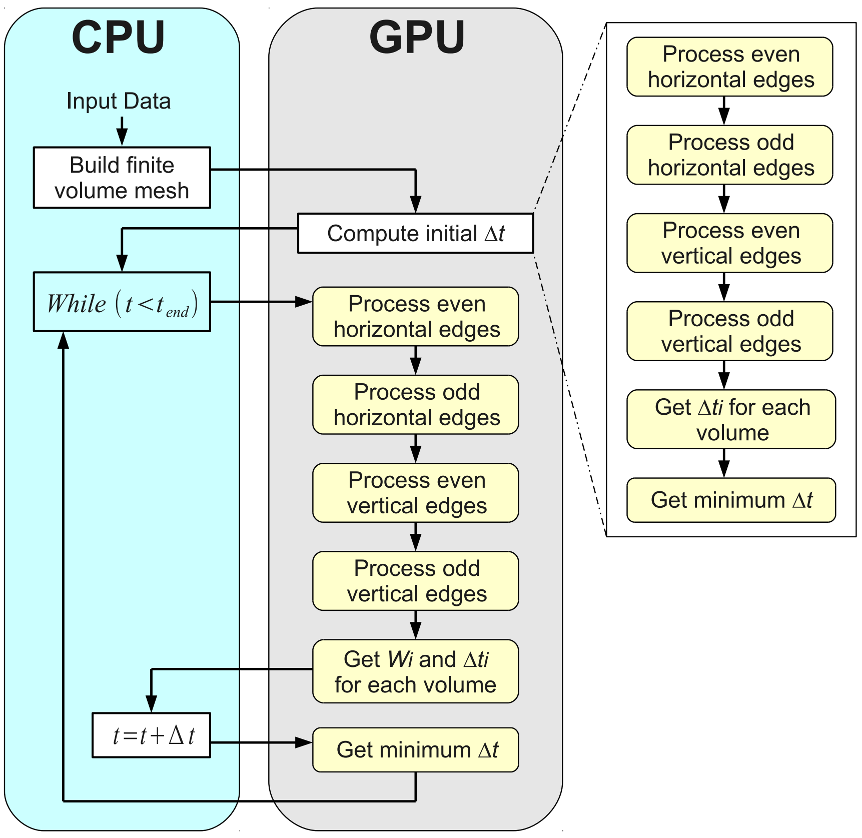

CPU
GPU
Input Data
Build finite volume mesh
Compute initial Δt
While (t<t_end)
Process even horizontal edges
Process odd horizontal edges
Process even vertical edges
Process odd vertical edges
Get Wi and Δti for each volume
t=t+Δt
Get minimum Δt
Process even horizontal edges
Process odd horizontal edges
Process even vertical edges
Process odd vertical edges
Get Δti for each volume
Get minimum Δt

Figure 4: Parallel CUDA implementation.

out using the CUDA framework. The general steps of the parallel implementation are shown in Figure 4. Each step executed on the GPU is assigned to a CUDA kernel. Next we describe each step of the algorithm:

- **Build finite volume mesh**: In this step the main data structure which will be used in GPU is built. For each volume V_i we store its state and depth H in two arrays of `float4` elements, where the size of both arrays is the number of volumes. The first array contains h_1, $q_{1,x}$, $q_{1,y}$ and H, whereas the second array contains h_2, $q_{2,x}$ and $q_{2,y}$. Since a structured mesh has a regular spatial pattern, both arrays can be accessed in a coalesced way and, therefore, they are stored in global memory [28]. Figure 5 depicts the format of these arrays.

  The area of the volumes and the length of the edges are precalculated

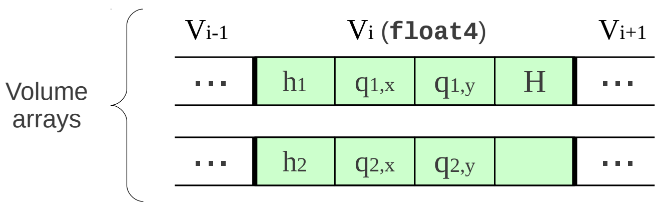


Figure 5: Main data structure used in GPU.

and passed directly to the CUDA kernels that need them.

We can know at runtime if an edge or volume is located at the frontier of the spatial domain and which is the value of the normal $\boldsymbol{\eta}_{ij}$ of an edge by checking the position of the thread in the grid.

- **Process edges**: In this step, each thread represents an edge and computes the contribution of the edge to its adjacent volumes. If the two adjacent volumes of the edge do not contain any layer ($h_1 \simeq 0$ and $h_2 \simeq 0$), the thread finishes without processing the edge. In this implementation we follow a similar approach to that of we applied in [10] and [8], where the edge processing was divided into horizontal and vertical edge processing, allowing some terms of the numerical calculation to be removed and thus improving the efficiency.

  The edges synchronize each other when contributing to a particular volume by means of two arrays (accumulators) of `float4` elements in global memory. The size of each accumulator is the number of volumes. The meaning and functionality of these accumulators is similar to that of used in [10]. Each element of the first accumulator stores the contribution of the edges to the layer 1 of W_i (a 3×1 vector) and to the local Δt of the volume (a `float` value), whereas each element of the second accumulator stores the contributions of the edges to the layer 2 of W_i (a 3×1 vector).

  Note that the division of the edge processing into four kernels (processing of even horizontal edges, odd horizontal edges, even vertical edges and odd vertical edges) has allowed us to use only two accumulators instead of the four accumulators which were used in [10], thus reducing the memory requirements.

- **Get W_i^{n+1} and Δt_i for each volume**: In this step, each thread represents a volume and obtains the next state W_i^{n+1} and the local Δt_i of the volume V_i by using the values stored in the position associated with the volume V_i of the two accumulators.

- **Get minimum** Δt: In this step the minimum of all the local Δt_i values of the volumes is obtained by applying a reduction algorithm in GPU, in the same way as we did in [10] and [8].

We assign sizes of 48 KB and 16 KB to the L1 cache and to the shared memory of the GPU, respectively, for all the edge processing kernels. Thread block size is $8 \times 8 = 64$ threads for the edge processing kernels and the kernel which obtains the next state W_i^{n+1}.

5.2. Multi-GPU Implementation

In this section we describe an extension of the CUDA implementation explained in Section 5.1 so that it can exploit efficiently a GPU cluster. Basically we perform a row-block decomposition of the volume mesh taking into account the load balancing algorithms described in Section 4, and each submesh is assigned to a CPU process, which in turn uses a GPU to carry out the computations associated to its submesh. We use MPI [12] for the communication between the processes. Hereafter, we will use the *submesh* term to refer either the submesh or its associated CPU process.

5.2.1. Creation of the Submesh Data

In order to create the submeshes we perform by default an equitable partition of the domain by creating disjoint blocks of consecutive rows. If the problem is suitable to take advantage of the SLB algorithm described in Section 4.1, we perform the partition by applying this algorithm, which is fully implemented in CPU since it is executed only once at the beginning.

An edge of a submesh is a *communication edge* if its two adjacent volumes belong to different submeshes. A *communication volume* is a volume which has, at least, a communication edge. Note that each submesh, in order to process its communication edges, also needs to store the data of the communication volumes of its neighbouring submeshes. Therefore, at each iteration of the process, each submesh will have to send its communication volumes to its neighbouring submeshes. Figure 6 shows a partition example of a 8×12 mesh into 3 submeshes of size 8×4, where the sendings that are performed at each iteration are also indicated.

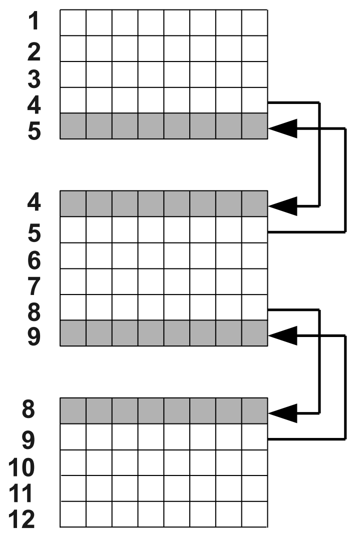


Figure 6: Example of data distribution among the submeshes using a 8 × 12 mesh. The ghost cells are noted in grey and the sendings performed at each iteration are noted with arrows.

We also impose that all the submeshes, but not necessarily the last, have an even number of rows, so that the edges are always processed globally in the same order regardless of the number of submeshes.

All the submeshes use the same data structures in GPU to store the data of their local volumes and edges, which were described in Section 5.1.

5.2.2. Multi-GPU Code

As we did in [17] for triangular meshes, we have implemented a multi-GPU algorithm which overlaps MPI communications with kernel processing and memory transfers between host and device. We use the *host* term to specify the CPU and its memory, while *device* refers to the GPU and its memory.

Algorithm 1 shows the general steps of the algorithm, in which the DLB algorithm is executed at every iteration for the sake of simplicity. We suppose that there are n submeshes numbered from 0 to $n-1$, where the submesh 0 is at the top. First of all, lines 1 and 2 get the local Δt of each submesh and the global initial Δt of all the submeshes applying MPI reduction, respectively. Line 4 evaluates the condition given by Equation (5) for the current submesh, and line 5 performs a reduction to check if at least one submesh satisfies this condition. If so, in line 7 the load balancing is carried out.

If no load balancing is performed at the current iteration, in lines 12–17 all the submeshes initiate the reception of the communication volumes from their adjacent submeshes by calling non-blocking MPI functions. Lines 18–19 copy from device to host the new states of the communication volumes obtained in the previous iteration. In lines 20–25 all the submeshes send their communication volumes to their adjacent submeshes. In lines 26–28 the vertical edges and the non-communication even horizontal edges are processed, since they do not need external data to be processed. In lines 29–34 all the submeshes wait for the communication volumes of their adjacent submeshes to arrive and, once they have arrived, in lines 35–36 they are copied to device. In line 37 only the even horizontal communication edges are processed by launching a specific kernel.

Note that the reception of the communication volumes from the adjacent submeshes (lines 12–17) could overlap with the memory transfers from device to host of the new states of the volumes (lines 18–19) and also with the processing of all the vertical edges and the even horizontal non-communication edges (lines 26–28). The sending of the communication volumes to the adjacent submeshes (lines 20–25) could also overlap with the edge processing performed in lines 26–28.

Finally, in line 39 the remaining edge processing kernel are executed for each submesh. Next, the state of the volumes of each submesh is updated and a new global time step Δt is computed.

Also note that, if the load balancing is performed, we do not need to exchange the communication volumes with the adjacent submeshes because a redistribution of the rows has been carried out inside the `loadBalancing` function of line 7. Next section gives more details about the load balancing implementation.

5.2.3. Load Balancing Code

Algorithm 2 shows the main steps of the `checkLoadBalancing` routine. We have four edge processing kernels (lines 1–4) to compute the weight of the submesh. The partial weight computed by each edge is stored in an accumulator of `float` elements in global memory, whose size is the number of volumes of the submesh. Line 5 performs a GPU reduction on the accumulator to get the weight Ω_s of the submesh, line 6 obtains the total weight of the global mesh, and finally, in lines 7 and 8 the condition given by Equation

Algorithm 1 Multi-GPU

```
1:  $\Delta t \leftarrow$ getInitialDeltaT(...)
2:  MPI_Allreduce($\Delta t$, min $\Delta t$, ...)
3:  while ($t < t_{end}$) do
4:    $balance\_submesh$ = checkLoadBalancing(...)
5:    MPI_Allreduce($balance\_submesh$, $balance$, ...)
6:    if ($balance$) then
7:      loadBalancing(...)
8:      processEvenVerticalEdges<<<grid, block>>>(...)
9:      processOddVerticalEdges<<<grid, block>>>(...)
10:     processEvenHorizontalEdges<<<grid, block>>>(...)
11:   else
12:     if ($submesh > 0$) then
13:       MPI_Irecv(Layers 1 and 2 of the lower comm. volumes of the upper submesh)
14:     end if
15:     if ($submesh < n - 1$) then
16:       MPI_Irecv(Layers 1 and 2 of the upper comm. volumes of the lower submesh)
17:     end if
18:     CudaMemcpy(Layers 1 and 2 of the upper comm. volumes from device to host)
19:     CudaMemcpy(Layers 1 and 2 of the lower comm. volumes from device to host)
20:     if ($submesh < n - 1$) then
21:       MPI_Isend(Layers 1 and 2 of the lower comm. volumes to the lower submesh)
22:     end if
23:     if ($submesh > 0$) then
24:       MPI_Isend(Layers 1 and 2 of the upper comm. volumes to the upper submesh)
25:     end if
26:     processEvenVerticalEdges<<<grid, block>>>(...)
27:     processOddVerticalEdges<<<grid, block>>>(...)
28:     processNonCommEvenHorizontalEdges<<<grid, block>>>(...)
29:     if ($submesh > 0$) then
30:       MPI_Waitall(Comm. volumes from upper submesh)
31:     end if
32:     if ($submesh < n - 1$) then
33:       MPI_Waitall(Comm. volumes from lower submesh)
34:     end if
35:     CudaMemcpy(Layers 1 and 2 of the upper comm. volumes from host to device)
36:     CudaMemcpy(Layers 1 and 2 of the lower comm. volumes from host to device)
37:     processCommEvenHorizontalEdges<<<grid, block>>>(...)
38:   end if
39:   processOddHorizontalEdges<<<grid, block>>>(...)
40:   getStateAndDeltaTVolumes<<<grid, block>>>(...)
41:   $t \leftarrow t + \min \Delta t$
42:   $\Delta t \leftarrow$ getMinimumDeltaT(...)
43:   MPI_Allreduce($\Delta t$, min $\Delta t$, ...)
44: end while
```

Algorithm 2 Check load balancing

```
getWeightsEvenHorizontalEdges<<<grid, block>>>(...)
getWeightsOddHorizontalEdges<<<grid, block>>>(...)
getWeightsEvenVerticalEdges<<<grid, block>>>(...)
getWeightsOddVerticalEdges<<<grid, block>>>(...)
$\Omega_s \leftarrow$ getWeightSubmesh(...)
MPI_Allreduce($\Omega_s$, $total\_weight$, ...)
$\Omega_i \leftarrow total\_weight/n$
$balance \leftarrow \left|1 - \frac{\Omega_s}{\Omega_i}\right| > d$
return $balance$
```

(5) is evaluated.

Algorithm 3 shows the general steps of the `loadBalancing` routine. First of all, in lines 1–3 we apply, for each row of the submesh and using CUDA streams, a GPU reduction on the accumulator which stores the partial weights. Once we have the weight of all the rows, we gather all the row weights of the global mesh in lines 4–5. Then we perform in CPU the domain partition by applying the SLB algorithm described in Section 4.1 and get the number of rows that must be sent and received from the lower and upper submeshes (if any). In lines 8 and 9 we copy to the host the rows that must be sent to the adjacent submeshes. Lines 10–15 perform the sending of the volumes, and in lines 16–23 we receive the volumes from the adjacent submeshes and copy them to device.

We suppose that every submesh only needs rows from its adjacent submeshes. The redistribution of the rows has been implemented using blocking MPI functions in order to keep the load balancing code independent and easily integrable in other numerical schemes.

6. Numerical Results

In this section we analyze the efficiency of the CUDA and multi-GPU implementations described in Section 5. For the multi-GPU implementation, we also study the influence on the obtained runtimes and scalability of the SLB and DLB algorithms by using several test problems with different amounts of wet and dry zones. Finally, the numerical scheme is validated by simulating the Lituya Bay tsunami.

Algorithm 3 Load balancing

```
1: for $i$ in $1 \dots nrows\_submesh$ do
2:     $\omega_i \leftarrow$ getWeightRow($i, \dots$)
3: end for
4: MPI_Allgather(Number of rows of all the submeshes)
5: MPI_Allgatherv(Vector of row weights of all the submeshes)
6: Apply SLB algorithm
7: Compute the number of rows to send/receive to/from adjacent submeshes
8: CudaMemcpy(Layers 1 and 2 of the upper volumes from device to host)
9: CudaMemcpy(Layers 1 and 2 of the lower volumes from device to host)
10: if ($submesh > 0$) and ($nrows\_to\_upper > 0$) then
11:     MPI_Bsend(Layers 1 and 2 of the upper volumes to the upper submesh)
12: end if
13: if ($submesh < n - 1$) and ($nrows\_to\_lower > 0$) then
14:     MPI_Bsend(Layers 1 and 2 of the lower volumes to the lower submesh)
15: end if
16: if ($submesh > 0$) and ($nrows\_from\_upper > 0$) then
17:     MPI_Recv(Layers 1 and 2 of the lower volumes of the upper submesh)
18:     CudaMemcpy(Layers 1 and 2 of the upper volumes from host to device)
19: end if
20: if ($submesh < n - 1$) and ($nrows\_from\_lower > 0$) then
21:     MPI_Recv(Layers 1 and 2 of the upper volumes of the lower submesh)
22:     CudaMemcpy(Layers 1 and 2 of the lower volumes from host to device)
23: end if
```

6.1. One GPU

In this section we analyze the efficiency of the CUDA implementation described in Section 5.1 for one GPU using a test problem consisting of an internal circular dam break in the $[-5, 5] \times [-5, 5]$ domain. The depth function is given by $H(x, y) = 5$ and the initial conditions are:

$$W_i^0(x, y) = \big(h_1(x, y), \quad 0, \quad 0, \quad h_2(x, y), \quad 0, \quad 0\big)^{\mathrm{T}}, \quad i = 1, \dots, L. \qquad (6)$$

where

$$h_1(x, y) = \begin{cases} 4.0 & \text{if } \sqrt{x^2 + y^2} > 1.5 \\ 0.5 & \text{otherwise} \end{cases} \quad , \quad h_2(x, y) = H(x, y) - h_1(x, y)$$

The ratio of densities is $r = 0.5$, CFL parameter is $\gamma = 0.9$ and simulation time is 4 seconds. The Coulomb angle is $\alpha = 12°$ and the friction parameters

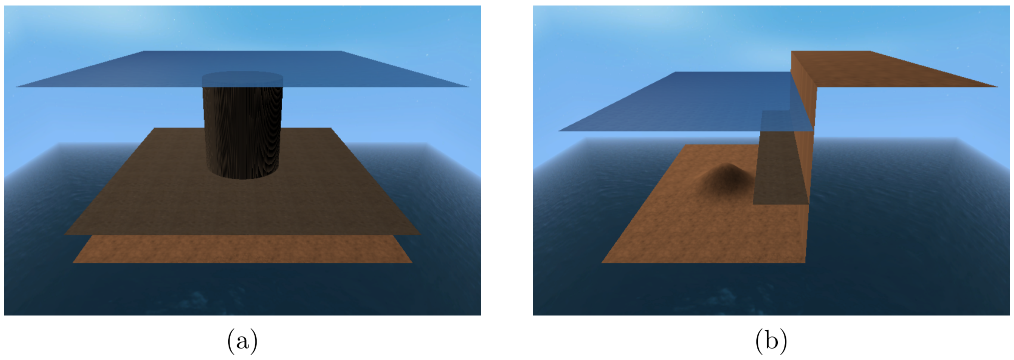

(a) (b)

Figure 7: Initial state of the artificial test problems: (a) Dam break, (b) Wet-dry.

Volumes	Iterations	GTX Titan Black	MVols/s
100 × 100	652	0.096	67.9
200 × 200	1315	0.38	139.2
400 × 400	2647	1.98	214.3
800 × 800	5322	13.90	245.0
1600 × 1600	10693	107.62	254.4
2400 × 2400	16075	366.28	252.8
3200 × 3200	21463	862.12	254.9

Table 1: Number of iterations performed, execution times in seconds and millions of volumes processed per second for the dam break test.

are $m_f = 0.02$, $n_1 = 0.02$ and $n_2 = 0.05$. We consider wall boundary conditions ($\boldsymbol{q}_1 \cdot \boldsymbol{\eta} = 0,\ \boldsymbol{q}_2 \cdot \boldsymbol{\eta} = 0$). Figure 7a shows the initial state.

The CUDA program was executed using a GeForce GTX Titan Black. Table 1 shows the number of iterations performed, the runtimes in seconds and the millions of volumes processed per second for all the mesh sizes. Figure 8 shows graphically the GFLOPS and GB/s, obtained using the `nvprof` utility, for the two main kernels. Theoretical maximums for the GTX Titan Black are 5.1 TFLOPS in single precision and 336 GB/s. We can see that both kernels have reached approximately half the maximum theoretical bandwidth, and the edge and volume processing kernels have achieved about 570 and 850 GFLOPS, respectively. The CUDA implementation has also been able to process up to 254 millions of volumes per second.

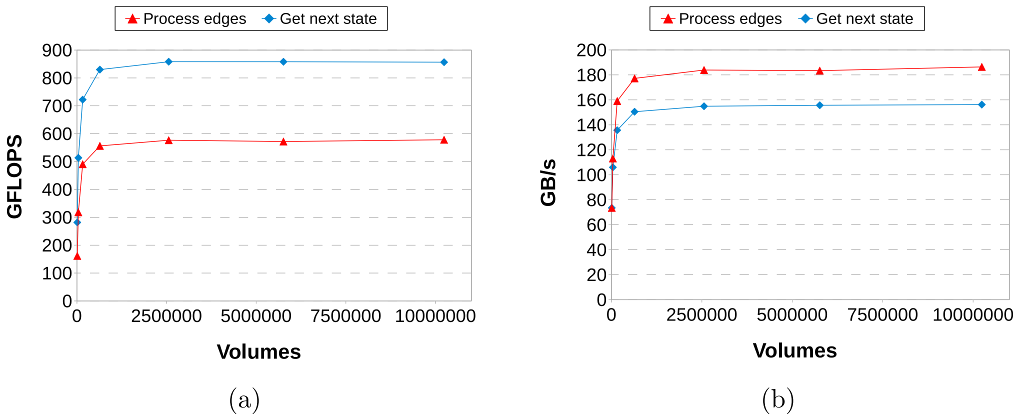


Figure 8: GFLOPS and GB/s reached for the two main kernels in the dam break test problem: (a) GFLOPS, (b) GB/s.

6.2. Multi-GPU

In this section we analyze the strong and weak scalability [29] obtained with the multi-GPU distributed implementation described in Section 5.2 for several real and artificial examples. In order to measure the strong scalability, the number of GPUs is increased keeping constant the global problem size (in our case, we consider the number of volumes to specify the problem size), whereas the weak scalability is studied by increasing the global problem size so that the problem size per GPU remains constant.

The multi-GPU implementation has been executed, for all the real and artificial test problems, on the Picasso supercomputer located at the Supercomputing and Bioinnovation Center (SCBI) of the University of Málaga. This cluster is formed by 16 nodes, where each node has two Intel Xeon E5-2670 processors, 64 GB of RAM memory and two Tesla M2075 cards. The nodes are interconnected by an InfiniBand FDR network [30]. We have used up to 24 GPUs of this cluster to perform the experiments.

In the load balancing algorithms (see Section 4), we have obtained the weights of a wet/dry edge and a wet/dry volume by considering the dam break test problem presented in Section 6.1 with the 3200 $\times$ 3200 mesh. Using a Tesla M2075 we have obtained the following weights for our distributed solver: 1.0 and 0.44 for a wet and dry edge, respectively, and 0.98 and 0.74 for a wet and dry volume, respectively.

6.2.1. Artificial Problems

We consider two artificial test problems:

- *Dam break*: The dam break example used in Section 6.1.
- *Wet-dry*: This test is also defined in the $[-5,5] \times [-5,5]$ interval. Initially layer 1 exists in the 60 % of the domain, and layer 2 exists in the 15 % of the domain. Figure 7b shows the initial state. The depth function is:

$$H(x,y) = \begin{cases} 1.0 & \text{if } x < -1.0 \\ 6.0 - e^{-(x-2)^2-y^2} & \text{otherwise} \end{cases}$$

The initial conditions are given by (6), where

$$h_2(x,y) = \begin{cases} 2.0 - e^{-(x-2)^2-y^2} & \text{if } -1.0 \leq x \leq 0.5 \\ 0.0 & \text{otherwise} \end{cases},$$

$$h_1(x,y) = \begin{cases} 4.0 - e^{-(x-2)^2-y^2} - h_2(x,y) & \text{if } x \geq -1.0 \\ 0.0 & \text{otherwise} \end{cases}$$

For both tests problems, the ratio of densities is $r = 0.5$, CFL parameter is $\gamma = 0.9$, Coulomb angle is $\alpha = 12^\circ$ and friction parameters are $m_f = 0.02$, $n_1 = 0.02$ and $n_2 = 0.05$. We consider wall boundary conditions ($\boldsymbol{q}_1 \cdot \boldsymbol{\eta} = 0$, $\boldsymbol{q}_2 \cdot \boldsymbol{\eta} = 0$).

The strong and weak scalabilities are also measured with and without the SLB algorithm for the wet-dry test problem rotated 90 degrees with respect to the z axis in order to study the influence of the distribution of the wet and dry zones among the submeshes and the effectiveness of the SLB algorithm.

For measuring the strong scalability, we have considered 1, 2, 4, 8, 12, 16, 20 and 24 GPUs. In order to guarantee that all the submeshes have the same number of rows, we set the total number of rows to be 3360 in all the cases. The mesh sizes considered are 2000×3360, 4000×3360 and 8000×3360. The simulation time is 20 seconds.

For measuring the weak scalability, we have also considered 1, 2, 4, 8, 12, 16, 20 and 24 GPUs. For all the test problems, we analyze the obtained scalability when assigning one million of volumes per GPU. The simulation time is also 20 seconds.

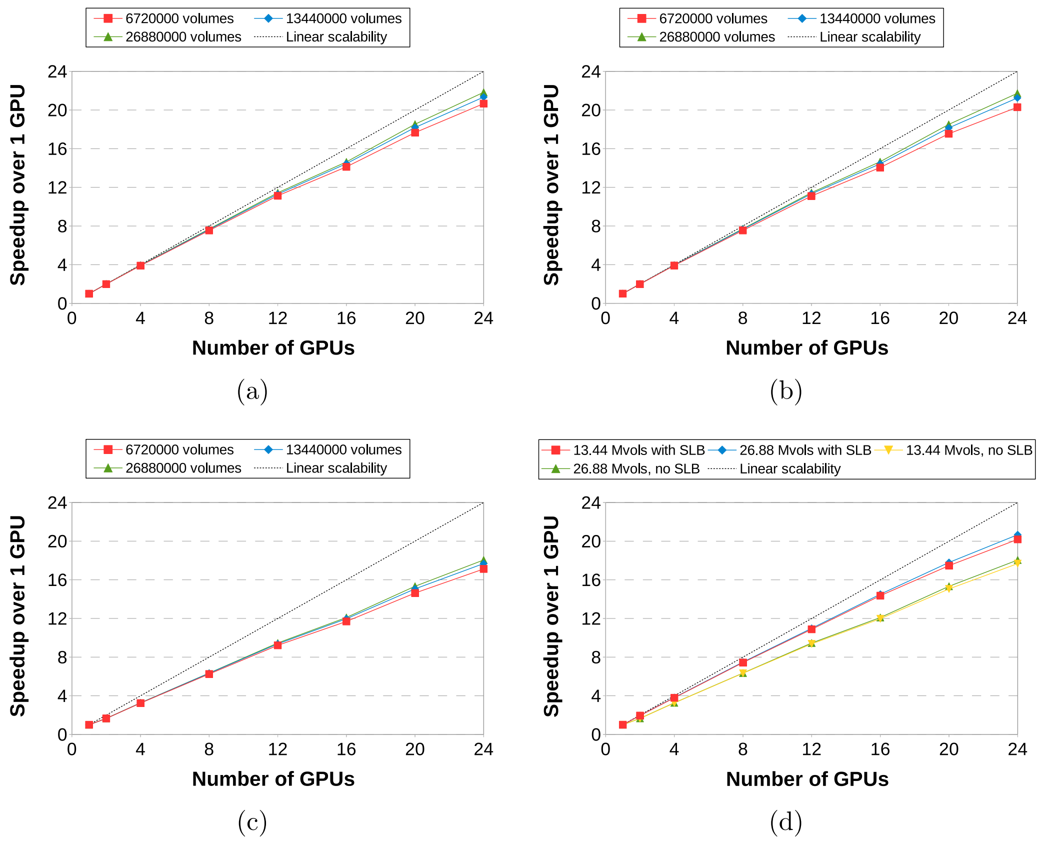
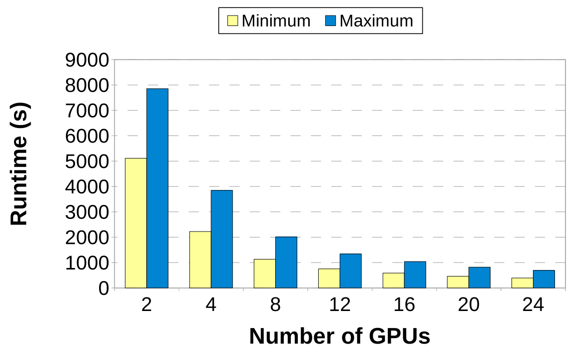
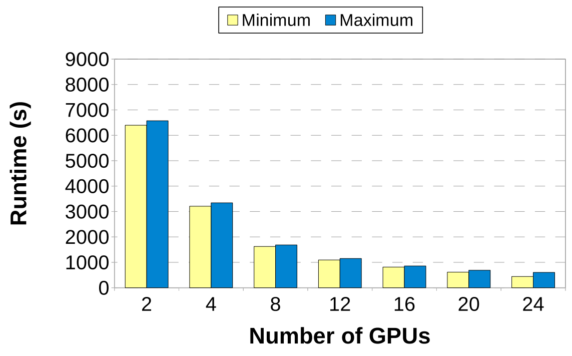


Figure 9: Strong scalability for all the artificial test problems: (a) Dam break, (b) Wet-dry, (c) Wet-dry rotated 90°, (d) Wet-dry rotated 90° with and without SLB.

Figure 9 is associated with the strong scalability and it represents the evolution of the speedup with respect to one GPU when increasing the number of GPUs for all the mesh sizes and examples. As we can see, in general, the bigger the mesh size, the better the speedup, because for small meshes the GPUs are not fully exploited and MPI communications have a greater impact in the execution time than with bigger meshes. In the dam break and wet-dry test problems, we have obtained a scalability close to linear for the two biggest meshes using up to 24 GPUs because all the submeshes have a similar computational load. However, in the rotated wet-dry problem, we have reached a notably worse strong scalability as expected, since some submeshes have different amounts of wet and dry zones than other submeshes. When applying the SLB algorithm to the two biggest meshes in the rotated



(a)



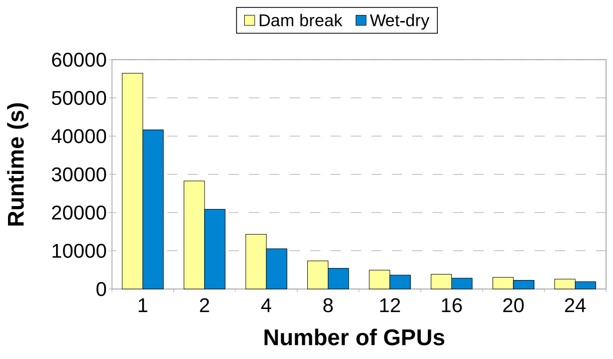
(b)

Figure 10: Minimum and maximum runtimes in seconds of the processing of edges and volumes for all the MPI processes using the mesh of 13.440.000 volumes for the rotated wet-dry test problem: (a) Without the SLB algorithm, (b) With the SLB algorithm.

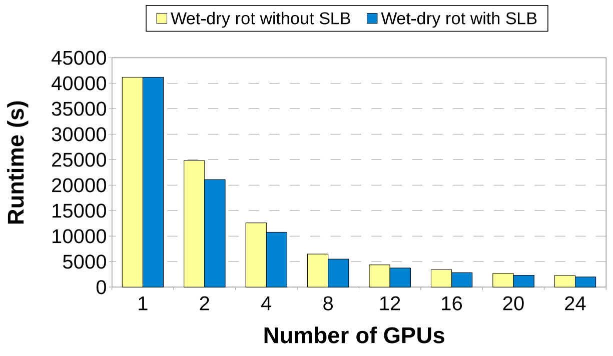
wet-dry test problem, in both cases the runtimes have reduced up to a 17 % (see Table 2 for a detailed percentage of reduction using the biggest mesh) and the speedup with respect to one GPU has improved up to a 20 %.

In order to make it clear the benefits of the SLB algorithm, in Figure 10 we depict the minimum and maximum runtime of the processing of edges and volumes for all the MPI processes using the rotated wet-dry test problem and the medium mesh with and without the SLB algorithm. When applying this algorithm, the computational load becomes much better balanced in all the cases.

In the dam break test problem we have been able to process approximately 1900 millions of volumes per second using the largest mesh and 24 GPUs (see



(a)



(b)

Figure 11: Execution times in seconds using the mesh of 26.880.000 volumes for all the artificial test problems: (a) Dam break and wet-dry, (b) Wet-dry rotated 90° with and without the SLB algorithm.

Figure 16). Finally, Figure 11 shows the execution times for all the artificial test problems and number of GPUs with the mesh size of 8000×3360 volumes.

Figure 12, for its part, is associated with the weak scalability and it represents the evolution of the efficiency when increasing the number of GPUs keeping constant the number of volumes per GPU. As it happens with the strong scalability, we have obtained rather similar weak scalability behaviour for the dam break and wet-dry test problem for up to 24 GPUs (approximately efficiency values of 0.94–0.95 using one million of volumes per GPU). In the rotated version of the wet-dry example, nevertheless, the scalability drops significantly again because of the uneven distribution of the wet and dry zones among the submeshes, obtaining efficiencies of 0.78–0.79. When

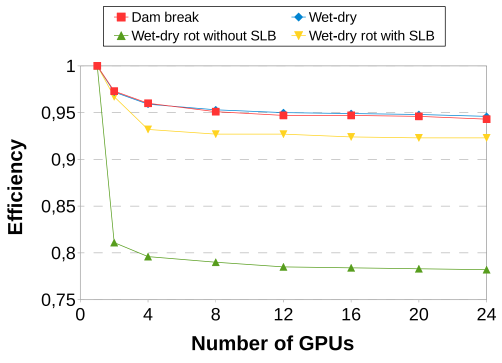


Figure 12: Weak scalability for all the artificial test problems using one million of volumes per GPU.

Number of GPUs	Runtime reduction (%)		
	Wet-dry rot	Sumatra	Lituya 2x5m
2	15.0	7.1	3.3
4	14.5	10.2	7.9
8	15.2	10.9	6.6
12	13.8	10.7	8.2
16	16.6	10.8	8.7
20	13.9	8.0	4.9
24	12.7	4.6	4.4

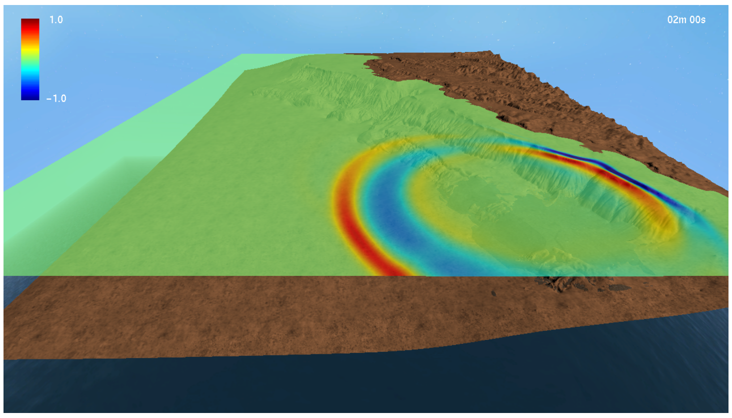
Table 2: Percentage of runtime reduction when applying the load balancing algorithms.

applying the SLB algorithm to this example, the efficiency has improved notably reaching 0.92–0.93.

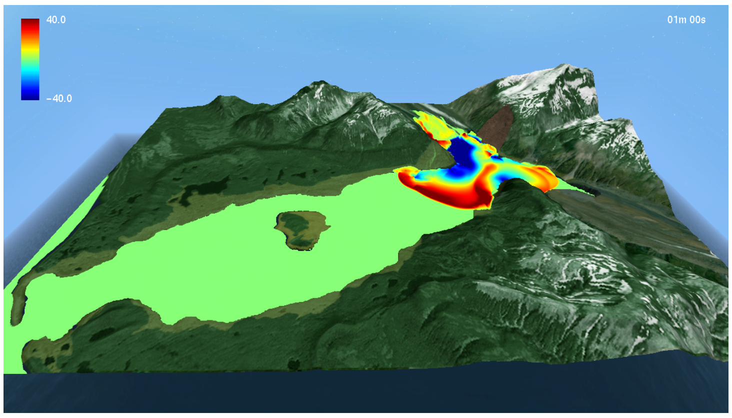
6.2.2. Real Problems

In order to study the performance of the solver when applied to realistic scenarios, we have considered two test problems using real data:

- *Tsunami simulation in Sumatra*: This example simulates a paleotsunami occurred in 1797 in Sumatra. The tsunami is generated by a submarine landslide near the Siberut island. The topographic and bathymetric data have been provided by the Institut de Physique du Globe of Paris. The mesh size is 2410×3092 volumes with a resolution of 22 meters.



(a)

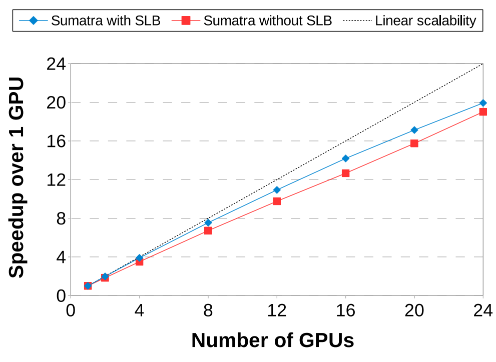


(b)

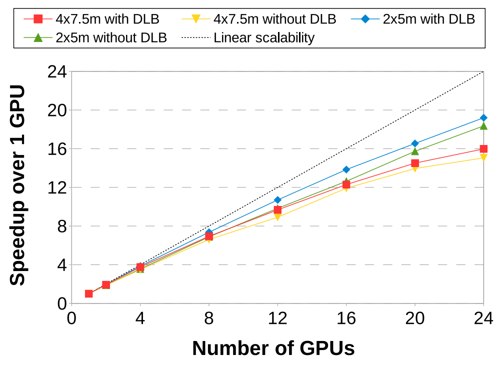
Figure 13: Tsunami simulations using real data: (a) Sumatra, 2 min. (b) Lituya Bay, 1 min.

The ratio of densities is $r = 0.4$, CFL parameter is $\gamma = 0.8$, Coulomb angle is $\alpha = 10^\circ$ and friction parameters are $m_f = 10^{-4}$, $n_1 = 0.05$ and $n_2 = 0.4$. We consider open boundary conditions. The simulation time is 40 minutes. Figure 13a shows an image of the simulation.

- *Tsunami simulation in Lituya Bay*: This example simulates the tsunami occurred on July 9, 1958 in Lituya Bay, Alaska. In this case, the tsunami is generated by a subaerial landslide. The topographic and bathymetric data are taken from several sources (digital elevation models from the Shuttle Radar Topography Mission, and data from two



(a)



(b)

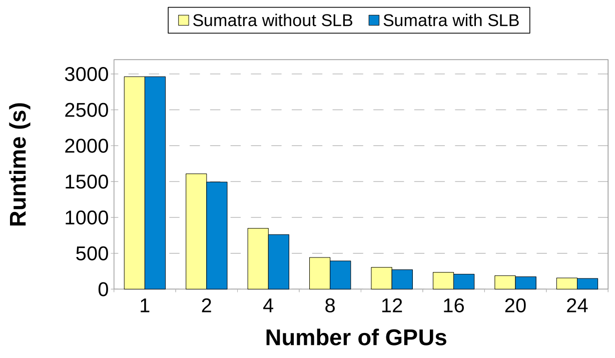
Figure 14: Strong scalability for all the real test problems: (a) Sumatra, (b) Lituya Bay.

oceanographic campaigns of the NOAA's National Geophysical Data Center). We have also performed a data reconstruction process in order to represent the Gilbert Inlet bathymetry before the event. For this test problem, we use two different meshes: a small mesh of $3650 \times 1271 = 4.639.150$ volumes with a resolution of 4×7.5 m^2, and a large mesh of $7301 \times 1907 = 13.923.007$ volumes with a resolution of 2×5 m^2.

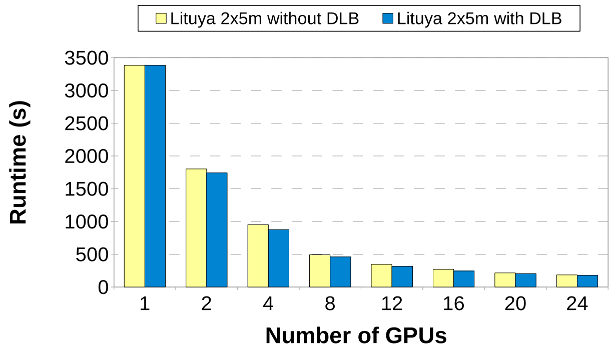
The ratio of densities is $r = 0.44$, CFL parameter is $\gamma = 0.9$, Coulomb angle is $\alpha = 13°$ and friction parameters are $m_f = 0.08$, $n_1 = 0.002$ and $n_2 = 0.05$. We consider open boundary conditions. The simulation time is 12 minutes. Figure 13b shows an image of the simulation using the small mesh.

We have executed the Sumatra test with and without the SLB algorithm, and the Lituya Bay test with and without the DLB algorithm since there is a significant inundation of the coastal areas in this last test. Specifically, the wet zone occupies the 24 % of the domain at the initial state, and it reaches a maximum of 37 %. In the DLB algorithm, we check the balancing condition every 1000 iterations and $d = 5\%$.

Figure 14 shows graphically the obtained strong scalability for both test problems using up to 24 GPUs, and Figure 15 shows the execution times. In the Sumatra example, as expected, we have reached better speedup with respect to one GPU in all the cases by using the SLB algorithm than with the trivial domain decomposition. In particular, the runtimes have reduced up to a 11 % (see Table 2), and the speedup over one GPU has improved



(a)



(b)

Figure 15: Execution times in seconds for all the real test problems: (a) Sumatra, (b) Lituya Bay.

up to a 12 %. Using 24 GPUs, we have achieved speedups of approximately 20 and 19 with and without the SLB algorithm, respectively, with respect to one GPU.

About the Lituya Bay example, like in the artificial test problems, we have obtained better strong scalability with the largest mesh. When applying the DLB algorithm, the runtimes have reduced up to a 9 % (see Table 2) and the speedup over a single GPU has augmented up to a 10 % for the biggest mesh. We have reached an approximate speedup of 19 and 18 with and without the DLB algorithm, respectively, with respect to one GPU using the biggest mesh and 24 GPUs. The overhead introduced by the DLB has represented at most the 0.2 % of the total runtime. Better results may be expected in problems which present even higher variations of the wet and

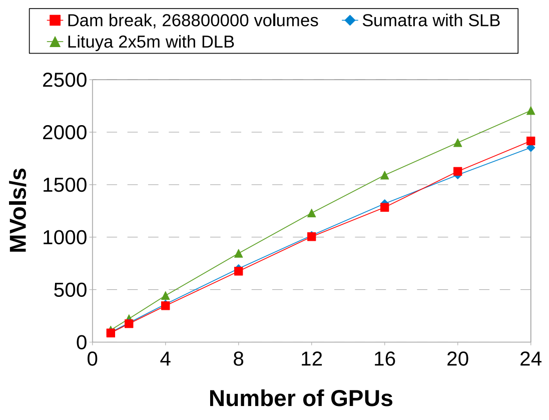


Figure 16: Millions of volumes processed per second for some test problems.

dry zones among the subdomains during the simulation.

In Figure 16 we can see that, in the dam break and the Sumatra problems, it have been processed approximately 1900 millions of volumes per second using 24 GPUs, while in the Lituya Bay example we have been able to process 2200 millions of volumes per second due to the big dry area of the domain.

6.2.3. The Lituya Bay Tsunami

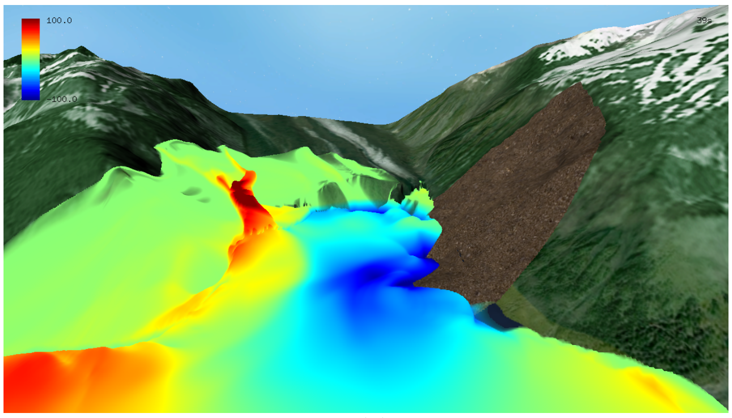
In this section we carry out a validation of the numerical scheme described in Section 3 by simulating the Lituya Bay tsunami, introduced in Section 6.2.2, and comparing the obtained results with the real field data.

A magnitude 8.3 earthquake in Alaska on July 9, 1958, triggered a landslide in Gilbert Inlet which caused the largest tsunami ever recorded. The upper limit of destruction of forest by water (known as trimline) extended up to 524 meters above mean sea level, and the water reached a maximum distance of 1100 meters in from the shoreline at Fish Lake. See [31] for a more detailed description of the tsunami.

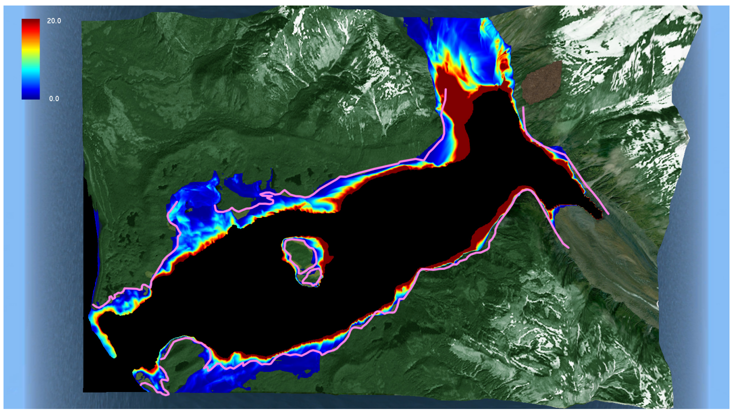
The two meshes used in Section 6.2.2 for this test problem provide similar results. Therefore, here we only show the results with the small mesh.

The input parameters were cited in Section 6.2.2. In order to find the optimal values for the ratio of densities, the Coulomb angle and the friction parameters, many executions have been carried out so that the obtained results were as close as possible to the real field data.

In general, there is a good agreement between the obtained impact times, wave heights and trimlines with the real measures described in [31]. Figure



(a)



(b)

Figure 17: Validation of the numerical scheme using the Lituya Bay tsunami: (a) Maximum run-up of 523.9 meters, (b) Real trimline (in pink) and maximum wave heights.

17a shows a screenshot of the simulation corresponding to the maximum run-up of 523.9 meters achieved in Gilbert Inlet at 39 seconds. Figure 17b depicts the real trimline taken from [31] along with the flooded areas and wave heights reached with the simulation. We can see that the trimline is well adjusted in Gilbert Inlet, the east area of the north shore, The Paps, the Cenotaph island, La Chausse Spit and the south coast in front of it. The water also reaches more than 1100 meters in from the shoreline at Fish Lake. Note also that the water breaks through the Cenotaph island opening a narrow channel as described in [31]. Figure 18 shows closer views for the Gilbert Inlet and the Cenotaph island. However, in the flat zones of Fish

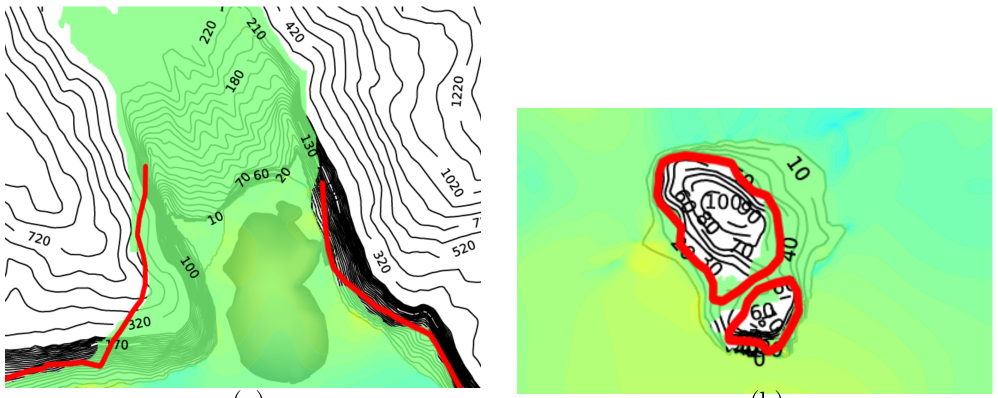

(a) (b)

Figure 18: Closer view of maximum run-up and the real trimline in red: (a) Gilbert Inlet, (b) Cenotaph island.

Lake and the east of The Paps in the south shoreline there is a worse fit of the trimline, probably because it would be necessary to consider different friction coefficients depending on the type of vegetation. See [32] for more details about this numerical simulation.

7. Conclusions

In this paper we have proposed a 2D IFCP numerical scheme based on a first order two-layer Savage-Hutter type model to simulate tsunamis generated by submarine landslides. A multi-GPU implementation over structured meshes which uses overlapping techniques to improve the efficiency has been developed and tested using several artificial and real problems. The results show good strong and weak scalability in problems with equitable computational load among the subdomains using up to 24 GPUs. The numerical scheme has also been validated using the Lituya Bay tsunami, showing that the obtained results are comparable to the real field data.

We have also proposed a static and a dynamic load balancing algorithm which are useful in problems where a trivial row-block domain decomposition provides an unbalanced computational load due to different amount of wet and dry areas among the submeshes. These load balancing algorithms have been integrated in the multi-GPU solver and have allowed us to improve the speedup with respect to one GPU up to a 12 % in some real problems. Better results may be expected if the variations of wet and dry zones are even higher, such as big landslides and inundations.

Appendix A. Definition of the 1D numerical fluxes

In this section we describe the 1D numerical fluxes $\Phi^{\pm}_{\boldsymbol{\eta}_{ij}}$ and $\Phi^{\pm}_{\boldsymbol{\eta}^{\perp}_{ij}}$ corresponding to the 1D projected Riemann problem associated to edge Γ_{ij}. The following notation is used here: $W_{\boldsymbol{\eta}_{ij}} = \begin{bmatrix} h_1 & q_{1,\boldsymbol{\eta}_{ij}} & h_2 & q_{2,\boldsymbol{\eta}_{ij}} \end{bmatrix}^{\mathrm{T}} = \big(T_{\boldsymbol{\eta}_{ij}} W\big)_{[1,2,4,5]}$, and $W_{\boldsymbol{\eta}_{ij}{}^{\perp}} = \begin{bmatrix} q_{1,\boldsymbol{\eta}^{\perp}_{ij}} & q_{2,\boldsymbol{\eta}^{\perp}_{ij}} \end{bmatrix}^{\mathrm{T}} = \big(T_{\boldsymbol{\eta}_{ij}} W\big)_{[3,6]}$, where $W_{[i_1,\ldots,i_s]}$ is the vector defined from W taking its components $i_1, \ldots, i_s$.

Appendix A.1. Definition of $\Phi^{\pm}_{\boldsymbol{\eta}_{ij}}$

$\Phi^{\pm}_{\boldsymbol{\eta}_{ij}}$ corresponds to the IFCP numerical flux applied to 1D two-layer shallow-water system and it is defined as follows:

$$\begin{aligned} \Phi^{-}_{\boldsymbol{\eta}_{ij}} &= \frac{1}{2}\Big(R_{ij} - \big(\alpha_0 \widetilde{I}^{\boldsymbol{\tau}}_{ij} + \alpha_1 R^{\boldsymbol{\tau}}_{ij} + \alpha_2 \mathcal{A}_{ij} R^{\boldsymbol{\tau}}_{ij}\big)\Big) + F_Q(W_{\boldsymbol{\eta}_{ij},i}) \,, \\ \Phi^{+}_{\boldsymbol{\eta}_{ij}} &= \frac{1}{2}\Big(R_{ij} + \big(\alpha_0 \widetilde{I}^{\boldsymbol{\tau}}_{ij} + \alpha_1 R^{\boldsymbol{\tau}}_{ij} + \alpha_2 \mathcal{A}_{ij} R^{\boldsymbol{\tau}}_{ij}\big)\Big) - F_Q(W_{\boldsymbol{\eta}_{ij},j}) \,, \end{aligned} \tag{A.1}$$

where the coefficients α_k, $k = 0, 1, 2$ are defined in terms of the eigenvalues of the 1D two-layer shallow-water system following [26].

Here, the matrix $\mathcal{A}_{ij}$ is defined by

$$\mathcal{A}_{ij} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ g h_{1,ij} - \big(u_{1,\boldsymbol{\eta}_{ij}}\big)^2 & 2u_{1,\boldsymbol{\eta}_{ij}} & g h_{1,ij} & 0 \\ 0 & 0 & 0 & 1 \\ r g h_{2,ij} & 0 & g h_{2,ij} - \big(u_{2,\boldsymbol{\eta}_{ij}}\big)^2 & 2u_{2,\boldsymbol{\eta}_{ij}} \end{pmatrix} \tag{A.2}$$

with

$$u_{l,\boldsymbol{\eta}_{ij}} = \begin{pmatrix} u_{l,ij,x} & u_{l,ij,y} \end{pmatrix} \cdot \boldsymbol{\eta}_{ij}, \quad u_{l,ij,\alpha} = \frac{\sqrt{h_{l,i}}\, u_{l,i,\alpha} + \sqrt{h_{l,j}}\, u_{l,j,\alpha}}{\sqrt{h_{l,i}} + \sqrt{h_{l,j}}} \,,$$

and

$$h_{l,ij} = \frac{h_{l,i} + h_{l,j}}{2} \,, \quad l = 1, 2, \quad \alpha = x, y.$$

$F_Q(W_{\boldsymbol{\eta}_{ij}})$ is given by

$$F_Q(W_{\boldsymbol{\eta}}) = \begin{pmatrix} q_{1,\boldsymbol{\eta}} & \dfrac{q_{1,\boldsymbol{\eta}}^2}{h_1} & q_{2,\boldsymbol{\eta}} & \dfrac{q_{2,\boldsymbol{\eta}}^2}{h_2} \end{pmatrix}^T .$$

$\widetilde{I}_{ij}^{\boldsymbol{\tau}}$ is defined by

$$\widetilde{I}_{ij}^{\boldsymbol{\tau}} = \begin{cases} \widetilde{I}_{ij} & \text{if } |h_{2,ij}\, u_{2,\boldsymbol{\eta}_{ij}}| > \Delta t\, \sigma_{ij}^c \\ \begin{pmatrix} \mu_{1,j} - \mu_{1,i} \\ q_{1,j,\boldsymbol{\eta}} - q_{1,i,\boldsymbol{\eta}} \\ 0 \\ q_{2,j,\boldsymbol{\eta}} - q_{2,i,\boldsymbol{\eta}} \end{pmatrix} = \begin{pmatrix} \Delta_{ij}\mu_1 \\ \Delta_{ij} q_{1,\boldsymbol{\eta}} \\ 0 \\ \Delta_{ij} q_{2,\boldsymbol{\eta}} \end{pmatrix} & \text{otherwise} \end{cases}$$

with $\sigma_{ij}^c = g(1-r)h_{2,ij}\tan(\alpha)$ and

$$\widetilde{I}_{ij} = \begin{pmatrix} \Delta_{ij}\mu_1 - \Delta_{ij}\mu_2 & \Delta_{ij} q_{1,\boldsymbol{\eta}} & \Delta_{ij}\mu_2 & \Delta_{ij} q_{2,\boldsymbol{\eta}} \end{pmatrix}$$

where $q_{l,i,\boldsymbol{\eta}}$, $l = 1,2$, is the value of $q_{l,\boldsymbol{\eta}}$ in V_i, and $\mu_{l,i}$, $l = 1,2$, is the value of μ_l in V_i, with $\mu_1 = h_1 + h_2 - H$ and $\mu_2 = h_2 - H$.

R_{ij} is defined by

$$R_{ij} = F_Q(W_{\boldsymbol{\eta}_{ij},j}) - F_Q(W_{\boldsymbol{\eta}_{ij},i}) + P_{ij},$$

where

$$P_{ij} = \Big(0 \quad gh_{1,ij}\Delta_{ij}\mu_1 \quad 0 \quad gh_{2,ij}\big(r\Delta_{ij}\mu_1 + (1-r)\Delta_{ij}\mu_2\big)\Big)^T$$

and finally, $R_{ij}^{\boldsymbol{\tau}}$ is defined by:

$$R_{ij}^{\boldsymbol{\tau}} = \begin{cases} R_{ij} & \text{if } |h_{2,ij}\, u_{2,\boldsymbol{\eta}_{ij}}| > \Delta t\, \sigma_{ij}^c \\ F_Q(W_{\boldsymbol{\eta}_{ij},j}) - F_Q(W_{\boldsymbol{\eta}_{ij},i}) + P_{ij}^* & \text{otherwise} \end{cases}$$

where

$$P_{ij}^* = \Big(0 \quad gh_{1,ij}\Delta_{ij}\mu_1 \quad 0 \quad rgh_{2,ij}\Delta_{ij}\mu_1\Big)^T .$$

Appendix A.2. Definition of $\Phi^{\pm}_{\boldsymbol{\eta}^{\perp}_{ij}}$

As in the one-layer and the two-layer shallow-water system, $q_{l,\boldsymbol{\eta}^{\perp}}$, $l = 1,2$, behaves as a passive scalar that is transported by $u_{l,\boldsymbol{\eta}}$, $l = 1,2$. Using this fact, we could define the following numerical flux for the two transport equations:

$\Phi^{\pm}_{\boldsymbol{\eta}^{\perp}_{ij}} = \mp \left[\left(\Phi^{-}_{\boldsymbol{\eta}_{ij}}\right)_{[1]} u^{*}_{1,\boldsymbol{\eta}^{\perp}_{ij}} \quad \left(\Phi^{-}_{\boldsymbol{\eta}_{ij}}\right)_{[3]} u^{*}_{2,\boldsymbol{\eta}^{\perp}_{ij}} \right]^{T}$, where $u^{*}_{k,\boldsymbol{\eta}^{\perp}_{ij}}$ is defined as follows:

$$u^{*}_{l,\boldsymbol{\eta}^{\perp}_{ij}} = \begin{cases} \dfrac{q_{l,i,\boldsymbol{\eta}^{\perp}_{ij}}}{h_{l,i}} & \text{if } \left(\Phi^{-}_{\boldsymbol{\eta}_{ij}}\right)_{[2l-1]} > 0 \\ \dfrac{q_{l,j,\boldsymbol{\eta}^{\perp}_{ij}}}{h_{l,j}} & \text{otherwise} \end{cases}, \quad l = 1,2.$$

Acknowledgements

This work has been partially supported by FEDER and the Spanish and Andalusian research projects MTM2012-38383-C02-01, MTM2014-52056-P, P11-RNM7069 and P11-FQM8179, and the BSC project FI-2012-2-0004. The authors also thank J. M. González, J. Macías and C. Sánchez for their work in the Lituya Bay tsunami problem.

References

[1] P. Heinrich, A. Piatanesi, H. Hébert, Numerical modelling of tsunami generation and propagation from submarine slumps: The 1998 Papua New Guinea event, Geophysical Journal International 145 (2001) 97–111.

[2] E. D. Fernández-Nieto, F. Bouchut, D. Bresch, M. J. Castro, A. Mangeney, A new Savage-Hutter type model for submarine avalanches and generated tsunami, Journal of Computational Physics 227 (2008) 7720–7754.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (2008) 1370–1380.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU computing, Proceedings of the IEEE 96 (2008) 879–899.

[5] NVIDIA, CUDA C Programming Guide, http://docs.nvidia.com/cuda/index.html, Accessed February 2016.

[6] Khronos OpenCL Working Group, The OpenCL Specification, http://www.khronos.org/opencl, Accessed February 2016.

[7] J. Fang, A. L. Varbanescu, H. Sips, A comprehensive performance comparison of CUDA and OpenCL, in: 40th International Conference on Parallel Processing (ICPP 2011), Taipei (Taiwan), pp. 216–225.

[8] M. de la Asunción, J. M. Mantas, M. J. Castro, Simulation of one-layer shallow water systems on multicore and CUDA architectures, Journal of Supercomputing 58 (2011) 206–214.

[9] A. R. Brodtkorb, M. L. Sætra, M. Altinakar, Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation, Computers & Fluids 55 (2012) 1–12.

[10] M. de la Asunción, J. M. Mantas, M. J. Castro, Programming CUDA-based GPUs to simulate two-layer shallow water flows, in: P. D'ambra, M. Guarracino, D. Talia (Eds.), Euro-Par 2010, volume 6272 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 353–364.

[11] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, J. M. Gallardo, GPU computing for shallow water flow simulation based on finite volume schemes, Comptes Rendus Mécanique 339 (2011) 165–184.

[12] Message Passing Interface Forum: A Message Passing Interface Standard, University of Tennessee, Knoxville, Tennessee.

[13] M. A. Acuña, T. Aoki, Real-time tsunami simulation on a multi-node GPU cluster, ACM/IEEE Conference on Supercomputing 2009 (SC 2009) [Poster], Portland (USA), November 2009.

[14] W. Xian, A. Takayuki, Multi-GPU performance of incompressible flow computation by Lattice Boltzmann method on GPU cluster, Parallel Computing 37 (2011) 521–535.

[15] P. Castonguay, D. M. Williams, P. E. Vincent, M. López, A. Jameson, On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids, in: 20th AIAA Computational Fluid Dynamics Conference, Honolulu (USA).

[16] M. Viñas, J. Lobeiras, B. B. Fraguela, M. Arenaz, M. Amor, J. A. García, M. J. Castro, R. Doallo, A multi-GPU shallow-water simulation with transport of contaminants, Concurrency and Computation: Practice and Experience 25 (2012) 1153–1169.

[17] M. de la Asunción, J. M. Mantas, M. J. Castro, E. D. Fernández-Nieto, An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems, Journal of Parallel and Distributed Computing, Special Issue on Accelerators for High-Performance Computing 72 (2012) 1065–1072.

[18] D. A. Jacobsen, I. Senocak, Multi-level parallelism for incompressible flow computations on GPU clusters, Parallel Computing 39 (2013) 1–20.

[19] M. Geveler, D. Ribbrock, S. Mallach, D. Göddeke, A simulation suite for Lattice-Boltzmann based real-time CFD applications exploiting multi-level parallelism on modern multi- and many-core architectures, Journal of Computational Science 2 (2011) 113–123.

[20] S. Rostrup, H. D. Sterck, Parallel hyperbolic PDE simulation on clusters: Cell versus GPU, Computer Physics Communications 181 (2010) 2164–2179.

[21] A. R. Brodtkorb, M. L. Sætra, Shallow water simulations on multiple GPUs, in: PARA 2010: State of the Art in Scientific and Parallel Computing, Reykjavik (Iceland).

[22] S. Blaise, A. St-Cyr, A dynamic hp-adaptive discontinuous Galerkin method for shallow-water flows on the sphere with application to a global tsunami simulation, Monthly Weather Review 140 (2012) 978–996.

[23] R. V. Dorneles, R. L. Rizzi, A. L. Martinotto, D. Picinin, P. O. A. Navaux, T. A. Diverio, Parallel computational model with dynamic load balancing in PC clusters, in: M. Daydé, J. Dongarra, V. Hernández,

J. M. L. M. Palma (Eds.), High Performance Computing for Computational Science - VECPAR 2004, volume 3402 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 468–479.

[24] B. F. Sanders, J. E. Schubert, R. L. Detwiler, ParBreZo: A parallel, unstructured grid, Godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale, Advances in Water Resources 33 (2010) 1456–1467.

[25] J. D. Teresco, K. D. Devine, J. E. Flaherty, Partitioning and dynamic load balancing for the numerical solution of partial differential equations, in: A. M. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, volume 51 of *Lecture Notes in Computational Science and Engineering*, Springer, 2006, pp. 55–88.

[26] E. D. Fernández-Nieto, M. J. Castro, C. Parés, On an intermediate field capturing Riemann solver based on a parabolic viscosity matrix for the two-layer shallow water system, Journal of Scientific Computing 48 (2011) 117–140.

[27] M. J. Castro, A. M. Ferreiro, J. A. García, J. M. González, J. Macías, C. Parés, M. E. Vázquez-Cendón, The numerical treatment of wet/dry fronts in shallow flows: applications to one-layer and two-layer systems, Mathematical and Computer Modelling 42 (2005) 419–439.

[28] NVIDIA, Tuning CUDA applications for Fermi, Version 1.5, 2011.

[29] A. L. Lastovetsky, J. J. Dongarra, High Performance Heterogeneous Computing, Wiley Series on Parallel and Distributed Computing, Wiley, First edition, 2009.

[30] I. T. Association, InfiniBand architecture specification, Volume 1, Release 1.2, 2004, http://www.infinibandta.org, Accessed February 2016.

[31] D. J. Miller, Giant waves in Lituya Bay, Alaska, Professional paper: United States Geological Survey, U.S. Government Printing Office, 1960.

[32] M. de la Asunción, M. J. Castro, J. M. González, J. Macías, S. Ortega, C. Sánchez, Modeling the Lituya Bay landslide-generated mega-tsunami with a Savage-Hutter shallow water coupled model, Technical Report, University of Málaga, 2013.