26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Structured Grid-based Parallel Simulation of a Simple DEM Model on Heterogeneous Systems

Alessio De Rango
Dept. of Math. & Comp. Science
University of Calabria
Rende, Italy
a.derango@mat.unical.it

Pietro Napoli
Dept. of Math. & Comp. Science
University of Calabria
Rende, Italy
pietro.napoli@mat.unical.it

Donato D'Ambrosio
Dept. of Math. & Comp. Science
University of Calabria
Rende, Italy
donato.dambrosio@unical.it

William Spataro
Dept. of Math. & Comp. Science
University of Calabria
Rende, Italy
spataro@unical.it

Alberto Di Renzo
Dept. of Envir. & Chemical Engineering
University of Calabria
Rende, Italy
alberto.direnzo@unical.it

Francesco Di Maio
Dept. of Envir. & Chemical Engineering
University of Calabria
Rende, Italy
francesco.dimaio@unical.it

***Abstract*—Here we present different preliminary parallel grid-based implementations of a simple particle system with the purpose to evaluate its performances on multi- and many-core computational devices. The system is modeled by means of the Discrete Element Method and the Extended Cellular Automata formalism, while OpenMP and OpenCL are used for parallelization. In particular, both the 3.1 and 4.5 OpenMP specifications have been considered, the latter also able to run on many-core computational devices like GPUs. The results of a first test simulation performed by considering a cubic domain with about 316,000 particles have shown a clear advantage of OpenCL on the considered Tesla K40 Nvidia GPU, while the OpenMP 3.1 implementation has performed better than the corresponding OpenMP 4.5 on the considered Intel Xeon E5-2650 16-thread CPU.**

***Index Terms*—Discrete Element Method, Cellular Automata, OpenMP, OpenCL, Heterogeneous Computing**

I. Introduction

Particle and granular systems are ubiquitous and can be found in various natural phenomena as well as several industrial processes. Differently than gases or liquids, flowing particles cannot be described by continuum models (e.g. Computational Fluid Dynamics), as they inherently fail to represent the discrete nature of the material. The Discrete or Distinct Element Method (DEM) [1] allows the dynamic simulation of particulate solids both under quasi-static and highly dynamic conditions, by tracking the motion of individual particles along the domain, i.e. within a Lagrangian framework.

Even if mesh-less approaches are usually adopted to implement DEM systems [2], we experimented an alternative approach based on the Cellular Automata computational paradigm [3]. A uniform grid based model was therefore developed, in which the space is subdivided in cubic cells, each one able to contain a limited number of particles. Collisions detection of a generic particle with the others is therefore limited to the central cell and its neighbors, which can in principle give rise to better performances. However, the major drawback of this alternative approach is the memory usage, needed to represent the cellular space.

Some preliminary implementations of the Cellular Automata model, which we will refer to as $CADEM$ in the following, were developed by considering the OpenMP and OpenCL APIs for Parallel Computing. OpenCL [4] is a low-level widely adopted solution for parallel heterogeneous computing. As a consequence, even if it is able to provide high computational performance and has the great advantage of the portability (OpenCL applications can run on a wide range of heterogeneous parallel computational devices), it generally requires a great developing effort during the parallelization process since it needs a significant reorganization of the serial code. In OpenMP 4.5 [5], at the contrary, a minimal effort is necessary, since it is based on the usage of a set of directives that can be added to the serial code, without the need to change the serial organization of the application, or to consider further complex data structures or explicitly manage data exchange between the host and the computational device (e.g. a GPU). A direct OpenMP implementation of $CADEM$was therefore developed by considering the 4.5 specification of the API, which is particularly interesting since it permits to exploit both multi-core CPUs and many-core devices, like Nvidia GPUs. Moreover, two additional versions were developed, both of them based on the OpenCAL computational library [6], which is particularly suitable for seamless parallel implementation of uniform grid Computational Fluid Dynamics (CFD) models. In fact, OpenCAL allows for the straightforward definition of Cellular Automata- and Finite Differences-based simulation models of complex systems, by also supporting Extended Cellular Automata (XCA) and all those computing paradigms based on regular computational grids. In particular, the OpenCAL-OMP and OpenCAL-CL components were taken into account, resulting in two $CADEM$implementations

Authors gratefully acknowledge the support of Nvidia Corporation for this research.

0-7695-6380-5/18/$31.00 ©2018 IEEE
DOI 10.1109/PDP2018.2018.00099

based on the OpenMP 3.1 and OpenCL 1.2 specifications, respectively. The first one is suitable for multi-core CPUs, while the second for a wide range of parallel heterogeneous devices, GPUs included. In the following, we will refer to the different implementations as $CADEM_{OMP_4}$ for the OpenMP 4.5 version, $CADEM_{OMP_3}$ for the OpenCAL-OMP, and $CADEM_{OCL_1}$ for the OpenCAL-CL versions, respectively. The paper is organized as follows: Section II briefly describes the considered DEM model and how it was modeled as a cellular automaton; Section III presents the different $CADEM$ implementations; Section IV describes the considered simulated system and the computational results obtained on the considered heterogeneous devices; eventually, Section V concludes the paper with a general discussion about the presented research and outlooks for possible future developments.

II. $CADEM$: A Simple Cellular Automata-based DEM Model

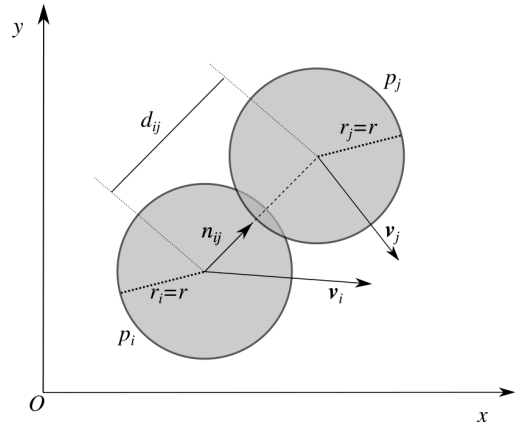

Fig. 1. Sketch of collision between two spherical particles, p_i and p_j, having the same radius r, in the physical model. The collision takes place when the distance between the particles' centers, d_{ij}, is lower than $r_i + r_j = 2r$. In such a case, a repulsive force, defined as in Equation 2, is applied to both particles along the normal direction $\boldsymbol{n}_{ij}$.

The $CADEM$ model is a Cellular Automata-based implementation of a simple three-dimensional DEM in which each particle is considered as perfect sphere of constant radius. Particles are subject to the gravity force and can collide with each other, besides with the domain boundaries. Only normal visco-elastic collisions are taken into account in this first version of the model (cf. Figure 1).

In the following part of this Section, DEM and XCA are briefly described, and eventually the $CADEM$ model formally defined.

A. DEM Models

As stated earlier, the Discrete Element Method requires individual particle trajectories to be resolved along the system. Generally, DEM models are available to include rotational effects, fluid drag force in multi-phase gas-solid or liquid-solid systems, particle-particle cohesive forces, etc. In the present simplified implementation, only the translational motion of the particles is calculated using Newton's second law of dynamics. Similar to the original formulation of Cundall and Strack [1], only gravity and contact forces arising from particle-particle and particle-wall contacts are considered. A soft-sphere approach is adopted to correctly handle simultaneous multi-particle and/or enduring contacts. Thus, for each particle considered we write:

$$m\boldsymbol{a} = m\boldsymbol{g} + \sum_{j=1}^{n_c} \boldsymbol{f}_{c,j}\,, \tag{1}$$

where m and $\boldsymbol{a}$ are the particle mass and linear acceleration, respectively; in the right hand-side, the total force is obtained by adding the sum of the contact forces $\boldsymbol{f}_{c,j}$ exerted by each j of the n_c bodies (i.e. partcles, walls) touching the particle considered to the gravity force.

During collisions, rather than taking micro-deformations into account, the integral solutions from classical mechanics are used, which relate the repulsive elastic forces to the macroscopic distance between the particle centers. In other words, the particles are assumed to keep their spherical shape while overlapping and the contact force is proportional to the degree of overlap. The contact forces at play during oblique collisions would require also tangential forces and deformations to be taken into account, but as previously mentioned the associated complexities (e.g. including effects of partial friction or total sliding) prevented the inclusion of such effects in this preliminary exercise.

On the other hand, realistic collisions including the restitution effect (e.g. as a result of plastic or visco-elastic deformations) are considered by means of a damping contribution in the calculation of the total force. When based on linear relationships, this corresponds to the linear spring-dashpot contact model. As demonstrated by [7] for single collisions, its simplicity is combined with a sufficient accuracy for dynamic systems. The overall implementation is based on the following simple formula:

$$\boldsymbol{f}_n = -k_n\,\boldsymbol{\delta}_n - \eta_n\,\boldsymbol{v}_{rn} \tag{2}$$

The material parameters are the spring constant k_n and the damping coefficient η_n. The first one determines the stiffness of the contact materials in the elestic regime; the second one influences the degree of energy dissipation by plastic deformation, giving rise to particles that bounce less (or more) depending on its high (or low) value. Both parameters can be related to data (Young's modulus, Poisson ratio, rebound height in particle drop tests) obtained by carefully designed experiments [8]. The other quantities are the relative displacement (overlap) between the contacting bodies $\boldsymbol{\delta}_n$ and their instantaneous relative velocity $\boldsymbol{v}_{rn}$.

Equation 2 applies during both particle-particle and particle-wall collisions. In the former case, the overlap is calculated by $\boldsymbol{\delta}_n = (r_i + r_j) - \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$, where $\boldsymbol{x}$ are the particle's centre positions and r their radii; the relative velocity is simply

$\boldsymbol{v}_{rn} = (\boldsymbol{v}_i - \boldsymbol{v}_j) \cdot \boldsymbol{n}_{ij}\ \boldsymbol{n}_{ij}$, in which $\boldsymbol{v}$ is the particle velocity. In the latter case, the force is a function only of the particle distance and velocity with respect to the wall.

The soft-sphere DEM requires the integration time-step to be a (small) fraction of the collision duration. By assuming purely elastic collisions to estimate such time τ, the following expression can be used to set the time-step size Δt:

$$\Delta t = \frac{\tau}{N_t} = \frac{\pi\sqrt{m/k_n}}{N_t} \tag{3}$$

where N_t is the number of time subdivisions, typically within the range 10–50.

Particle motion simulation is carried out by applying forward-projected integration formula to the equation of motion. Let $\boldsymbol{a}_i = \boldsymbol{f}_i/m$ be the acceleration of the i^{th} particle, then the new velocity, $\boldsymbol{v}'$, and position, $\boldsymbol{x}'$, are obtained by:

$$\begin{aligned} \boldsymbol{v}'_i &= \boldsymbol{v}_i + \boldsymbol{a}_i \Delta t \\ \boldsymbol{x}'_i &= \boldsymbol{x}_i + \boldsymbol{v}_i \Delta t \end{aligned} \tag{4}$$

B. Extended Cellular Automata

The Cellular Automata (CA) computational paradigm introduced a new approach in treating some complex systems, whose behaviour may be expressed in terms of local laws. Originally introduced by von Neumann in the 1950s to study self-reproduction issues, CA are particularly suited to model and simulate classes of complex systems characterized by a large number of interacting elementary components. The complexity of the system emerges from the interactions of its elementary (cellular) units, by applying relatively simple local rules. A CA can be intuitively considered as a d-dimensional space (the cellular space), partitioned into cells of uniform size. Each cell embeds an identical computational device: the finite automaton (f_a). Input for each f_a is given by the f_a's states located in the neighbouring cells. To this purpose, neighbourhood conditions have to be determined through a geometrical pattern, which is invariant in time and constant over the cells. At time $t = 0$, f_a's states define the CA initial configuration. The CA then evolves step by step by means of the f_a's transition function, which is simultaneously applied to each f_a. Despite their simple definition, CA can exhibit very interesting complex global behaviors. Moreover, from a computational point of view, they are equivalent to Turing Machines.

Among different fields, fluid-dynamics is one of the most important applications for CA and many different CA-based methods can be found in the literature to simulate fluid flows (e.g., [9]–[12]). Among them, Extended Cellular Automata (XCA) [13] represent an extension of the original CA computational paradigm and have proven to be suitable for simulating complex natural phenomena like lava flows (e.g., [14]–[16]) and debris flows (e.g., [17], [18]). XCA were firstly applied to the simulation of basaltic lava flows in the 80's [19] and many subsequent examples of application showed that the approach behind XCA can greatly make more straightforward the modeling of different (natural) complex systems.

Informally, XCA, compared to classical CA, are different because of the following properties that can be taken into account when evaluating the next state of a cell:

- Global operations, also known as *steering* operations, can be allowed (e.g. to model external influences or to perform reductions over the whole, or a subset, of the cellular space under consideration);
- A set of *parameters*, commonly used to characterize the dynamic behaviors of the considered phenomenon, can be defined;
- The cells state is decomposed in *substates*, each of them representing the set of admissible values of a given characteristic assumed to be relevant for the modeled system and its evolution (e.g., lava temperature, lava thickness, etc, in the case of a lava flow model). The set of states for the cell is simply obtained as the Cartesian product of the considered substates;
- As the cells state can be decomposed in substates, also the transition function can be split into elementary processes, each of them representing a particular aspect that rules the dynamic of the considered phenomenon. In turn, elementary processes can be split into *local interactions*, which refer to rules that deal with interactions among substates of the cell with neighbor ones (e.g. mass exchange with neighbors) and *internal transformations*, defined as the changes in the values of the substates due only to interactions among substates inside the cell (e.g. the solidification of the lava inside the cell due to temperature drop).

As a consequence, the natural phenomenon is modeled in terms of elementary processes, whose proper composition makes up the transition function of the automaton. By simultaneously applying the transition function to all the cells, the evolution of the phenomenon can be simulated in terms of modifications of the cell substates.

C. $CADEM$ Formal Definition

The $CADEM$XCA computational model is formally defined as:

$$CADEM =< R, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the three-dimensional domain over which the phenomenon evolves. The generic cell in R is individuated by means of a set of integer coordinates (i, j, k), where $0 \leq i < i_{max}$, $0 \leq j < j_{max}$, and $0 \leq k < k_{max}$.
- X is the three-dimensional von Neumann neighborhood relation (cf. Figure 2)
- Q is the set of cell states. Each cell is logically subdivided in *slots*, each of them representing a portion of space which can contain a particle. The number of particles that a cell can contain is evaluated by considering the maximum occupancy volume of a set of spheres having the same radius, according to Kepler's conjecture. Let N

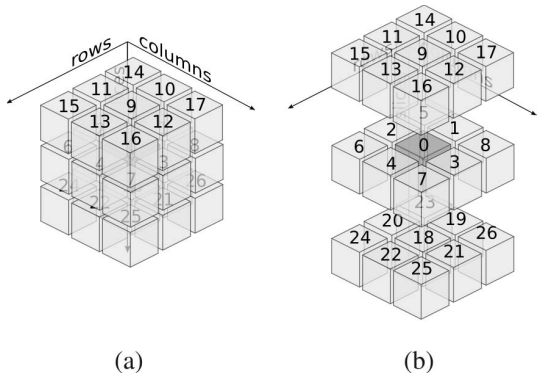

Fig. 2. The three-dimensional Moore neighborhood (a). Central cell is represented in dark gray, adjacent cells in light gray. A reference system is here considered to evaluate cells coordinates in terms of row, column and slice indices in a matrix-style representation, and a 0-based numerical identifier, which is better visible in (b), assigned to each cell in the neighborhood for straightforward access.

denote the number of cell slots, i.e. the maximum number of particles per cell, Q is subdivided in the following substates:

- $Q_{id} = \{BOUNDARY, VOID, PARTICLE\}^N$ is the set of values representing the slots state. The slot can in fact: 1) mark a piece of the boundary; 2) be void; 3) contain a particle;
- $Q_p = (Q_{p_x} \times Q_{p_y} \times Q_{p_z})^N$ is the set of values representing the position of the (at most N) particles inside the cell with respect to a 3D Cartesian coordinate system;
- $Q_F = (Q_{F_x} \times Q_{F_y} \times Q_{F_z})^N$ is the set of values representing the forces applied to the particles inside the cell;
- $Q_v = (Q_{v_x} \times Q_{v_y} \times Q_{v_z})^N$ is the set of values representing the velocities of the particles inside the cell.

The Cartesian product of the substates defines the overall set of states Q:

$$Q = Q_{id} \times Q_p \times Q_F \times Q_v$$

Accordingly, the slot state, q_s, is specified by the following tuple:

$$q_s = (q_{id}, q_{p_x}, q_{p_y}, q_{p_z}, q_{F_x}, q_{F_y}, q_{F_z}, q_{v_x}, q_{v_y}, q_{v_z})$$

and the cell state, q, by the q_s assignements, with $s = 1, 2, \cdots, N$.

- P is the set of parameters ruling the model dynamics:
  - p_{k_n} is the parameter specifying the contact stiffness;
  - p_η is the contact damping coefficient;
  - p_m is the particle mass;
  - p_r is the particle radius;
  - p_c is the cell side;
  - p_{N_t} is the number of time subdivisions for the Δt (cf. Equation 3).
- $\sigma : Q^{27} \rightarrow Q$ is the deterministic cell transition function. It is composed by the following elementary processes, listed below in the same order as they are applied:
  - $\sigma_1 : Q_{id} \times Q_F \rightarrow Q_F$ resets the applied forces to each particle in the cell to the gravity force. The force applied to boundary particles is initialized to zero and does not change during the simulation.
  - $\sigma_2 : Q_{id} \times Q_p \times Q_v \rightarrow Q_F$ evaluates changes in the applied forces due to collisions among particles inside the cell. In particular, a loop processes the generic particle p_i ($i = 1, 2, \ldots, N-1$) and, for each of them, evaluates the distances with respect to the others particles p_j of higher index ($j = i+1, i+2, \ldots, N$). If the distance d_{ij} between the two particles is lower than $2p_r$, then a collision occurs and the force is computed according to Eq. 2.
  - $\sigma_3 : (Q_{id} \times Q_p \times Q_v)^{27} \rightarrow Q_F$ evaluates changes in the applied force on the particles inside the central cell due to collisions with particles located in the neighbors. In particular, a loop processes the generic particle p_i ($i = 1, 2, \ldots, N$) and, for each of them, evaluates the distances with respect to the p_j ($j = 1, 2, \ldots, N$) particles in each neighbor. If the distance d_{ij} between the two particles is lower than $2p_r$, then a collision occurs and the force is computed for the particle p_i only as in Equation 2.
  - $\sigma_4 : Q_{id} \times Q_p \times Q_v \rightarrow Q_p \times Q_v$ the new positions, q'_{p_i}, and velocities, q'_{v_i} ($i = 1, 2, \ldots, N-1$), of the particles inside the cell are computed by applying Equation 4.
  - $\sigma_5 : (Q_{id} \times Q_p)^{27} \rightarrow Q$ assigns the particles to the cell, depending on the new position evaluated by the σ_4 elementary process. Specifically, the slots containing particles whose positions are outside the space region associated with the central cell are marked as available to receive another particle. Nevertheless, the particle currently in the slot is temporarily kept in memory, since it will be subsequently assigned to an available slot in a neighboring cell. At the end of this first phase, a second stage follows in which, if necessary, neighboring cells' particles are reassigned to the central cell according to their updated position.

III. The Different $CADEM$ Implementations

As already stated, different $CADEM$ parallel implementations were considered in this work, all of them adopting double precision floating point numbers. Some of them are based on the OpenCAL simulation library, namely $CADEM_{OMP_3}$ and $CADEM_{OCL_1}$, which are able to run on multi- and many-core devices, respectively. In addition, a further parallel version, namely $CADEM_{OMP_4}$, was developed by directly parallelizing the $CADEM$ serial implementation by means of OpenMP 4.5 APIs. In this latter case, the $CADEM$ serial implementation was still based on OpenCAL (as for the other parallel versions), but with the difference that only its lower-level components regarding buffers management were considered. This was necessary because the current OpenMP

4.5 support provided by the compilers (e.g. gcc and clang) ensures optimal speed-up performances with the simplest possible data structures and loops (cf. [20], [21]). This latter serial version was considered as reference for performance comparison. A static scheduling was considered for the CPU parallel implementations of CADEM, while a one-thread/one-cell policy for those implementations developed to run on many-core devices.

A. The OpenCAL parallel implementations of $CADEM$

Thanks to OpenCAL, the $CADEM_{OMP_3}$ and $CADEM_{OCL_1}$ implementations were straightforward. A `CALModel3D` model object was considered to define a three-dimensional computational domain where, for each cubic cell, the Moore neighborhood was considered (cf. Figure 2). No OpenCAL built-in specific optimizations, which can be considered at model object definition time, were adopted in these first implementations.

As regard particles modelling, an array of N substate objects was considered for each particle characteristic, like force or velocity, where N is the number of cell slots. For instance, the `CALSubstate3Dr Fx[N]` array of N substates was considered to model the x component of the force acting on the particles allocated into the N cell slots. A `calRunDef3D` simulation object was also considered for the $CADEM_{OMP_3}$ implementation, to transparently allow the simulation to be executed on OpenMP 3.1 compliant multi-core processors, while a `CALCLModel3D` device-side model object was defined in the case of the $CADEM_{OCL_1}$ implementation to transparently perform host to/from device data transfer. The device global memory was considered in this first $CADEM_{OCL_1}$ implementation, as well as in the $CADEM_{OMP_4}$ one. The `CALCLModel3D` device-side model object was also needed to run the simulation on an OpenCL compliant many-core device. A stopping criterion based on the time interval to be simulated was considered and the transition function's elementary processes defined in terms of OpenCAL callback functions and OpenCL kernels for the $CADEM_{OMP_3}$ and $CADEM_{OCL_1}$ implementations, respectively. As an example, the OpenCAL callback function implementing the σ_1 elementary process is shown below:

```
void sigma1(CALModel3D* dem,
                    int  i,
                    int  j,
                    int  k)
{
 CALreal F[3] = {0.0, 0.0, -PARTICLE_MASS*G};

 for (int s = 0; s < N; s++)
   if (calGet3Di(dem,ID[s],x,y,z) == PARTICLE)
   {
    calSet3Dr(dem,Fx[s],i,j,k,F[0]);
    calSet3Dr(dem,Fy[s],i,j,k,F[1]);
    calSet3Dr(dem,Fz[s],i,j,k,F[2]);
   }
}
```

In the above implementation of the σ_1 elementary process, `dem` represents the model object, while `i`, `j`, and `k` the generic cell's integer coordinates within the three-dimensional domain. The `s` index is used to access the `N` cell's slots, within which a particle can be found. The `F` array of `CALreal` values (i.e. double precision floating point numbers) is used to define the gravity force and then the cell's slots are checked. In case a particle is found (i.e. if the `calGet3Di()` query function returns the `PARTICLE` enumeral), the `Fx`, `Fy`, and `Fz` substates are set by means of the `calSet3Dr()` OpenCAL API function. Note that this elementary process does not contain any parallel code. Nevertheless, thanks to OpenCAL it is concurrently applied to the computational domain which, for this purpose, is transparently partitioned in uniform chunks. The OpenCL kernel implementation of the σ_1 elementary process is equivalent to the callback function shown above, with the difference that different OpenCAL API functions are adopted (e.g. the `calclSet3Dr()` function is used, instead of the `calSet3Dr()` one, to update values stored in the device's global memory, transparently to the user). The remaining elementary processes are implemented in a similar way by keeping the parallelism transparent to the user. Eventually, the simulation is run by means of the `calRun3D()` and `calclRun3D()` API functions for the $CADEM_{OMP_3}$ and $CADEM_{OCL_1}$ implementations, respectively.

B. The OpenMP 4.5 implementation of $CADEM$

According to OpenMP 4.5, the parallelization of $CADEM_{OMP_4}$ was based on `pragma` directives which, added to the serial code, allowed to both map data and run the computation on a compliant device (a Nvidia GPU in the specific case). Neither explicit low-level data mapping between host and device or CUDA/OpenCL kernels definition were required. However, a slight serial code reorganization was preliminary performed, which replaced the originally adopted OpenCAL high-level data structures with linear C buffers, since this is required by the current OpenMP implementation embedded in the adopted Clang compiler [20]. In particular, the $CADEM_{OMP_4}$ double precision substates were modeled by means of two linear buffers, `Q_current` and `Q_next`, which grouped the current and next buffers of all the $CADEM_{OMP_4}$ substates, respectively, and the `cal{G|S}etBuffer3DElement()`[1] functions adopted to explicitly access the buffer elements, in spite of the higher level `cal{G|S}et()` ones. The only integer substate was also split into two different buffers, which are `ID_current` and `ID_next`, respectively. Data was then straightforwardly mapped to the device memory before the beginning of the computation phase by means of the `enter data map` target, as shown below:

```
#pragma omp target \
    enter data map(to:ID_current[0:DIM])
#pragma omp target \
```

[1]The `{p1|p2|...|pn}` notation here defines a list of n mutually exclusive parameters.

```
    enter data map(to:ID_next[0:DIM])
#pragma omp target \
    enter data map(to:Q_current[0:DIM])
#pragma omp target \
    enter data map(to:Q_next[0:DIM])
```

In order to offload elementary processes to be executed on the compliant device, their prototypes were preliminary declared inside a `#pragma omp declare` target region, like in the following example:

```
#pragma omp declare target
void sigma1( CALint* ID_current,
             CALreal* Q_next
                 int  i,
                 int  j,
                 int  k);
#pragma omp end declare target
```

and therefore run by means of an OpenMP `distribute parallel for` target, as shown below for the case of the σ_1 elementary process.

```
#pragma omp target \
    teams distribute parallel for collapse(3)
  for (int i = 0; i < X_CELLS; i++)
    for (int j = 0; j < Y_CELLS; j++)
      for (int k = 0; k < Z_CELLS; k++)
        sigma1(ID_current, Q_next, i, j, k);
```

Here, in particular, the `teams` clause spawns a number of threads teams (i.e., CUDA blocks) with the same number of threads. It is OpenMP that sets both the number of teams and threads per team, depending on both the problem and the available hardware. However, these values can be explicitly set by using the `num_teams` and `thread_limit` clauses. In addition, the `collapse(3)` clause collapses the subsequent loops which process the whole domain, and the `distribute` directive distributes the iterations to the master thread of each team.

Eventually, at the end of the computation stage, results are copied back to the host from the device by means of the `update` target, as shown in the following example.

```
#pragma omp target \
    update from(to:ID_current[0:DIM])
#pragma omp target \
    update from(to:ID_next[0:DIM])
#pragma omp target \
    update from(to:Q_current[0:DIM])
#pragma omp target \
    update from(to:Q_next[0:DIM])
```

IV. Computational Results and Discussion

The different parallel implementations of $CADEM$ were built by considering different compilers. In particular, Clang version 4[2] (cloned by the Clang-ykt GitHub repository, commit 5340d) was adopted to build $CADEM_{OMP_4}$ and $CADEM_{OMP_3}$, while the OpenCL compiler embedded in the Nvidia nvcc compiler version 7.5 was considered[3] for the case of $CADEM_{OCL_1}$.

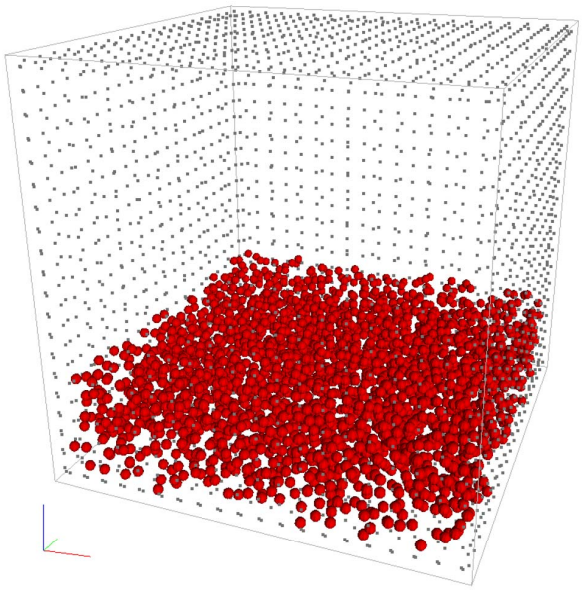

Fig. 3. A graphical representation of particle system similar to the one considered. The system here represented for illustrative purpose is composed by about 2600 particles, while the one considered in this work by 316,000 particles. Particles are initially randomly distributed in domain bottom, starting from the ground level up to the 20% of the overall domain height. Each particle is modeled as a sphere with 0.0005 meters radius. The cubic domain has a 0.2 meters side and is subdivided in a grid of $100 \cdot 100 \cdot 100 = 10^6$ cells, each one with 11 slots. A total of $11 \cdot 10^6$ particles can be therefore contained into the domain. The widget in the lower-left corner represents the adopted 3D coordinate system, where the red, green and blue segments point out the orientation of the x, y, and z axes, respectively. The origin coincides with the vertex of the cell located on the lower-left corner.

In order to evaluate the computational performances of the different implementations of $CADEM$, a simple particle system was considered, which is similar to the one shown in Figure 3 (this latter showing - for illustrative purposes - a configuration with few particles with respect to the one actually simulated). The particle system is constrained within a cubic domain of 0.2 m side. A set of 316,000 particles is initially randomly distributed in the bottom part of the domain, starting from the ground level, i.e. from $z = 0$ m, up to the 20% of the domain height, i.e. up to $z = 0.04$ m. The particles are positioned in a way so that there are no collisions. The system is therefore evolved for a total of 0.1 seconds, which is an amount of time sufficient to let all the particles reach the ground and to collide a sufficient number of times to properly evaluate the computational performances. The $CADEM$ parameters adopted for the simulation are listed in Table I. In particular, the choice of setting p_{k_n} to 8000

[2]The following flags were considered: `-O3 -ffast-math -fopenmp=libomp -Rpass-analysis -fopenmp-targets=nvptx64-nvidia-cuda -fopenmp-nonaliased-maps -ffp-contract=fast`

[3]The default nvcc flags were considered.

has determined a time-step equal to $1.118 \cdot 10^{-5}$ seconds. Accordingly, a total of 8945 steps were necessary to complete the simulation[4].

Two different devices were considered for simulating the above described dynamical system, namely an Intel Xeon 2.0GHz E5-2650 16-thread CPU and a Nvidia K40 GPU. The first one was considered for the $CADEM_{OMP_4}$ and $CADEM_{OMP_3}$ experiments, whose computational results are shown in Figure 4, while the second for the $CADEM_{OMP_4}$ and $CADEM_{OCL_1}$ ones, whose results are shown in Figure 5.

As regards the numerical correctness, the parallel simulations executed on the CPU perfectly matched the serial one, while those performed on the GPU evidenced a negligible difference in the particles' positions, which is of the 10^{-5} meters magnitude.

The speed-up registered on the CPU by the OpenMP 3.1 and OpenMP 4.5 implementations of $CADEM$(cf. Figure 4) showed a sub-linear speed-up increase trend according to the adopted threads. Here the OpenCAL-based implementation, i.e. the one based on OpenMP 3.1, exhibited slight better performances, probably due to the fact that the $CADEM_{OMP_4}$ implementation was mainly developed for execution on many-core devices, likely introducing an overhead when the code is run on the CPU.

A greater gap of $CADEM_{OMP_4}$ with respect to $CADEM_{OCL_1}$ is observed when the simulation is executed on the GPU, as is evident from Figure 5. Here, the reasons under the poor results obtained by $CADEM_{OMP_4}$ need to be better investigated. We think that one reason could be due to the not definitive implementation of the OpenMP 4.5 specifications embedded into the adopted Clang compiler that, for the considered model source code, is not able to properly translate some specific kernels written in the high level serial code into CUDA. In particular, `nvprof` profiling that was considered in tests has evidenced that one kernel, the σ_3 elementary process, takes about 75% of the total execution time, while the same process takes 40% on the OpenCL version. Nevertheless, the same version of the Clang compiler here considered was able to provide good performances on at least a different application [22]. As a consequence, the issue has certainly to be further investigated.

Eventually, as regards the obtained results in absolute terms, it is worth to note that the simulated system is characterized by a great load unbalance, since the particles are confined to the bottom 20% of the whole domain. Better performances could certainly be obtained in the case of uniform distribution or if a load balance algorithm is adopted.

V. Conclusions

In this work we have presented $CADEM$, a simple DEM model applied to the simulation of a system composed of about 316,000 particles in a cubic spacial domain. Particles are subject to gravity force and can collide with each other and

[4]The expression $0.1\sqrt{p_m/p_{k_n}}$ was adopted to evaluate the time-step.

TABLE I
The parameters adopted by the $CADEM$ simulation of the particle system of about 316,000 particles considered in this study. A graphical representation of a similar smaller system with about 2600 particles is given in Figure 3.

	Parameters					
	p_{k_n}	p_η	p_m	p_r	p_c	p_{N_t}
Values	8000	0.0015	0.0001	0.0005	0.002	10

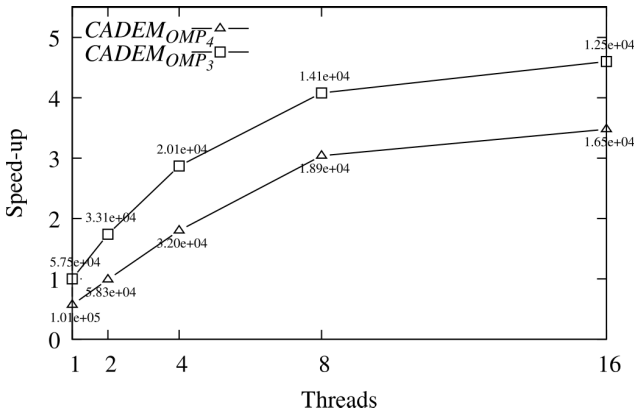

Fig. 4. Speed-up obtained by the $CADEM_{OMP_4}$ and $CADEM_{OMP_3}$ implementations of the $CADEM$model. Elapsed times in seconds are shown on top of each speed-up vertex. The considered case study is similar to the one shown in Figure 3, even if a total of about 316,000 particles was considered and the system was evolved for 0.1 seconds. The adopted CPU was an Intel Xeon 2.0GHz E5-2650.

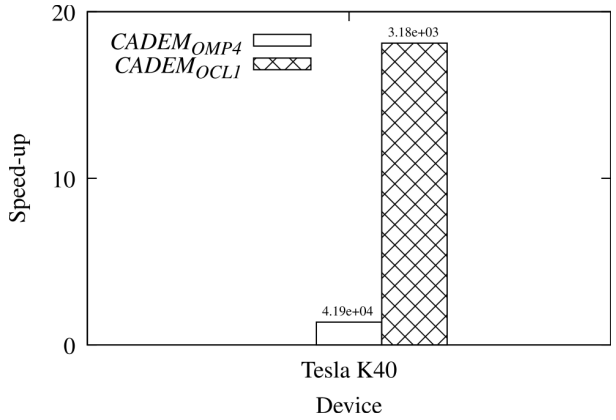

Fig. 5. Speed-up obtained by the $CADEM_{OMP_4}$ and $CADEM_{OCL_1}$ implementations of the $CADEM$model. Elapsed times in seconds are shown on top of each speed-up bar. The considered case study is similar to the one shown in Figure 3, even if a total of about 316,000 particles was considered and the system was evolved for 0.1 seconds. The adopted GPU was a Nvidia Tesla K40.

with the domain boundaries. In this preliminary model, only normal visco-elastic collisions have been taken into account. Three different implementations have been developed, each one based on the XCA formalism and characterized by a different parallelization approach. The 3.1 and 4.5 specifications of OpenMP and OpenCL version 1.2 have been considered for the different $CADEM$ parallelizations. In particular, the open source freely available OpenCAL parallel library was considered for the OpenMP 3.1 and OpenCL 1.2 implementations, since it provides a transparent support to these parallel computing APIs. According to the adopted modeling formalism, a uniform grid of cubic cells has been considered to model the spatial domain, resulting in a mesh-based implementation for each developed implementation. This choice was considered as a first step of a broader work in which a comparison with classical mesh-free implementations will be performed. Results have been evaluated on a 16 thread Intel Xeon CPU and a Nvidia K40 GPU. In particular the OpenMP 3.1 implementation has been tested one the CPU, the OpenCL one on the GPU, while the OpenMP 4.5 implementation on both devices. The OpenCL implementation has demonstrated to perform better, while the OpenMP 4.5 has been the one showing the worst performances in terms of speed-up. This could be due to the not mature implementation of the current OpenMP 4.5 specifications within the adopted Clang compiler, which is not able to fully optimize the CUDA code for the considered test case. This aspect will be investigated in detail in a future work and, besides considering other test cases, equivalent mesh-free implementations of $CADEM$ will be developed to compare them with the implementations here presented, both in terms of memory usage and computational performances. Moreover, once the next OpenCAL release will be completed, multi-node/multi-GPU implementations of $CADEM$ will be developed and, together with those here described, tested on different configurations in order to account for both computationally unbalanced and balanced particle systems.

REFERENCES

[1] P. A. Cundall and O. D. L. Strack, "A discrete numerical model for granular assemblies," *Géotechnique*, vol. 29, no. 1, pp. 47–65, jan 1979.

[2] H. Zhu, Z. Zhou, R. Yang, and A. Yu, "Discrete particle simulation of particulate systems: Theoretical developments," *Chemical Engineering Science*, vol. 62, no. 13, pp. 3378 – 3396, 2007, frontier of Chemical Engineering - Multi-scale Bridge between Reductionism and Holism.

[3] J. von Neumann, *Theory of Self-Reproducing Automata*, A. W. Burks, Ed. Champaign, IL, USA: University of Illinois Press, 1966.

[4] J. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engineering*, vol. 12, no. 3, pp. 66–72, 2010.

[5] OpenMP Architecture Review Board, "Openmp application program interface version 4.5," 2013.

[6] D. D'Ambrosio, A. De Rango, M. Oliverio, D. Spataro, W. Spataro, R. Rongo, G. Mendicino, and A. Senatore, "The Open Computing Abstraction Layer for Parallel Complex Systems Modeling on Many-Core Systems," *Journal of Parallel and Distributed Computing*, 2018, accepted under revision.

[7] A. Di Renzo and F. P. Di Maio, "Comparison of contact-force models for the simulation of collisions in DEM-based granular flow codes," *Chemical Engineering Science*, vol. 59, no. 3, pp. 525–541, 2004.

[8] F. P. Di Maio and A. Di Renzo, "Modelling particle contacts in distinct element simulations - Linear and non-linear approach," *Chemical Engineering Research Design*, vol. 83, no. A11, pp. 1287–1297, 2005.

[9] D. D'Ambrosio and W. Spataro, "Parallel evolutionary modelling of geological processes," *Parallel Computing*, vol. 33, no. 3, pp. 186–212, 2007.

[10] M. Avolio, S. Di Gregorio, and G. Trunfio, "A randomized approach to improve the accuracy of wildfire simulations using cellular automata," *Journal of Cellular Automata*, vol. 9, no. 2-3, pp. 209–223, 2014.

[11] G. Mendicino, J. Pedace, and A. Senatore, "Stability of an overland flow scheme in the framework of a fully coupled eco-hydrological model based on the Macroscopic Cellular Automata approach," *Communications in Nonlinear Science and Numerical Simulation*, vol. 21, no. 1-3, pp. 128–146, 2015.

[12] V. Ntinas, B. Moutafis, G. Trunfio, and G. Sirakoulis, "Parallel Fuzzy Cellular Automata for Data-Driven Simulation of Wildfire Spreading," *Journal of Computational Science*, 2016.

[13] S. Di Gregorio and R. Serra, "An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata," *Future Generation Computer Systems*, vol. 16, pp. 259–271, 1999.

[14] D. D'Ambrosio, G. Filippone, D. Marocco, R. Rongo, and W. Spataro, "Efficient application of GPGPU for lava flow hazard mapping," *The Journal of Supercomputing*, vol. 65, no. 2, pp. 630–644, 2013.

[15] D. Spataro, D. D'Ambrosio, G. Filippone, R. Rongo, W. Spataro, and D. Marocco, "The new SCIARA-fv3 numerical model and acceleration by GPGPU strategies," *The International Journal of High Performance Computing Applications*, vol. 31, no. 2, pp. 163–176, 2017.

[16] M. Oliverio, W. Spataro, D. D'Ambrosio, R. Rongo, G. Spingola, and G. Trunfio, "OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers," in *Procedia Computer Science*, vol. 4, 2011, pp. 271–280.

[17] D. D'Ambrosio, W. Spataro, and G. Iovine, "Parallel genetic algorithms for optimising cellular automata models of natural complex phenomena: An application to debris flows," *Computers & Geosciences*, vol. 32, no. 7, pp. 861–875, 2006.

[18] F. Luca', D. D'Ambrosio, G. Robustelli, R. Rongo, and W. Spataro, "Integrating geomorphology, statistic and numerical simulations for landslide invasion hazard scenarios mapping: An example in the Sorrento Peninsula (Italy)," *Computers and Geosciences*, vol. 67, no. 1811, pp. 163–172, 2014.

[19] G. M. Crisci, S. Di Gregorio, and G. Ranieri, "A cellular space model of basaltic lava flow," in *International AMSE Conference Modelling & Simulation*, ser. HPG '10. Aire-la-Ville, Switzerland: Group 11 "Terrestrial resources and phenomena", 1982, pp. 65–67.

[20] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Integrating GPU Support for OpenMP Offloading Directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:11.

[21] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 12–21.

[22] G. D. Balogh, I. Z. Reguly, and G. R. Mudalige, "Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs," *arXiv*, nov 2017.