Received: 20 August 2020 | Revised: 17 December 2021 | Accepted: 20 December 2021

DOI: 10.1002/cpe.6839

RESEARCH ARTICLE

WILEY

Parallel CPU–GPU computing technique for discrete element method

Vasyl Skorych | Maksym Dosta

Institute of Solids Process Engineering and Particle Technology, Hamburg University of Technology, Hamburg, Germany

Correspondence
Vasyl Skorych, Institute of Solids Process Engineering and Particle Technology, Hamburg University of Technology, Denickestrasse 15, 21073 Hamburg, Germany.
Email: vasyl.skorych@tuhh.de

Abstract

The efficiency of the simulations with the discrete element method (DEM) is significantly improved using a novel computational strategy. The new method is developed with a focus on platforms equipped with multi-core central processing units (CPU) and general-purpose graphics processing units (GPU). The DEM calculations are performed in parallel on the CPU and on the GPU using pre-calculated Verlet lists with a posteriori analysis of their consistency. The operations related to the search for possible contacts are performed on the CPU, whereas the processing of interactions, and integration of motion, are executed on the GPU. Performance analysis done for various types of tasks has shown that the new method allows to significantly decrease the average computational time and to utilize available computational resources more efficiently compared to the sequential CPU–GPU execution mode. Furthermore, due to more efficient calculations, the overall energy requirement for the proposed strategy does not exceed the demand for conventional sequential CPU–GPU computations.

KEYWORDS

CPU–GPU, discrete element method, GPU-DEM, hybrid computing

1 | INTRODUCTION

The discrete element method (DEM) is the most widely used numerical approach for the microscale modeling of granular materials. It was initially introduced by Cundall and Strack,[1] and in the last decades, a significant increase in interest in this method can be observed. The DEM was applied for a wide range of tasks in different areas,[2] starting from the standard problems of granular mechanics, like investigation of mixing behavior,[3] ending with modeling of agglomeration of enzymes,[4] or sintering of ceramics.[5] There exist various extensions of the DEM, such as the multisphere approach, the bonded particle method (BPM)[6–8] or the DEM coupling to computational fluid dynamics,[9,10] population balance models,[11,12] or finite element method.[13]

The central role in such widespread use of the DEM has played a significant improvement in the computational resources that took place in recent years. If in the early 90s an average simulation scene consisted of around 1000 particles, nowadays the average number of particles being modeled is about 200,000.[3] However, more and more examples can be found with a number over several[14,15] or even hundreds of millions[16] of objects.

An increase in the number of discrete objects, together with small time steps imposed by an explicit calculation scheme, resulting in a growth in the computational complexity of DEM calculations and, accordingly, lead to long simulation times. One of the easiest ways to cope with this problem

is to parallelize the computations. The relative simplicity of this approach stems from the nature of the DEM method: some of the basic operations, namely the calculation of forces between each pair of objects and the integration of motion for each object, are independent operations and therefore can easily be performed concurrently for each object or pair of objects.

Initially, the parallelization strategies in open-source or commercial DEM software packages were focused on the central processing units (CPU) implementation only. To speed-up calculations, a static or dynamic domain decomposition was applied, and calculations were executed on multi-processor or multi-core systems. For this purpose, the message passing interface (MPI), OpenMP interface, or hybrid approaches[17] were applied. However, due to the increase in the computational power of graphics processing units (GPU) and the development of application programming interfaces such as CUDA or OpenCL, more and more GPU-based DEM algorithms and software environments are emerging. For example, He et al.[18] have proposed a GPU-based implementation of a multi-grid algorithm for contact detection and applied it for modeling of compaction of particles with a wide size distribution. Kureck et al.[15] analyzed the behavior of non-spherical tablets during coating. To perform modeling of large-scale systems, combined MPI and GPU-based calculations were applied by different groups.[14,16,19]

One of the challenges related to the DEM simulations on the GPU is the efficient utilization of available computing resources. During the calculations on GPU, the CPU remains idle, and powerful multi-core processors are not used. To improve the efficiency of calculations, cooperative CPU–GPU computing techniques can be applied. Lee et al.[20] proposed a cooperative heterogeneous computing paradigm in which, depending on the distribution of the workload at runtime, a part of the initially GPU-belonging work is executed on the CPU. Wrede and Ernsting[21] developed a library for simultaneous CPU–GPU computing for data parallel skeletons. Navarro et al.[22] developed a partitioning strategy for parallel loops and applied it for particle simulations. A more comprehensive survey on different other techniques of heterogeneous computing and their applications can be found in Mittal and Vetter[23] and in Raju and Chiplunkar.[24]

Nowadays, in almost all GPU implementations of the DEM, most of the calculations, including a time-consuming contact detection,[14,18,25] are executed on the GPU. In some rare cases, hybrid CPU–GPU computing is performed, and a part of operations is transferred to the CPU.[9,10] Nonetheless, the DEM calculations are performed in the pipeline mode either on the CPU or on the GPU.

In this contribution, a new strategy for the concurrent usage of CPU and GPU for the DEM simulations is proposed. Compared to the previously used techniques,[20,21] this method is based on the high-level algorithmic parallelization of calculations.

2 | CALCULATION ALGORITHM

2.1 | The main principle of the DEM

The DEM is an explicit mesh-free approach for modeling granular materials. Each iteration of DEM can be subdivided into three main steps:

- Step 1: contacts detection;
- Step 2: calculation of interactions;
- Step 3: integration of motion.

In the first step, the positions of all objects (particles, droplets, walls, etc.) are analyzed, and interacting objects are identified. Due to a large number of modeled objects, this can be a very time-consuming task. Additional complexity arises when it is necessary to treat non-spherical particles, particles with a large size ratio,[26] or to consider periodic boundary conditions. Moreover, the contact detection between a particle and a wall is associated with additional complexity, since it is necessary to distinguish between three different types of contacts and to perform their post-processing.[27]

In the second step, the interactions between objects are calculated. There is a wide variety of contact models, starting from relatively simple linear interaction models consisting of several operations of multiplication or summation, ending with complex non-linear models.[8]

Compared to the contact detection and the calculation of interactions, the integration of motion in the third step is a relatively simple operation. Here new velocities and positions of objects are calculated.

It should be noted, that the three steps listed above are the most commonly used DEM scheme. However, depending on the type of problem being solved, additional steps can be included. For example, during the simulation of the breakage process, the positioning of fragments must be performed.[28,29]

2.2 | Contact detection algorithm

The novel parallelization strategy has been implemented in the component-based modeling framework MUSEN.[30,31] To perform contact detection, it applies a method based on the Verlet lists.[32] This is a widely used approach for particle-based simulations.[33,34] Its main idea is as follows: for each

particle P_i, a list of all neighboring particles or walls located within a certain cut-off distance (Verlet distance) L_v is found and stored. Thereafter, to detect all the contacts of P_i it is enough to take into account and analyze only the potential contacts from this list. Moreover, there is no need to update the list in each simulation time step, which allows reducing the amount of calculations. The condition for including a potential contact between particles P_i and P_j in the list is:

$$\left|x_i\left(t_v\right)-x_j\left(t_v\right)\right| \leq L_v + R_i + R_j \tag{1}$$

where R_i and R_j are the contact radii of particles, $x_i\left(t_v\right)$ and $x_j\left(t_v\right)$ are their positions at time point t_v, when the Verlet list is updated. The list of potential contacts is considered to be consistent at any time point $t > t_v$ if the following condition is satisfied:

$$\max\left\{\left|x_i(t)-x_i\left(t_v\right)\right|\right\} \leq L_v/2 \tag{2}$$

The frequency of updating the Verlet list is strongly dependent on the dynamics of objects and on the threshold value L_v. On the one hand, small values of L_v allow reducing the size of Verlet lists. On the other hand, increasing L_v decreases the updating frequency. Moreover, one of the additional limiting factors is the amount of available global GPU memory. To avoid time-consuming memory reallocation on the GPU on each iteration, memory is allocated once for all potential contacts each time the Verlet list is updated. As a result, for very large systems, L_v will be limited by the amount of available GPU memory. In the MUSEN framework, the default value of L_v is calculated as twice the minimum particle radius in the system.

To reduce computational costs and to avoid analysis between all possible pairs of objects, the linked-cell algorithm[35] was also applied. The whole simulation domain is divided into cubic cells of a known size and all particles are distributed among them according to their positions. The cell size L_C is defined depending on the maximum particle radius $R_{\max}$ and should be larger or equal to $2R_{\max} + L_v$. Due to this, to find all the possible contacts of P_i, it is enough to consider only the objects in the current and all adjacent cells. Moreover, to avoid duplicate comparison of the same cell pairs, only limited combinations can be processed, which reduces the number of cell pairs to be considered from 27 to only 14.[36]

The efficiency of the linked-cell algorithm can be negatively affected by the presence of particles with wide size distributions. Thus, to improve this method, it was extended by a multigrid approach.[26,37] Its main idea is to omit the evaluation of contacts between small particles on the coarse grids. The method implies the introduction of several grid levels j with cell sizes L_c^j varying from $2R_{\max} + L_v$ to $2R_{\min} + L_v$. On each level j, only the particles with certain sizes within the limits $U^j_{\max}$ and $U^j_{\min}$ are considered, where

$$U^j_{\max} = \left(L_c^j - L_v\right)/2 \tag{3}$$

$$U^j_{\min} = \left(L_c^j/2 - L_v\right)/2 \tag{4}$$

Each cell i on each level j maintains two lists of particles: the main list M_i^j for particles with $U^j_{\min} < R_i \leq U^j_{\max}$ and the secondary one S_i^j for particles with $R_i \leq U^j_{\min}$. Then considering two neighboring cells A and B, Equation (1) is evaluated for pairs $P\left(M_A^j\right) : P\left(M_B^j\right)$, $P\left(M_A^j\right) : P\left(S_B^j\right)$, and $P\left(S_A^j\right) : P\left(M_B^j\right)$. Thus, at each level, only contacts between pairs of "large-large" and "large-small" particles are processed. All "small-small" contacts are processed at the lower levels of the hierarchy with more suitable grid sizes.

A general idea of the particle contact detection algorithm is presented in Algorithm 1.

3 | PARALLELIZATION APPROACH

3.1 | Parallel CPU–GPU calculations

The contact detection using Verlet lists (Algorithm 1) involves intensive dynamic memory allocations, as well as a large number of heterogeneous operations. Both of these aspects are critical for GPU execution and may lead to a significant reduction in computing performance. It should be noted, that there exist alternative strategies of contact detection on GPU without generation of Verlet lists.[14,16,25] However, Verlet lists offer a set of significant advantages and, in this work, we propose a new simulation strategy based on their use.

The conventional way to perform Verlet-based DEM simulations on hybrid CPU–GPU architecture is to perform calculations sequentially, as is shown in Figure 1A. Here, the detection of potential contacts and the generation of Verlet lists is performed on the CPU (A_1). The generated lists are transferred to the GPU (A_2) as fixed-size arrays, which allows avoiding any memory reallocation on the GPU. In further steps, these lists are used for iterative calculations of interactions (A_4) and integration of motion. These calculations are repeated as long as the condition in Equation (2) is satisfied (A_3). Finally, the information about the coordinates of the objects (particles, walls, droplets, etc.) is transferred back to the CPU (A_5), where it is used to generate new Verlet lists, and the algorithm is repeated (A_1).

Algorithm 1. Schematic illustration of the contact detection algorithm

$L_c = 2 \cdot R_{\max} + L_v$;
forall *grid levels j* **do**
  $U^j_{\max} = L_c/2^{j+1} - L_v/2$;
  $U^j_{\min} = L_c/2^{j+2} - L_v/2$;
end for
while $t < t_{END}$ **do**
  if $\max_i |x_i(t) - x_i(t_v)| \geq L_v/2$ **then**
    forall *particles i* **do**
      forall *grid levels j* **do**
        if $U^j_{\min} < R_i < U^j_{\max}$ **then**
          $M^j \leftarrow i$;
        else if $R_i \leq U^j_{\min}$ **then**
          $S^j \leftarrow i$;
        end if
      end for
    end for
    forall *grid levels j* **do**
      forall *neighboring cell pairs c*1 **and** *c*2 **do**
        forall *particle pairs* $\left(i1 : M^j_{c1}\ \&\ i2 : M^j_{c2}\right)$ **and** $\left(i1 : M^j_{c1}\ \&\ i2 : S^j_{c2}\right)$ **and** $\left(i1 : S^j_{c1}\ \&\ i2 : M^j_{c2}\right)$ **do**
          if $|x_{i1}(t) - x_{i2}(t)| \leq (R_{i1} + R_{i2} + L_v)$ **then**
            Add pair $(i1, i2)$ to the Verlet list;
          end if
        end for
      end for
    end for
    $t_v = t$;
  end if
  Further DEM steps to calculate interactions and integrate motion;
  ...
end while

As it can be seen from Figure 1A, there are idle time intervals when the GPU is in the standby mode, waiting for new lists to be generated on the CPU. As a result, the efficiency of the sequential approach depends on the average clock time τ_{CPU}, needed to regenerate Verlet lists (A_1). If τ_{CPU} is much shorter than the average time τ_{GPU} needed to perform DEM calculations on the GPU (A_4), the sequential execution approach reveals high efficiency. However, if τ_{CPU} and τ_{GPU} have the same order of magnitude, the fraction of idle intervals increases and hence overall efficiency decreases.

To improve calculation performance, the parallel execution mode (Figure 1B) with two-stage instruction pipelining and task-level parallelism is proposed. The whole computational work is divided into two stages, which can be performed relatively independently: pre-calculating contacts in Verlet lists and using them to calculate the forces and motion of objects. In this case, the CPU–GPU system can be considered as a two-stage pipeline, where the first instruction is to calculate the Verlet list on the CPU, and the second one is to use it on the GPU. The pipelined commands processing allows maximizing the load of both computational units during the entire simulation time. Such a static distribution of tasks between the two devices eliminates complex synchronization concepts, making the algorithm more general and applicable for a wide range of problems.

The main idea of the algorithm is to start pre-calculation of new Verlet lists on the CPU (B_5) even before the DEM calculations with the current list ($B_{6,7}$) are finished. During the execution of the algorithm, the initial Verlet list Λ_1 is first calculated on the CPU (B_1), while the GPU is idle. Λ_1 is then passed to the GPU (B_2), after which the GPU immediately begins time integration ($B_{6,7}$). Meanwhile, the CPU starts a wait loop (B_3). In each simulation step, GPU analyzes the distance between the current spatial positions of the objects and their positions used to calculate Λ_1

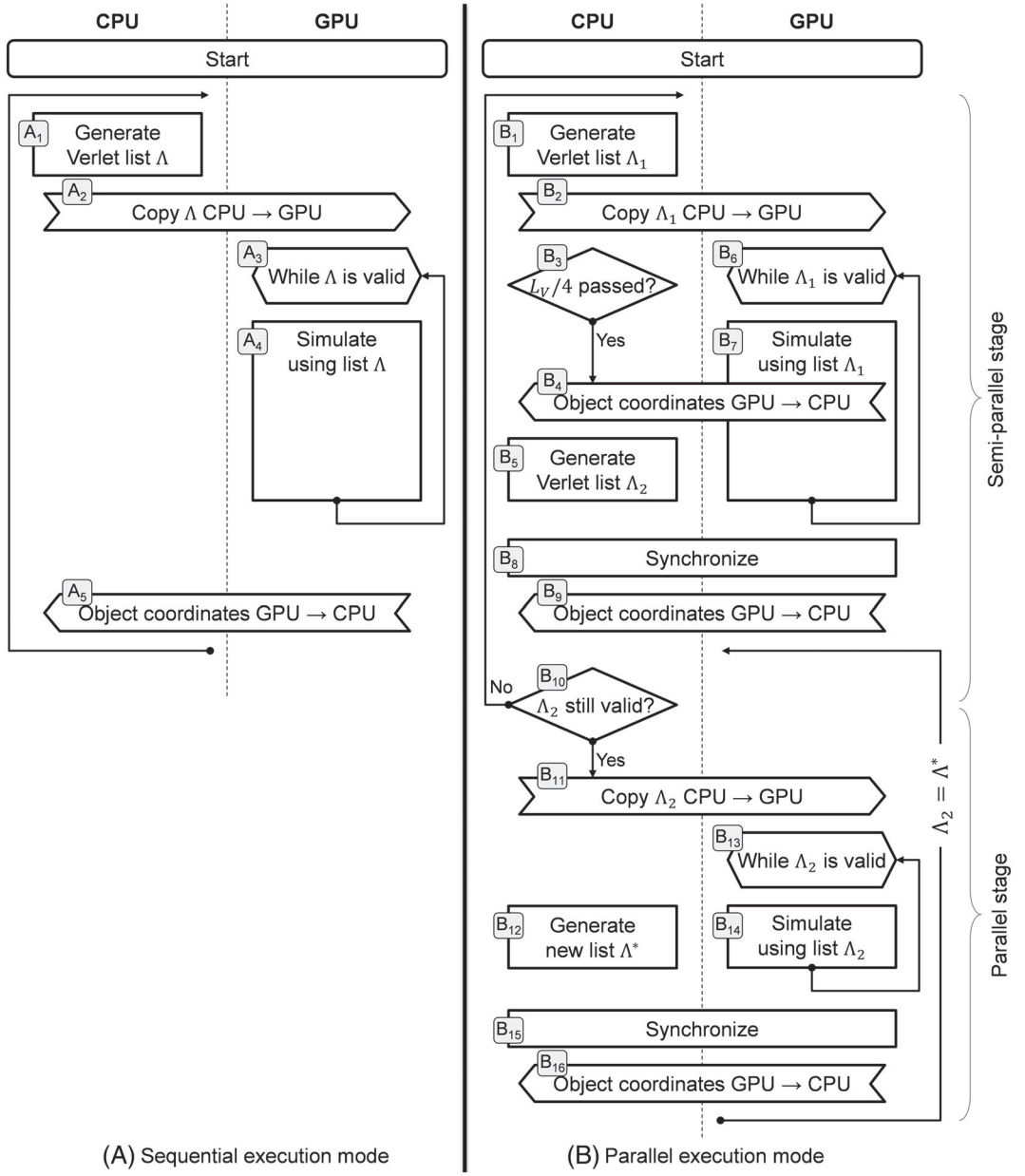

FIGURE 1 Sequential and parallel algorithms for hybrid CPU–GPU DEM simulations. CPU and GPU execution blocks shown next to each other run in parallel

(Equation 2). As soon as the distance for any particle exceeds a quarter of the Verlet distance ($L_v/4$), the current positions of the objects are synchronously copied to the CPU (B_4), the CPU exits the wait loop (B_3), and starts to calculate the next Verlet list Λ_2 (B_5). At the same time, the GPU continues the time integration loop for Λ_1 ($B_{6,7}$), so this part of the algorithm is executed in parallel (B_5 and $B_{6,7}$). Synchronization step (B_8) is needed since the calculation of the Verlet lists on the CPU and time integration on the GPU are executed independently and in general, do not complete at the same time.

At this stage, precalculated Λ_2 may already be inconsistent with the new coordinates of the objects. Therefore, after data transfer (B_9), an additional analysis is needed to verify whether Equation (2) is met for Λ_2 (B_{10}). If Λ_2 is not consistent, then the semi-parallel stage of the algorithm ($B_{1\text{-}10}$) starts from the beginning, where the calculation of the new Verlet list will be based on the current positions of the objects (B_1).

On the other hand, if Λ_2 is still valid, a parallel stage of the algorithm starts. Λ_2 is transferred to the GPU (B_{11}), where the calculations proceed using Λ_2 ($B_{13,14}$). Parallel to this, the CPU starts the calculation of the next list Λ^* (B_{12}), so (B_{12}) and ($B_{13,14}$) are executed concurrently. Here one should pay attention to the fact that the calculation of the Verlet list Λ_2 is performed for the state of the system in the middle of the calculations on

Λ_1 (moment (B_4) on the diagram). Thus, by the time of switching to the list Λ_2 ($B_{9\text{-}11}$), objects on the scene may have already traveled some distance relative to their position at point (B_4). Because of this, condition (B_{13}) may cease to be fulfilled earlier than it was for ($B_{6,7}$), which means that the GPU calculations in the parallel stage ($B_{13,14}$) of the algorithm may be shorter compared to the semi-parallel stage ($B_{6,7}$). In the case when the average speed of objects on the scene does not change over time, the difference between them will be about 2 times.

Once the calculations using Λ_2 are finished, synchronization (B_{15}) and data transfer (B_{16}) occur. After that, condition (B_{10}) is checked again, and, depending on it, either the semi-parallel ($B_{1\text{-}10}$) or parallel ($B_{10\text{-}16}$) part of the algorithm is repeated. It should be noted that steps ($B_{8,9}$) and ($B_{15,16}$) are equivalent and the latter were added to the diagram only for better readability.

As it was mentioned above, the efficiency of one or another execution mode depends on the average time needed for the contact detection on the CPU (τ_{CPU}) and for the remaining part of DEM operations on the GPU (τ_{GPU}). Both of these characteristics are influenced by many factors, and their a priori estimation is a very challenging task. They are highly dependent on the hardware resources, on the type of problem being solved, and even on the state of the simulated scene at any given moment. Let q be a ratio calculated as:

$$q = \frac{\tau_{CPU}^{seq}}{\tau_{GPU}^{seq}} \tag{5}$$

where τ_{CPU}^{seq} and τ_{GPU}^{seq} are the average clock time needed for CPU and GPU calculations in the sequential execution mode (Figure 1A). As stated earlier, in the ideal case, the number of the GPU simulation time steps at each parallel stage will be 2 times less compared to the sequential execution mode. At the same time, the number of calculations of the Verlet lists doubles, but the time of their operation does not change. Also, the number of data exchanges doubles. Thus, to calculate the same number of time steps in both modes, the following holds:

$$\begin{aligned} \tau_{CPU}^{seq} &= \tau_{CPU}^{par} = \tau_{CPU} \\ \tau_{GPU}^{seq} &= 2\tau_{GPU}^{par} = \tau_{GPU} \\ \tau_{exch}^{seq} &= 0.5\tau_{exch}^{par} = \tau_{exch} \end{aligned} \tag{6}$$

where τ_{exch} is the time needed to transfer the calculated Verlet list from the CPU to the GPU (A_2, B_2, B_{11}), plus the time it takes to copy the object coordinates back to the GPU (A_5, B_4, B_9, B_{16}).

As a result, two cases can be distinguished:

1. $q \leq 0.5$—application of the parallel execution mode fully loads the GPU;
2. $q > 0.5$—application of the parallel execution mode fully loads the CPU.

The differences between these two cases are demonstrated in Figure 2, where the exemplary time diagrams for both calculation modes and for two different hardware configurations are shown schematically. It is assumed that independent of the current state of the system, the following conditions are met: the time needed to generate Verlet list on the CPU (τ_{CPU}) is constant and the GPU calculation time is directly proportional to the Verlet distance L_v. For clarity, the diagrams show the case when objects move at a constant speed—then τ_{GPU} in parallel execution mode is reduced by exactly 2 times compared to the sequential mode. In real simulations, this reduction will depend on the dynamics of the process. For the assumed ideal case, τ_{CPU} (B_{12}) does not depend on the algorithm and therefore remains constant. Moreover, to simplify the representation, the time required for data transfer and for other supplementary operations is not included.

The first two timelines in Figure 2 depict the differences between the sequential and the parallel approaches, for the case when $q = 0.25$. The calculations are started on the CPU with the generation of the first Verlet list Λ_1, which is then transferred to the GPU (operations A_1 and A_2 in Figure 1). The GPU uses it to perform the simulation while the condition in Equation (2) is satisfied and this list remains valid ($A_{3,4}$). Only after that, the new coordinates of the objects are copied to the CPU (A_5), which updates the Verlet list Λ_2 (A_1). These operations are repeated until the end of the simulation. It can be seen that with this approach, the GPU resources are periodically idle while the CPU is busy generating new lists of potential contacts.

The parallel algorithm starts exactly the same with (B_1), (B_2), ($B_{6,7}$). But in the middle of the first iteration of the GPU, when the condition B_3 is satisfied, the transfer of the current positions of objects is initiated (B_4). After the transfer is finished, the CPU immediately starts calculating the next Verlet list Λ_2 (B_5). Thus, both computing devices work for some time in parallel: the GPU calculates the remaining interval ($B_{6,7}$), and the CPU prepares a new list of possible contacts (B_5). In the case of $q < 0.5$, the CPU ends its work earlier and waits for synchronization with the GPU. Thus, at the end of the GPU calculations, which are based on the Verlet list Λ_1, the following list Λ_2 is already calculated. At this point, data exchange takes place ($B_{9,11}$): the last calculated coordinates of the objects are transmitted to the CPU, and if condition (B_{10}) is satisfied, the new list is transferred to the GPU. If the condition from Equation (2) is fulfilled, both devices can continue working in parallel: the CPU prepares the following Verlet list

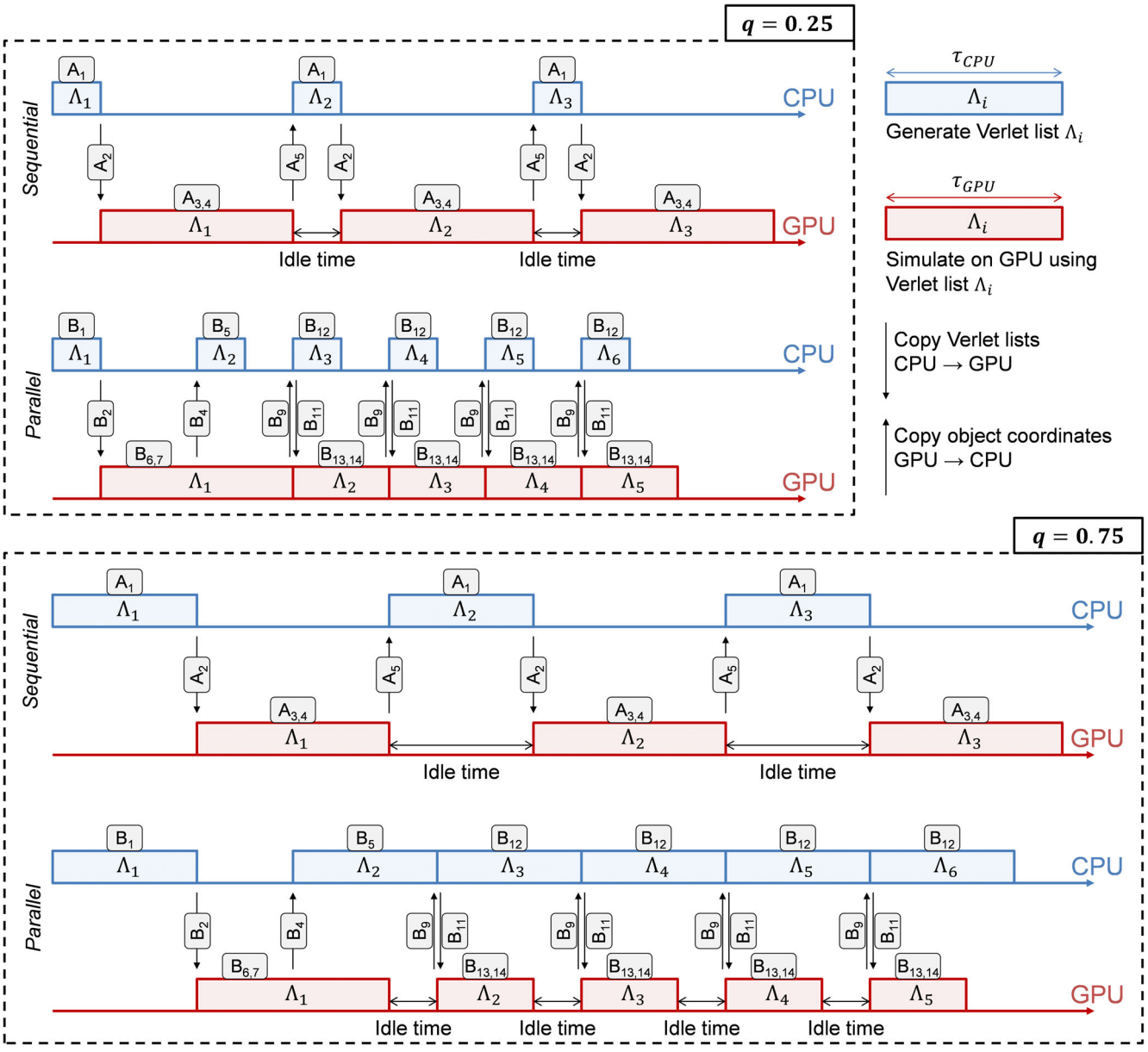

FIGURE 2 Exemplary timing diagrams of sequential (1, 3) and parallel (2, 4) algorithms for two cases with $q = 0.25$ and $q = 0.75$. Labels of operations correspond to those shown in Figure 1

Λ_3 (B_{12}), and the GPU performs integration based on the current one Λ_2 ($B_{13,14}$). Since the positions of objects from the middle of Λ_1 interval were used to calculate the current list of possible contacts Λ_2, the current GPU calculation cycle will be shorter than the initial one.

When both devices complete their iterations, the algorithm repeats. It can be observed that by applying this algorithm ($q < 0.5$) one can ensure that the GPU is never idle, but is constantly busy with computing operations. If during the data transfer ($B_{9,11}$) it turns out that the condition from Equation (2) is not satisfied, the timing diagram resets into the initial state.

If $q = 0.75$, the sequential approach works the same as in the previous case, but the overall simulation time is longer since the detection of contacts takes more time (Figure 2). The main difference of the parallel strategy is that the calculation of the new Verlet (B_5) cannot be finished before the end of the integration step on the GPU ($B_{6,7}$). Thus, only a partial overlap of the activity periods of the CPU and the GPU is possible. The GPU idle intervals remain, but their size is reduced. Nevertheless, the simulation time is reduced compared to the sequential approach, since the CPU idle time is eliminated.

Define performance gain from using the parallel execution mode G as

$$G = \frac{\tau_{\text{seq}}}{\tau_{\text{par}}} - 1 \tag{7}$$

where $\tau_{seq} = \tau^{seq}_{CPU} + \tau^{seq}_{GPU} + \tau^{seq}_{exch}$ is the execution time of one CPU–GPU stage in the sequential mode and τ_{par} is the time required to calculate the same number of simulation steps in the parallel mode. Using (6) it turns to

$$G = \begin{cases} \frac{\tau^{seq}_{CPU}+\tau^{seq}_{GPU}+\tau^{seq}_{exch}}{2\tau^{par}_{GPU}+\tau^{par}_{exch}} - 1 = \frac{\tau_{CPU}+\tau_{GPU}+\tau_{exch}}{\tau_{GPU}+2\tau_{exch}} - 1, & q \le 0.5 \\ \frac{\tau^{seq}_{CPU}+\tau^{seq}_{GPU}+\tau^{seq}_{exch}}{2\tau^{par}_{CPU}+\tau^{par}_{exch}} - 1 = \frac{\tau_{CPU}+\tau_{GPU}+\tau_{exch}}{2\tau_{CPU}+2\tau_{exch}} - 1, & q > 0.5 \end{cases} \tag{8}$$

Since the semi-parallel stage usually needs to be performed very rarely, it is neglected here. If $\tau_{exch} \ll \tau_{CPU} + \tau_{GPU}$, the time needed for data exchange can also be neglected. Then using (5), the maximum theoretical gain G_{max}, which can be reached by migrating from the sequential to the parallel execution mode, can be estimated as:

$$G_{max} = \begin{cases} \frac{\tau_{CPU}+\tau_{GPU}}{\tau_{GPU}} - 1 = q, & q \le 0.5 \\ \frac{\tau_{CPU}+\tau_{GPU}}{2\tau_{CPU}} - 1 = \frac{1-q}{2q}, & q > 0.5 \end{cases} \tag{9}$$

Thus, even under ideal conditions, the maximum theoretical gain is 50%, and it is reached at $q = 0.5$. The achieved G must decrease with an increase in τ_{exch} and with any change in q. Moreover, the performance gain turns even negative if $q > 1$, meaning that the algorithm loses its effectiveness if $\tau^{seq}_{CPU} > \tau^{seq}_{GPU}$.

Worth noting that q, despite depending on the case being simulated, is not a constant intrinsic characteristic of that case. It depends on many factors and changes over time during the simulation. Therefore, q and hence G und G_{max} are defined for only one CPU–GPU stage and are constantly changing. Nevertheless, in real calculations, G_{max} can be measured (possibly averaged over a period of time) and effectively used as a criterion for the applicability of the algorithm in each specific case at any time point of the simulation.

3.2 | Parallel calculations on CPU

Almost all calculation steps of the main algorithm depicted in Figure 1 are parallelized according to the shared memory model for a multi-core CPU (A_1, B_1, B_5, B_{12}) and for a general-purpose GPU (A_4, B_7, B_{14}). On the CPU, a multithreading software design pattern *Thread Pool* for automatic parallelization has been used. The implementation is based on the *std::thread* class from the C++ standard library and allows efficient parallelization of loops. During the initialization of the program, a set of threads equal to the number of available processors *std::thread::hardware_concurrency()* is created. Afterward, during the entire execution of the program, this collection of threads is maintained. The designed *ThreadPool* class has a task queue that is populated with jobs by the main thread. These jobs are executed in parallel by all threads using static load balancing, where the load is distributed only depending on the total amount of tasks. The proposed parallelization approach is designed to run on general-purpose personal computers, which have a relatively small number of computing cores. Besides, the parallel tasks are usually computationally homogeneous, and their number directly depends on the number of modeled objects and usually exceeds the number of cores by several orders of magnitude. Therefore, dynamic load balancing is not required and is not performed in the current implementation of *ThreadPool*.

Applied to the proposed algorithm, two main operations during the calculation of Verlet lists are performed in parallel using *ThreadPool*. These are the distribution of particles over the grids and the contact detection between particles in adjacent grid cells.

Verlet list Λ_2 and Λ^* must be calculated by the CPU in parallel with the GPU. Since a function that evaluates the list should not return a value, std::thread is also used to execute it. When it is required to start the calculation, a std::thread object is created with a callable (a function for calculating the Verlet list), associated with a tread of execution, and immediately started asynchronously with respect to the GPU.

3.3 | Parallel calculations on GPU

The Verlet list, calculated on the CPU, contains a list of potential contacts that can take place but are not necessarily active at the current time point. During the simulation (Figure 1: A_4, B_7, B_{14}), the number of active contacts is constantly changing. However, to avoid frequent memory reallocations, a predefined data structure with the information about collisions is created for each potential contact. Regardless of whether a particular contact takes place or not, such structure is maintained until the next recalculation of the Verlet list. This structure consists of:

- *pre-calculated parameters*: some parameters that do not change during contact, like equivalent Young's modulus;
- *incremental values*: values calculated in previous steps that should be maintained for further calculations, like tangential overlap;

- *results*: data fields in which simulation results are stored.

When the data structures with possible contacts are generated, the calculation of interactions and integration of motion starts. Here, three consecutive operations are performed in each step:

- *active contacts detection*: the Verlet list with all potential contacts is analyzed and, depending on the current object coordinates, a shorter list of currently active collisions is created;
- *calculation of particle-particle or particle-wall interactions*: for each contact, the corresponding contact model is executed;
- *calculation of solid or liquid bonds*: the corresponding contact model is executed for each bond;

The GPU calculations are implemented using the CUDA computing platform. For the parallel execution of each calculation step, a constant number of blocks and threads per block is specified to run the computational kernels. Both values are usually defined depending on the GPU architecture. For all GPU configurations used in this study, the number of blocks was set equal to the number of multiprocessors available on the device (cudaDeviceProp::multiProcessorCount), and the number of threads per block was set to 256.

All kernels run on a single stream and therefore execute synchronously with respect to the GPU. The data copy operations between the compute units are synchronous (cudaMemcpy) from the point of view of the main CPU thread. To organize data exchange, pinned memory (allocated with cudaMallocHost) is used.

4 | RESULTS

4.1 | Investigated case studies

To analyze the efficiency of the new approach, several DEM simulations have been performed. Overall, 12 different case studies have been analyzed. Figure 3 provides an overview of all case studies. Here the numbers of modeled discrete objects consisting of spherical particles (N_p), triangular walls in surface meshes (N_w), and solid bonds (N_b) are given.

In the first case study, the dynamics of grinding balls and grinding medium in a ball mill was simulated. In the last decades, the DEM was widely applied for this type of apparatuses.[38] In the second case study, the behavior of a material in a high shear granulator[39] is investigated. Case 3 represents the dynamic impact of a large particle on a granular bed of loose particles. Three different types of mixers are simulated in case studies 4, 5, and 6. A detailed description of scenes 4 and 5 can be found in Dosta et al.,[40] Lee et al.,[41] and McGuire et al.[3] In Case 6, the free-fall motion of particles in a static mixer is modeled. Studies 7 and 8 investigate the process of silo emptying. In Case 7, spherical particles, and in Case 8, half spheres are modeled. To represent the non-spherical shape of the particles, the bonded-particle model (BPM) was used,[8] where each particle of complex shape is represented as a set of primary particles connected with solid bonds. In Case 9, the vertical vibrations of rod-like particles in a cylindrical container are analyzed. Here the BPM is also used to represent the shapes of objects. The three points bending test applied to biopolymer aerogels is modeled in case study 10.[6] In Case 11, a random motion of particles in a cubic volume with periodic boundary conditions[42] was studied. The last case study 12 simulates the process of filling the conical container with particles under the influence of gravitational force.

All the investigated case studies consisted of a relatively large number of discrete objects of the order of 10^5 and more. Using the developed parallel CPU–GPU strategy, as well as the general application of the GPU, for scenes with a small number of particles is inefficient. Simulations consisting of less than 10^4 objects can usually be more effectively calculated on a multicore CPU rather than on a GPU.

4.2 | Performance analysis

To get reliable statistics, the case studies shown in Figure 3 have been modeled for a relatively long interval of process time on five different hardware configurations. The simulation end time has been chosen so that the calculation clock time on HD1 was at least 1 hour. Each simulation has been repeated 10 times on each configuration. In the scope of this work, the following hardware was used for running case studies:

- HD1: Intel Core i7-7700K, NVidia GeForce GTX 1080 Ti
- HD2: Intel Core i7-6800K, NVidia GeForce GTX 1080 Ti
- HD3: Intel Core i7-7820X, NVidia GeForce GTX 1080 Ti

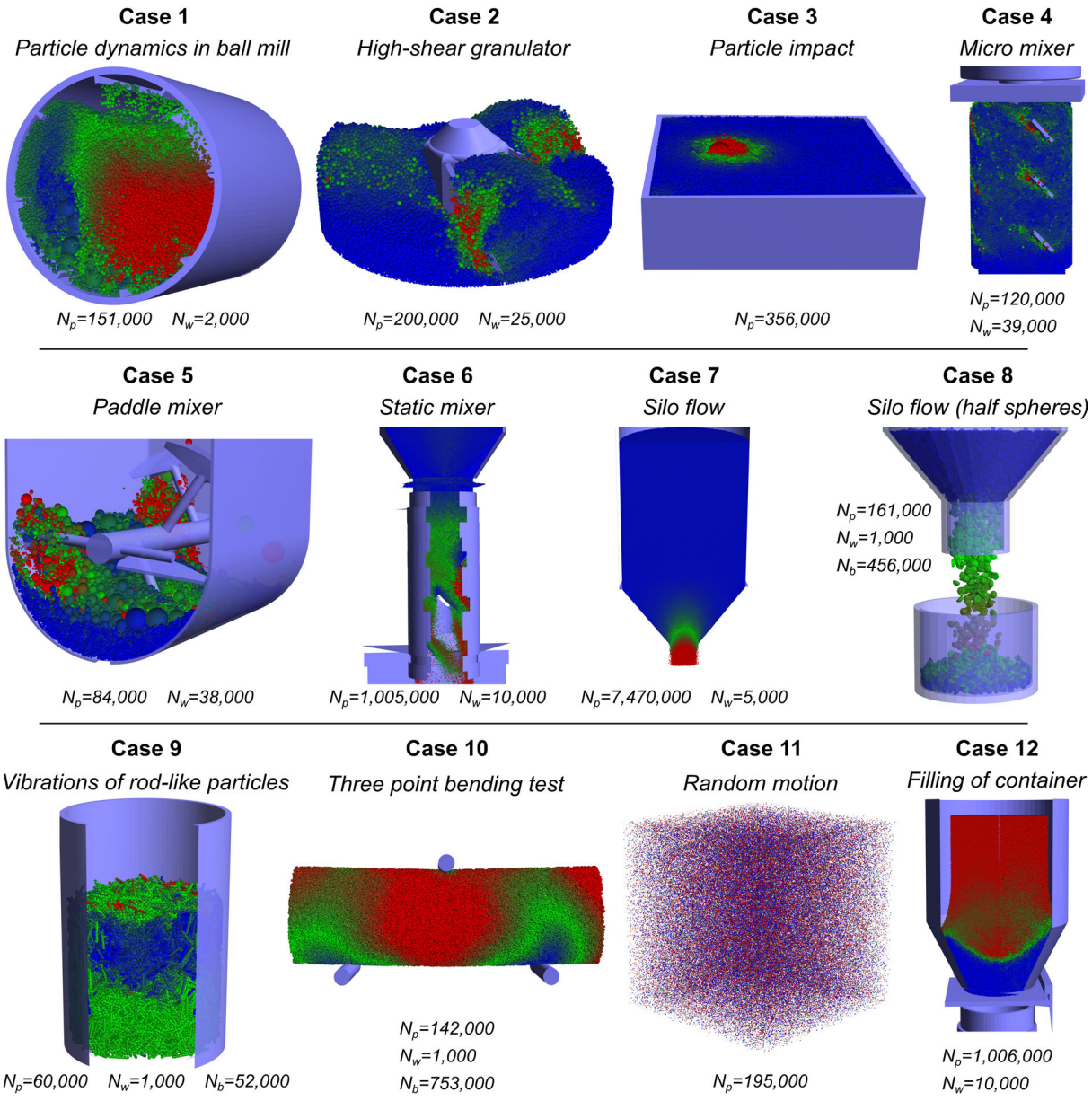

FIGURE 3 Investigated case studies. In all figures, particles are colored according to their translational velocity (red—high velocity, blue—low velocity)

- HD4: AMD Ryzen 9 3900X, NVidia TITAN RTX
- HD5: Intel Core i7-4790K, NVidia GeForce GTX 970

All CPU and GPU were operated at their reference frequencies in all configurations; all GPU were connected via PCIe 3.0 ×16 interface.

To carry out the measurements described below, the source code was instrumented by measuring the execution time of specific code sections using the std::chrono::steady_clock::now() function from the C++ standard library.

Figure 4 shows the performance gain obtained by using the new strategy for different case studies. Due to a large number of discrete objects and the limited memory capacity of NVidia GTX 970, Cases 6, 7, and 12 were not simulated on HD5. From the obtained results, it can be seen, that performance gain strongly depends on the type of the problem being solved, on the number of primary particles, on their dynamics and so forth and, as a consequence, on the ratio between τ_{CPU} to τ_{GPU} (Equation 5).

Several groups can be distinguished depending on the performance gain. So, Cases 4, 9, 10, and 11 show the worst performance gains close to zero. On some configurations, the execution time even slightly increased compared to the sequential mode. For example, parallel CPU–GPU simulation of vertical vibrations of rod-like particles (Case 9) does not lead to performance improvement over the sequential mode: the average

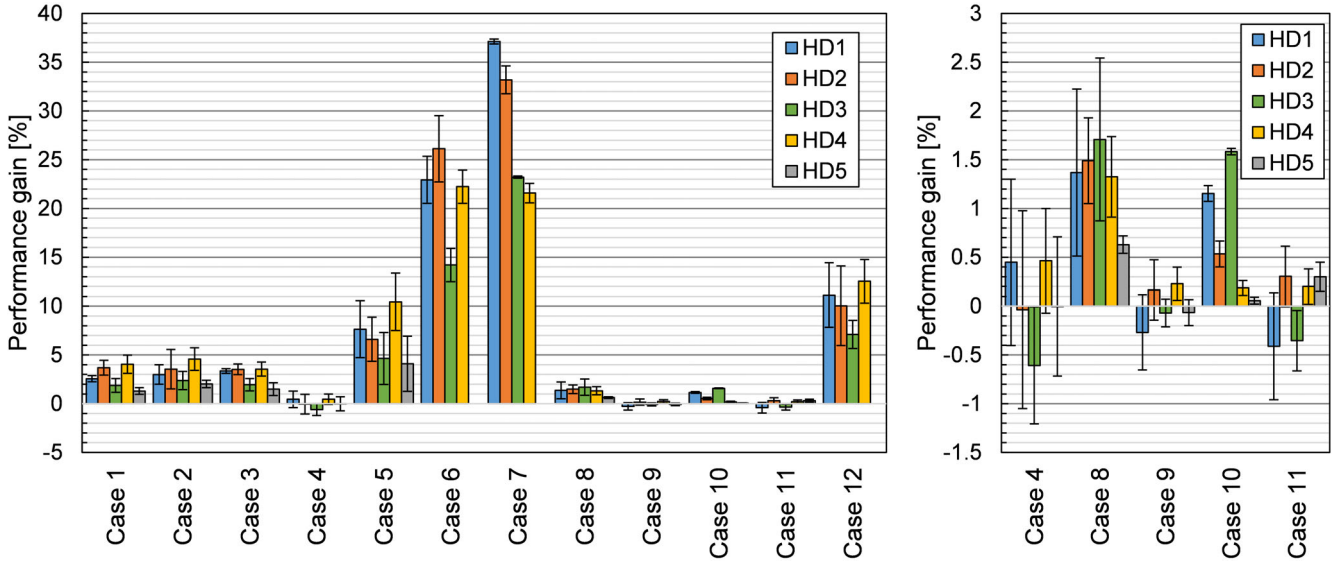

FIGURE 4 Performance gain, obtained through the usage of the parallel CPU–GPU calculation strategy for all cases (left) and selected cases with small performance gains on a finer grid (right)

performance gain for all configurations is either about zero or even negative. The average time ratio q (Equation 5) and the maximum theoretical gain G_{max} (Equation 9) for this study on all hardware configurations are equal to 0.56%.

Another group (Cases 1, 2, 3, 8) reveals a noticeable, but small improvement of up to 5%. For example, the concurrent CPU–GPU calculation of the ball mill filled with 150,000 particles (Case 1) on different architectures allows reaching an average gain of 2.7%. The maximum performance gain of 4.05% is obtained on the configuration HD4, and the minimum of 1.3% is on HD5. The deviations are caused mostly by different performance ratios of CPU and GPU and, as a result, different values of q. A much larger improvement can be observed for case studies 5, 6, 7, and 12, where due to a large number of discrete objects, the average gain reaches 6.67%, 21.37%, 28.77%, and 10.2%, accordingly. Simulation of a large densely packed bed (Case 7) on HD1 allows the best performance gain of 37.13% to be achieved. Here, the calculation time of Verlet τ_{CPU} is comparable to the rest of the DEM operations on GPU τ_{GPU}, and G_{max} equals 41.2%.

The results obtained generally indicate that the new algorithm makes it possible to significantly improve computational performance. However, the improvement is highly dependent on various conditions, and in some cases, the new method can even lead to a performance drop. Therefore, a prerequisite for its efficient use is an online time measurement and automatic switching between sequential and parallel execution modes.

To ensure the effectiveness of the implementation of the proposed algorithm, the obtained performance gain for all simulated problems, calculated according to Equation (7) on each hardware architecture was compared to the maximum theoretical gain (Equation 9). To calculate G, total calculation times in sequential and parallel execution modes were used. Time parameters τ_{CPU} and τ_{GPU}, needed to calculate q and G_{max}, were measured and averaged for each case and hardware configuration. The obtained results are shown in Figure 5. Each point on the

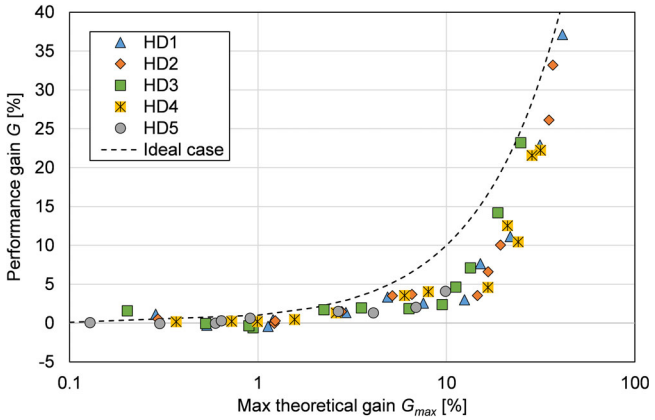

FIGURE 5 The obtained average performance gain in comparison with the maximum possible gain

diagram represents the averaged result of the same 10 simulations (as in Figure 4) of a particular test case on a given hardware configuration. As the ideal case, $G = G_{max}$ is assumed. The data demonstrate that the current implementation of the proposed algorithm reveals high efficiency. The average deviation of the obtained performance gain from the calculated G_{max} is 3.56%. Averaging the results by the case, gives a maximum of 8.91% for Case 2; averaging by the hardware configuration results in a maximum of 5.06% for HD4; the absolute maximum is 13.61% (Case 5 on HD4).

For all cases, the averaged measured value q (5) was less than 0.5, therefore the equality $G_{max} = q$ is valid for all results in the diagram.

Overall, there are two main conclusions from the results. First, the current implementation does not contain major flaws but has potential for improvement. Second, the proposed equation for the approximate calculation of G_{max} (9) is suitable for a rough estimation of the possible performance gain from the application of the new algorithm.

The transfer time was not considered when calculating the theoretical benefit. However, this parameter can have a significant impact on the performance of the algorithm. Therefore, this influence was additionally investigated during the simulation of all case studies. It was found that the use of the new method leads to an increase in the number of data exchange operations (CPU to GPU and back) almost 2 times: from 3094 to 6173, on average for all case studies. This growth is expected and is associated with a twofold reduction in the GPU cycle. The average number of Verlet lists re-evaluations due to not meeting the condition (B_{10}) is about 2.61 per simulation, which is 0.2% of the total number of Verlet calculations. The average size of a data packet when transferring from the CPU to the GPU is 50.84 MB, and back 23.24 MB. The total volumes of transmitted data during the entire simulation varied from 2.42 and 104.58 GB with an average of 39.07 GB for the sequential mode and 4.8 to 209.45 GB with an average of 77.62 GB for the parallel mode. At the same time, despite the significant amounts of transmitted data and the number of exchanges, the time spent on data transfer between the CPU and GPU in both directions averaged 0.62% (with variations from 0.04% to 1.5%) of the GPU clock cycle time or about 0.58% (with variations from 0.04% to 1.39%) of the entire simulation time for the parallel execution mode. If we also take into account all the time it takes to prepare the data for transmission and format them after they are received, the values increase to 2.49% of the total simulation time, with variations from 0.19% to 6.08%. Thus, it can be concluded that the data transfer time has an insignificant effect on the performance gain from the new algorithm (for presented case studies).

In general, one can observe a tendency that it is meaningful to use the new simulation strategy only for DEM scenes with a relatively large number of modeled objects. In Figure 6, the influence of the number of objects on the performance gain is shown. Here, according to the obtained performance gain, it is proposed to classify the influence and to distinguish four main groups: *no improvement* ($\lesssim 1\%$), *noticeable improvement* (from 1% to 5%), *moderate improvement* (from 5% to 15%), *significant improvement* ($\gtrsim 15\%$). The majority of the simulated DEM scenes exhibit *noticeable* or better improvement, with four cases gaining more than 5% performance. As it can be seen from the results, there is a general trend towards an increase in the obtained performance gain with an increase in the number of modeled objects.

One of the main factors here is the constant threshold distance L_v (Equation 1), which for all case studies, except Case 7 and Case 5, was equal to the diameter of the smallest particle. It is expected that enhancement of this approach and the usage of an auto-adjustable threshold, which will be calculated at runtime, can lead to a significant increase in efficiency.

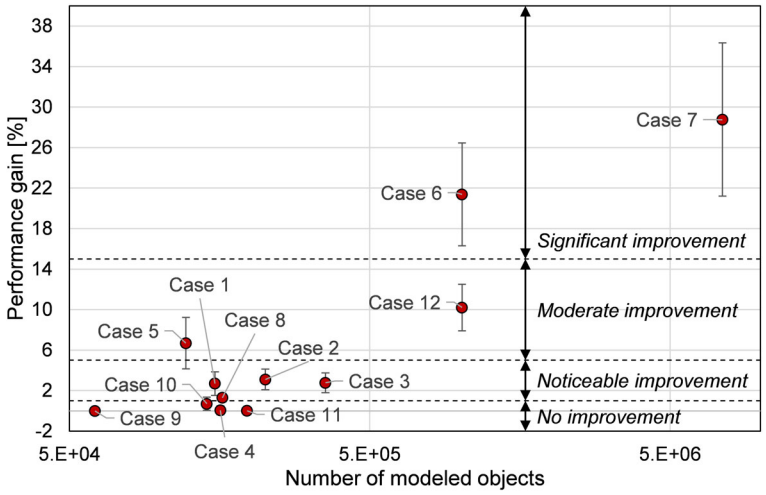

FIGURE 6 Average performance gain versus the total number of modeled objects

To further explore the differences in performance gains between test cases from different groups, time diagrams for Case 1 (noticeable improvement) and Case 6 (significant improvement) are provided. The diagrams shown in Figures 7 and 8 were obtained using the NVIDIA Nsight Systems 2020.4.3.7 profiling tool. The sampling rate of the process tree was set to the default value of 1 kHz.

Case 1 was sampled over 1.2 ms of process time (Figure 7). During this time interval, 11 calculations of Verlet lists took place in the sequential execution mode. The average pause in the work of the GPU was about 145 ms. By using the proposed parallel algorithm, these short intervals of idle time were almost eliminated. As expected, the number of calculations of the Verlet lists doubled to 21. It can also be seen that the impact of data exchange was insignificant.

Case 6 was sampled while calculating 2 ms process time (Figure 8). When comparing Case 1 to the timeline for Case 6, the difference in performance gain becomes apparent. In sequential mode, there were 16 calculations of Verlet lists, each lasting about 3.13 s. The entire simulation lastes 130 s, which means that the GPU was idle for more than a third of the whole time. Therefore, the application of the parallel algorithm was much more efficient here. The number of calculations of the Verlet list, as in the previous case, has doubled, and the GPU downtime has been reduced by more than four times.

Figure 8 also allows one to identify some flaws in the current implementation that were not visible in the previous case. Ideally, when using the parallel algorithm, either CPU or the GPU should be busy 100% of the time, however, the diagram still shows a GPU downtime of about 0.64 s for each iteration and significant dips in the load of the CPU. This indicates the imperfection of the current implementation, whose influence increases with an increase in the number of simulated objects. Further investigations are needed to find the reasons for the observed CPU and GPU dips and possible optimizations to improve the implementation.

In the case of parallel execution, the number of Verlet list calculations performed on the CPU is about twice that of the sequential mode. Thus, it is expected that the use of the concurrent strategy will lead to an increase in energy consumption. To check this, the change in power consumption of a computer with an HD4 configuration was measured during the transition from sequential to parallel mode. The measurements were performed using the electricity consumption meter Voltcraft SEM4500. Each test scene was simulated for 1 hour of real time, during which the total energy consumption of the system was measured. The results of these studies are summarized in Figure 9. As expected, power consumption in the parallel mode increases due to the influence of two factors: more frequent recalculation of Verlet lists on the CPU and shorter idle time of the GPU. It can also be seen that, in general, an increase in calculation performance directly correlates with an increase in energy consumption. Thus, due to the achieved performance gain, the overall energy requirements for the new strategy do not exceed or are even less than the power consumption for the sequential mode. The new approach is more energy efficient in 75% of the cases studied. The scenes with the worst energy efficiency are at the same time the cases with the worst performance gain, which again indicates that the new approach is not applicable for these simulations.

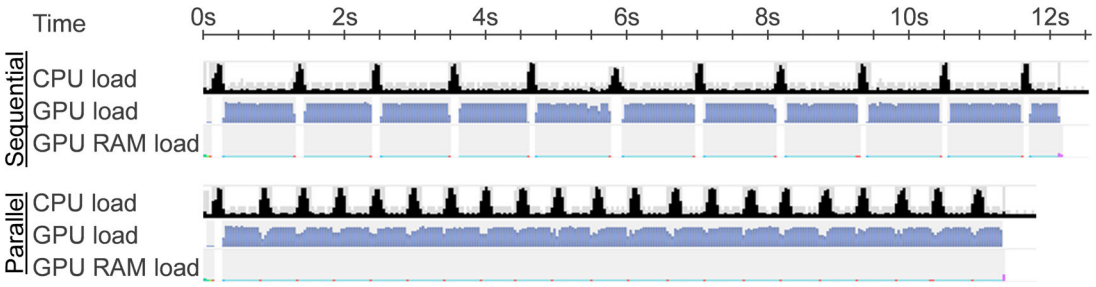

FIGURE 7 Timeline diagram for Case 1 on HD1 for sequential and parallel execution mode

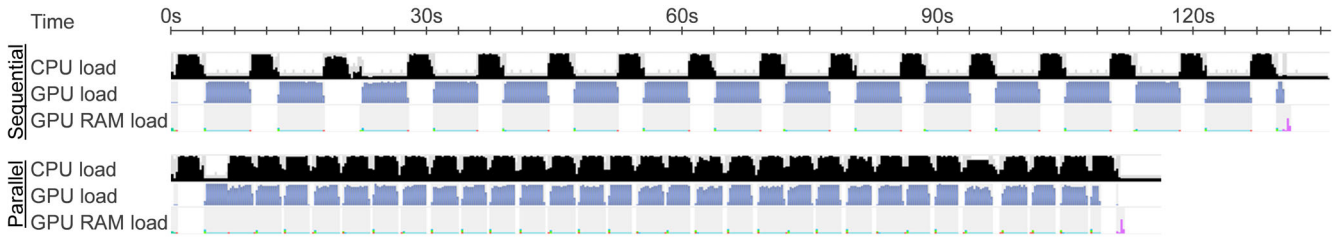

FIGURE 8 Timeline diagram for Case 6 on HD1 for sequential and parallel execution mode

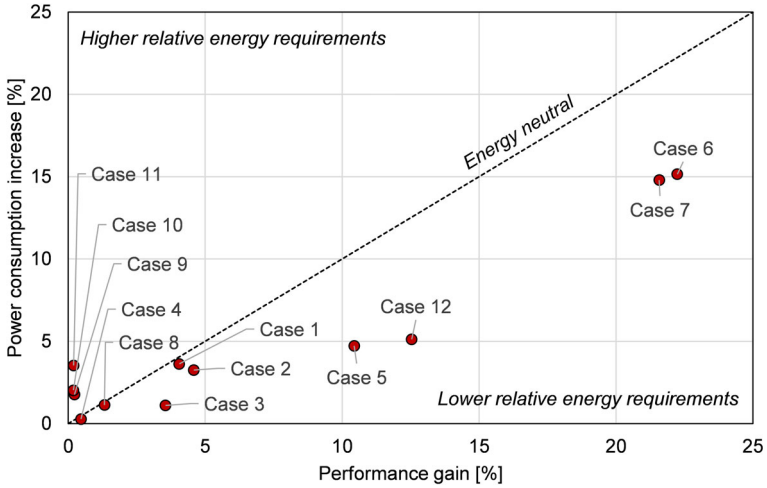

FIGURE 9 Dependence of the increase in power consumption on the obtained performance gain for the hardware configuration HD4

5 | CONCLUSIONS

In this contribution, the concurrent execution strategy has been proposed to improve the efficiency of DEM simulations on hybrid CPU–GPU architectures. The novel approach was developed with a focus on multi-core personal computers equipped with a general-purpose graphics processing unit. To analyze the efficiency of the new method, the calculation performance for various case studies, such as emptying a silo, mixing in a paddle mixer, dynamic impact, have been analyzed.

The obtained results have shown that for most of the investigated scenes, the novel approach allows achieving a noticeable performance gain. Due to the parallel co-execution, the average calculation time for the tested cases has been reduced by up to 37% compared to the sequential execution using hybrid CPU–GPU architectures. Furthermore, the new execution strategy allows reducing energy consumption when simulating DEM scenes consisting of a large number of discrete objects. Although the performance gain strongly depends on the type of problem being solved, on the number of modeled objects, and on the hardware configuration, we have shown the effectiveness of the new approach and high potential for further research and developments in this area. One of the possible improvements that can reduce the load imbalance between the CPU and the GPU is the implementation of the automatically adjustable threshold distance used to populate the Verlet lists, which would be calculated depending on the properties of each specific scene.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Vasyl Skorych https://orcid.org/0000-0002-6358-7385

REFERENCES

1. Cundall PA, Strack ODL. A discrete numerical model for granular assemblies. *Geotechnique*. 1979;29:47-65. doi:10.1680/geot.1979.29.1.47
2. Zhu H, Zhou Z, Yang R, Yu A. Discrete particle simulation of particulate systems: a review of major applications and findings. *Chem Eng Sci*. 2008;63:5728-5770. doi:10.1016/j.ces.2008.08.006
3. McGuire AD, Lee KF, Dosta M, et al. Compartmental residence time estimation in batch granulators using a colourimetric image analysis algorithm and discrete element modelling. *Adv Powder Technol*. 2017;28:2239-2255. doi:10.1016/j.apt.2017.06.005
4. Depta PN, Jandt U, Dosta M, Zeng A-P, Heinrich S. Toward multiscale modeling of proteins and bioagglomerates: an orientation-sensitive diffusion model for the integration of molecular dynamics and the discrete element method. *J Chem Inf Model*. 2018;59:386-398. doi:10.1021/acs.jcim.8b00613
5. Besler R, da Silva MR, Dosta M, Heinrich S, Janssen R. Discrete element simulation of metal ceramic composite materials with varying metal content. *J Eur Ceram Soc*. 2016;36:2245-2253. doi:10.1016/j.jeurceramsoc.2015.12.051
6. Dosta M, Jarolin K, Gurikov P. Modelling of mechanical behavior of biopolymer alginate aerogels using the bonded-particle model. *Molecules*. 2019;24:2543. doi:10.3390/molecules24142543
7. Liu D, Bu C, Chen X. Development and test of CFD–DEM model for complex geometry: a coupling algorithm for fluent and DEM. *Comput Chem Eng*. 2013;58:260-268. doi:10.1016/j.compchemeng.2013.07.006
8. Potyondy DO. The bonded-particle model as a tool for rock mechanics research and application: current trends and future directions. *Geosyst Eng*. 2015;18:1-28. doi:10.1080/12269328.2014.998346

9. Norouzi H, Zarghami R, Mostoufi N. New hybrid CPU-GPU solver for CFD-DEM simulation of fluidized beds. *Powder Technol*. 2017;316:233-244. doi:10.1016/j.powtec.2016.11.061
10. Xu M, Chen F, Liu X, Ge W, Li J. Discrete particle simulation of gas–solid two-phase flows with multi-scale CPU–GPU hybrid computation. *Chem Eng J*. 2012;207-208:746-757. doi:10.1016/j.cej.2012.07.049
11. Sampat C, Bettencourt F, Baranwal Y, et al. A parallel unidirectional coupled DEM-PBM model for the efficient simulation of computationally intensive particulate process systems. *Comput Chem Eng*. 2018;119:128-142. doi:10.1016/j.compchemeng.2018.08.006
12. Spettl A, Dosta M, Klingner F, Heinrich S, Schmidt V. Copula-based approximation of particle breakage as link between DEM and PBM. *Comput Chem Eng*. 2017;99:158-170. doi:10.1016/j.compchemeng.2017.01.023
13. Lindner J, Menzel K, Nirschl H. Simulation of magnetic suspensions for HGMS using CFD, FEM and DEM modeling. *Comput Chem Eng*. 2013;54:111-121. doi:10.1016/j.compchemeng.2013.03.012
14. Gan J, Evans T, Yu A. Application of GPU-DEM simulation on largescale granular handling and processing in ironmaking related industries. *Powder Technol*. 2020;361:258-273. doi:10.1016/j.powtec.2019.08.043
15. Kureck H, Govender N, Siegmann E, Boehling P, Radeke C, Khinast JG. Industrial scale simulations of tablet coating using GPU based DEM: a validation study. *Chem Eng Sci*. 2019;202:462-480. doi:10.1016/j.ces.2019.03.029
16. Tian Y, Zhang S, Lin P, Yang Q, Yang G, Yang L. Implementing discrete element method for large-scale simulation of particles on multiple GPUs. *Comput Chem Eng*. 2017;104:231-240. doi:10.1016/j.compchemeng.2017.04.019
17. Berger R, Kloss C, Kohlmeyer A, Pirker S. Hybrid parallelization of the LIGGGHTS open-source DEM code. *Powder Technol*. 2015;278:234-247. doi:10.1016/j.powtec.2015.03.019
18. He Y, Evans T, Yu A, Yang R. A GPU-based DEM for modelling large scale powder compaction with wide size distributions. *Powder Technol*. 2018;333:219-228. doi:10.1016/j.powtec.2018.04.034
19. Potluri S, Wang H, Bureddy D, Singh A, Rosales C, Panda DK. Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum; 2012; IEEE. 10.1109/ipdpsw.2012.228
20. Lee C, Ro WW, Gaudiot J-L. Boosting CUDA applications with CPU–GPU hybrid computing. *Int J Parallel Program*. 2013;42:384-404. doi:10.1007/s10766-013-0252-y
21. Wrede F, Ernsting S. Simultaneous CPU–GPU execution of data parallel algorithmic skeletons. *Int J Parallel Program*. 2017;46:42-61. doi:10.1007/s10766-016-0483-9
22. Navarro A, Corbera F, Rodriguez A, Vilches A, Asenjo R. Heterogeneous parallel for template for CPU–GPU chips. *Int J Parallel Program*. 2018;47:213-233. doi:10.1007/s10766-018-0555-0
23. Mittal S, Vetter JS. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput Surv*. 2015;47:1-35. doi:10.1145/2788396
24. Raju K, Chiplunkar NN. A survey on techniques for cooperative CPUGPU computing. *Sustain Comput-Inform*. 2018;19:72-85. doi:10.1016/j.suscom.2018.07.010
25. Zheng J, An X, Huang M. GPU-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations. *Comput Struct*. 2012;112-113:193-204. doi:10.1016/j.compstruc.2012.08.003
26. Mio H, Shimosaka A, Shirakawa Y, Hidaka J. Cell optimization for fast contact detection in the discrete element method algorithm. *Adv Powder Technol*. 2007;18:441-453. doi:10.1163/156855207781389519
27. Su J, Gu Z, Xu XY. Discrete element simulation of particle flow in arbitrarily complex geometries. *Chem Eng Sci*. 2011;66:6069-6088. doi:10.1016/j.ces.2011.08.025
28. Brosh T, Kalman H, Levy A. Fragments spawning and interaction models for DEM breakage simulation. *Granular Matter*. 2011;13:765-776. doi:10.1007/s10035-011-0286-z
29. Zhou W, Xu K, Ma G, Chang X. On the breakage function for constructing the fragment replacement modes. *Particuology*. 2019;44:207-217. doi:10.1016/j.partic.2018.08.006
30. Dosta M, Antonyuk S, Heinrich S. Multiscale simulation of agglomerate breakage in fluidized beds. *Ind Eng Chem Res*. 2013;52:11275-11281. doi:10.1021/ie400244x
31. Dosta M, Skorych V. MUSEN: An open-source framework for GPU-accelerated DEM simulations. *SoftwareX*. 2020;12:100618. doi:10.1016/j.softx.2020.100618
32. Verlet L. Computer εexperimentsε on classical fluids. I. thermodynamical properties of Lennard-Jones molecules. *Phys Rev*. 1967;159:98-103. doi:10.1103/physrev.159.98
33. Lipscomb TJ, Zou A, Cho SS. Parallel verlet neighbor list algorithm for GPU-optimized MD simulations. Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine; 2012; ACM Press. 10.1145/2382936.2382977
34. Wellmann C, Wriggers P. Homogenization of granular material modeled by a 3d DEM. In: Oñate E, Owen R, eds. *Particle-Based Methods*. Springer; 2011:211-231. doi:10.1007/978-94-007-0735-1_8
35. Quentrec B, Brot C. New method for searching for neighbors in molecular dynamics computations. *J Comput Phys*. 1973;13:430-432. doi:10.1016/0021-9991(73)90046-6
36. Fang X, Tang J, Luo H. Granular damping analysis using an improved discrete element approach. *J Sound Vib*. 2007;308:112-131. doi:10.1016/j.jsv.2007.07.034
37. Iwai T, Hong C-W, Greil P. Fast particle pair detection algorithms for particle simulations. *Int J Mod Phys C*. 1999;10:823-837. doi:10.1142/s0129183199000644
38. Rodriguez VA, de Carvalho RM, Tavares LM. Insights into advanced ball mill modelling through discrete element simulations. *Miner Eng*. 2018;127:48-60. doi:10.1016/j.mineng.2018.07.018
39. Oka S, Kašpar O, Tokárová V, et al. A quantitative study of the effect of process parameters on key granule characteristics in a high shear wet granulation process involving a two component pharmaceutical blend. *Adv Powder Technol*. 2015;26:315-322. doi:10.1016/j.apt.2014.10.012
40. Dosta M, Brockel U, Gilson L, Kozhar S, Auernhammer GK, Heinrich S. Application of micro computed tomography for adjustment of model parameters for discrete element method. *Chem Eng Res Des*. 2018;135:121-128. doi:10.1016/j.cherd.2018.05.030

41. Lee KF, Dosta M, McGuire AD, et al. Development of a multi-compartment population balance model for high-shear wet granulation with discrete element method. *Comput Chem Eng*. 2017;99:171-184. doi:10.1016/j.compchemeng.2017.01.022
42. Dranishnykov S, Dosta M. Advanced approach for simulation results saving from discrete element method. *Adv Eng Softw*. 2019;136:102694. doi:10.1016/j.advengsoft.2019.102694

How to cite this article: Skorych V, Dosta M. Parallel CPU–GPU computing technique for discrete element method. *Concurrency Computat Pract Exper*. 2022;34(11):e6839. doi: 10.1002/cpe.6839