A Multi-GPU PCISPH Implementation with Efficient Memory Transfers

Kevin Verma*† Chong Peng* Kamil Szewc* Robert Wille†
*ESS Engineering Software Steyr GmbH, Austria
†Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
Email: {kevin.verma, chong.peng, kamil.szewc}@essteyr.com robert.wille@jku.at

***Abstract*—*Smoothed Particle Hydrodynamics* (SPH) is a particle-based method for fluid flow modeling. One promising variant of SPH is *Predictive-Corrective Incompressible SPH* (PCISPH), which employs a dedicate prediction-correction scheme and, by this, outperforms other SPH variants by almost one order of magnitude. However, similar to other particle-based methods, it suffers from a huge numerical complexity. In order to simulate real world phenomena, several millions of particles need to be considered. To make SPH applicable to real world engineering problems, it is hence common to exploit massive parallelism of multi-GPU architectures. However, certain algorithmic characteristics of PCISPH make it a non-trivial task to efficiently parallelize this method on multi-GPUs. In this work, we are, for the first time, proposing a multi-GPU implementation for PCISPH. To this end, we are proposing a scheme which allows to overlap the memory transfers between GPUs by actual computations and, by this, avoids the drawbacks caused by the mentioned algorithmic characteristics of PCISPH. Experimental evaluations confirm the efficiency of the proposed methods.**

I. Introduction

The simulation of fluid flows has increasingly become an important task in engineering and science. It allows to model complex scenarios, which are difficult or costly to measure in the real world. Grid-based *Computational Fluid Dynamics* (CFD, [1]) methods such as *Finite Volume Methods* (FVM, [2]), *Finite Element Methods* (FEM, [3]), or *Electrophoretic Deposition* (EPD, [4], [5]) find many applications in a wide range of industries. However, in the recent years, so-called particle-based methods have gained more importance for the simulation of fluid flows. In such methods, the fluid is discretized by a set of particles, which completely define the fluid and move in time and space. The advantage of particle-based methods compared to grid-based methods is their ability to handle free surface flows as well as flows involving complex physics.

One of the most common particle-based methods is *Smoothed Particle Hydrodynamics* (SPH), which was initially proposed by Monaghan et al. [6]. In SPH, the governing equations are discretized by discrete particles and a variety of particle-based formulations are employed to calculate physical properties such as density or velocity. One of the main difficulties in particle-based methods is to satisfy the incompressibility conditions of fluids. For that, *Weakly-Compressible SPH* (WCSPH) was proposed by Becker et al. [4], which has become one of the most popular variants of SPH. In WCSPH, a stiff equation of state is used to model pressure. Although by that, incompressibility is satisfied, WCSPH suffers from a severe time step restriction. In order to enforce higher incompressibility, smaller time steps need to be considered during the simulation. Thus, the computational costs increase with increasing incompressibility. The resulting computational expenses frequently prevent WCSPH from being applied in complex real-world scientific and engineering problems.

To address these shortcomings, *Predictive-Corrective Incompressible SPH* (PCISPH) was proposed by Solenthaler et al. [7]. In PCISPH, incompressibility is enforced by employing a prediction-correction scheme to compute particle pressures. In this scheme, particle positions and velocities are temporarily forwarded in time to estimate particle densities. Based on these estimated densities, the particle pressures are iteratively computed such that the predicted destiny fluctuation is smaller than a user-defined threshold. By this scheme, PCISPH allows to use a time step significantly larger compared to WCSPH. Eventually, this yields very accurate results while, at the same time, outperforms other SPH variants such as WCSPH by almost one order of magnitude [8].

However, in contrast to WCSPH, PCISPH does not provide an obvious capability for parallelization and, by this, the exploitation of *High Performance Computing* (HPC) methods for PCISPH was rather limited thus far. This is a serious drawback as efficient parallelization is considered essential to further improve the scalability of corresponding approaches and, by this, make them applicable for the simulation of real-world phenomena in appropriate resolution, where typically millions of particles need to be considered. In fact, other SPH methods such as WCSPH employ methods for *General Purpose Computation on Graphics Processing Units* (GPGPU; utilizing the parallelism of GPUs) and extend that further for multi-GPU architectures [9], [10], [11].

For PCISPH, such methods could not been utilized yet – although some attempts have been made on this in the recent years. For example in [12], a parallel framework which allows the execution of PCISPH on a single GPU has been proposed. The resulting speedup was 23 on a single GPU in comparison to a sequential CPU implementation. However, solutions utilizing *multi-GPU* architectures for PCISPH have, to the best of our knowledge, not been proposed so far. This is likely caused by the fact that typical multi-GPU methods can not be directly applied to PCISPH. In fact, certain algorithmic characteristics of PCISPH (which are discussed in detail later in Section III) make it a non-trivial task to efficiently parallelize PCISPH for multi-GPU architectures. As a consequence, although PCISPH clearly outperforms WCSPH by almost one order of magnitude, it still remains limited with respect to parallelization – significantly restricting the potential for further improvements of PCISPH.

In this work, we try to overcome this limitation. To this end, we analyze and discuss the algorithmic characteristics mentioned above which prevent the exploitation of parallelization of PCISPH on multi-GPUs. Based on this analysis, we are then proposing a scheme which overcomes these difficulties and, eventually, allows for an efficient parallelization of PCISPH on multi-GPUs. The proposed scheme allows to efficiently conduct memory transfers and, therefore, reduces idle times of GPUs. Moreover, the proposed scheme can not only be applied to PCISPH, but to all common SPH variants (including WCSPH). Experimental evaluations with an industrial PCISPH-based simulation tool confirm that, by efficiently handling memory transfers, the performance can be increased significantly.

The remainder of this paper is organized as follows: The following section provides the background on SPH as well as the dedicated PCISPH method. Afterwards, Section III discusses the resulting problems that are addressed by the proposed scheme described in Section IV. Section V provides insights on the implementation of the proposed scheme and, finally, Section VI summarizes the obtained results from the experimental evaluations before the paper is concluded in Section VII.

II. Background

In order to keep this work self-contained, this section reviews the basics on the SPH method in general – followed by a review on the dedicated *Predictive-Corrective Incompressible SPH* (PCISPH) scheme. We keep the descriptions brief and focus on the issues which are relevant for this work. For a more detailed treatment of the SPH and PCISPH methods, we refer to [13], [14], and [7], respectively.

A. Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a particle-based, fully Lagrangian method for fluid-flow modeling and simulation. This method was independently proposed by Gingold and Monaghan [15] to simulate astrophysical phenomena at the hydrodynamic level (compressible flow). Nowadays, the SPH approach is increasingly used for simulating hydro-engineering applications – involving free-surface flows where the natural treatment of evolving interfaces makes it an enticing approach.

The main ideas of the *SPH* method rely on the following basis: The fluid to be simulated is represented in terms of discrete particles summarized in a set J. Then, a scalar quantity A is interpolated at position r by a weighted sum of contributions from J, i.e.

$$\langle A(r)\rangle = \sum\nolimits_{j\in J} A_j V_j W(r-r_j, 2h), \tag{1}$$

where V_j is the volume of the respective particle j, r_j is the position of this particle, and A_j the field quantity at position r_j. W is a smoothing kernel with the so-called smoothing length $2h$ as a width – defining that only particles within a distance shorter than $2h$ will interact with a particle j. This kernel function W is a central part of SPH simulations and the appropriate choice of a smoothing kernel for a specific problem is of great importance. At the same time, a kernel must satisfy three conditions/properties, namely

1) the *normalization condition*

$$\int_r W(r-r_j, 2h)dr = 1 \tag{2}$$

stating that the integral over its full domain is unity,

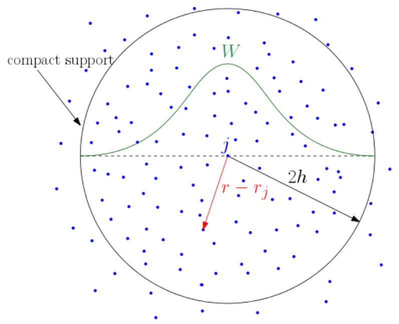


Fig. 1: Illustration of a 2D domain for a particle $j \in J$.

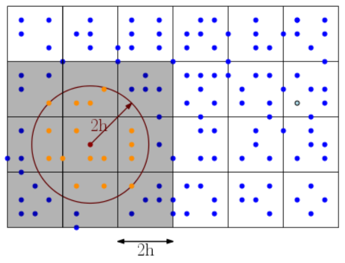


Fig. 2: Virtual search grid.

2) the *Delta function property*

$$\lim_{h\to 0} W(r-r_j, 2h) = \delta(r-r_j) \tag{3}$$

stating that, if the smoothing length $2h$ approaches zero, a delta distribution is applied (with δ being the Dirac delta function), and

3) the *compact support condition*

$$W = 0 \text{ when } |r-r_j| > 2h \tag{4}$$

ensuring that only particles within the smoothing length $2h$ are considered.

Example 1. *Fig. 1 illustrates a 2D domain with a kernel function W and smoothing length $2h$ for a particle $j \in J$.*

Now, in order to update the behavior of each particle, any SPH simulation need to frequently access neighboring particles during one computational iteration. More precisely, all particles which are located within the influence radius $2h$ need to be accessed (cf. Eq. 1; note that $2h$ itself is defined by the smoothing length). Realizing that in a naive fashion would require the iteration through the entire fluid domain – yielding a complexity of $O(|J|^2)$. Despite the polynomial nature of this complexity, this is usually impractical because of the sheer number $|J|$ of particles to consider.

Because of that, optimizations are utilized which rely on a so-called *virtual search grid* as illustrated in Fig. 2. Here, the entire fluid domain is divided into a search grid where each cell has the size of the influence radius $2h$. By this, it can be guaranteed that, for a considered particle $j \in J$ (exemplarily denoted by a red dot in Fig. 2), all neighboring particles must be located within the adjacent cells. This way, instead of iterating through the entire fluid domain, only a subset of it (highlighted grey in Fig. 2) has to be considered in order to determine the neighboring particles (denoted by orange dots in Fig. 2).

The most expensive part in terms of computational time in particle-based methods is typically to enforce the incompressibility of the fluid. In SPH, the most common method to do that is *Weakly Compressible SPH* (WCSPH), where pressure is modeled by employing an equation of state. Although, by that, incompressibility is satisfied, WCSPH suffers from the fact that higher incompressibility requires to consider smaller time steps during the simulation. To address this shortcoming, *Predictive-Corrective Incompressible SPH* (PCISPH) was proposed, which is discussed next.

B. Predictive-Corrective Incompressible SPH

Predictive-Corrective Incompressible SPH (PCISPH) was initially proposed by Solenthaler et al. [7] to avoid the computational expenses of solving a pressure Poisson equation. By avoiding that, PCISPH allows to use a larger time step which typically results in an increased performance.

In PCISPH, a prediction-correction scheme is a employed, where positions and velocities are temporarily forwarded in time to estimate particle densities. Based on this estimated density $\rho^*(t+1)$, the pressure $p_j(t)$ is iteratively computed for each particle $j \in J$ such that the predicted density fluctuation $\rho^*_{err}(t+1)$ is smaller than a user-defined threshold η.

More precisely, the predicted densities $\rho^*(t+1)$ are computed using the SPH density summation equation similar to Equation 1, namely

$$\rho^*(t+1) = \sum\nolimits_{j\in J} m_j W(r^* - r^*_j, 2h), \tag{5}$$

where m_j is the particle mass and r^*_j is the predicted particle position. W is the smoothing kernel with the smoothing length $2h$ as a width. To minimize the occurring density error ρ^*_{err}, a correction of the current pressure is computed subsequently. Finally, the pressure force

$$F_p(t) = -m_i \sum\nolimits_{j\in J} m_j \left(\frac{p_i(t)}{\rho^*_i(t)} + \frac{p_j(t)}{\rho^*_j(t)}\right) \nabla W(r^* - r^*_j, 2h) \tag{6}$$

is used to to recompute the predicted positions and velocities for all particles $j \in J$. This procedure is repeated until it converges, i.e. $\rho^*_{err}(t+1) < \eta$. Although additional iterations are required within one time step to achieve this convergence (hereinafter referred to as *inner iterations*), PCISPH allows to use a larger time step as compared to WCSPH.

Example 2. *Consider the simulation of a process with a duration of* $1s$*, which is divided into discrete time steps. Assuming that for this process, WCSPH needs to employ time steps of e.g.* $1e^{-4}s$*, which yields* 1000 *discrete time steps. For the same process however, PCISPH allows to employ a significantly larger time step, e.g. a time step of* $1e^{-3}s$*, which results in* 100 *discrete time steps.*

Therefore, PCISPH typically outperforms the WCSPH by almost one order of magnitude [8].

III. Motivation: Parallelization of PCISPH on Multi-GPUs

Similar to the WCSPH method, the discrete particle formulation of physical quantities also makes the PCISPH method generally suitable for parallelization – e.g. using the *General Purpose Computations on Graphics Processing Units* (GPGPU) technology. However, certain characteristics of PCISPH result in an exceeding synchronization effort on multi-GPU architectures. In this section, we review the state-of-the-art for parallelization of SPH simulations on multi-GPUs and, afterwards, discuss why PCISPH was not able to exploit these accomplishments yet due to the synchronization problems. This provides the motivation of this work in which we aim for overcoming these problems and eventually allow for an efficient parallelization of PCISPH on multi-GPUs.

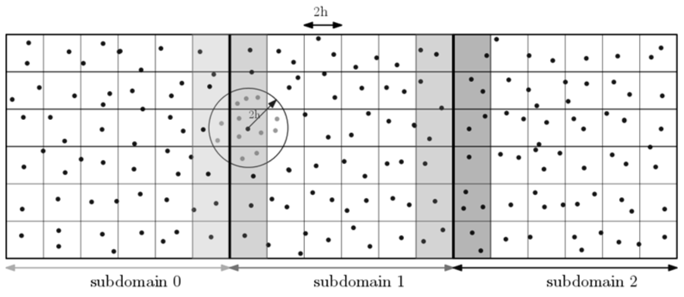


Fig. 3: Subdomain distribution for 3 GPUs.

A. State-of-the-Art

SPH solutions utilizing the computational power of GPUs have initially been introduced by Kolb and Cuntz [16] as well as Harada et al. [17], where the *Open Graphics Library* (OpenGL) was employed. Later, SPH implementations based on the *Compute Unified Device Architecture* (CUDA) have been developed [18].

However, in order to simulate huge domains involving millions of particles, a single GPU device is usually not sufficient anymore. In these cases, the underlying SPH implementation needs to be distributed over several devices – yielding a *multi-GPU architecture* as originally proposed by Dominguez et al. [9]. Here, CUDA and *Message Passing Interface* (MPI) have been employed to parallelize the SPH simulation with up to 128 GPUs where each GPU covered the simulation of up to 8 million particles. Besides that, a similar architecture has also been utilized in the solution proposed in [10].

These SPH multi-GPU solutions employ a spatial subdivision of the domain to partition the whole domain into individual subdomains. These subdomains are distributed to the corresponding GPUs and executed in parallel. Fig. 3 illustrates a subdivision into 3 subdomains. Generally, this method yields a setup where each subdomain has two neighboring subdomains, except for those at the perimeter of the domain, which have only one neighbor. However, the boundaries of each subdomain need special consideration on a multi-GPU architecture as illustrated by the following example.

Example 3. *Consider a particle* $j \in J$ *at the perimeter of one centered sub-domain (see Fig. 3). The neighbors of* j *within* $2h$ *are not only located in cells of its own subdomain (covered by GPU 1 in Fig. 3), but also in cells of the neighboring subdomain (covered by GPU 0). Since the subdomains are distributed to different GPUs, the neighbors of* j *are located in a distinct device and, hence, a different memory pool. This hinders fast neighbor access.*

In order to accelerate neighbor access, each GPU should therefore hold a copy of the data located at the *edge* of its adjacent subdomains, i.e. all cells within $2h$ at the perimeter of a subdomain. These edges are also referred to as *halo* of a subdomain.

Algorithm 1 PCISPH

```
1:  $T \leftarrow$ time steps
2:  $N_g \leftarrow$ number of GPUs
3:  $\eta \leftarrow$ maximum allowed density error
4:  for $t \in \{0, ..., T\}$ do
5:      for $GPU g_i \in \{g_0, \dots, g_{N_g}\}$ do
6:          Update Boundary Particles
7:          ExchangeHalo($g_i$, $g_{i+1}$)
8:          Compute $F^{v,g,ext}$
9:          ExchangeHalo($g_i$, $g_{i+1}$)
10:         while $\rho^*_{err} > \eta$ do
11:             Update Boundary Particles
12:             ExchangeHalo($g_i$, $g_{i+1}$)
13:             Predict Density $\rho^*(t+1)$
14:             Predict Density Variation $\rho^*_{err}(t+1)$
15:             Compute Pressure $p(i) += f(\rho^*_{err}(t+1))$
16:             ExchangeHalo($g_i$, $g_{i+1}$)
17:             Pressure Correction
18:             ExchangeHalo($g_i$, $g_{i+1}$)
19:             Compute Pressure Force $F_p(t)$
20:             ExchangeHalo($g_i$, $g_{i+1}$)
```

More precisely, after every particle modification, a brief synchronization over all halos takes place. Afterwards, the parallel computation on all subdomains can continue with the updated values. Overall, this allows for significant speed-ups e.g. for WCSPH due to parallelization on multiple GPUs.

B. Synchronization Problems with PCISPH

The halo exchange as reviewed above constitutes the major bottleneck of any parallelization method for SPH. But while e.g. WCSPH still gains significant total improvements from that, it yields to a "showstopper" for PCISPH. This is because PCISPH requires a much more dedicated synchronization scheme.

In order to discuss this explicitly, let's consider a possible multi-GPU PCISPH implementation as shown in Algorithm 1. Here, as discussed in Section II-B, inner iterations are used to enforce the incompressibility by a prediction-correction scheme to determine the particle pressures. The velocities and positions are temporarily forwarded in time and used to estimate the new particle densities (see Line 14). For each particle $j \in J$, the predicted variation from the reference density is computed and used to compute the pressure values, which are then used for the computation of the pressure force (see Line 15-18). This process is iterated until it converges, i.e. until all particle density fluctuations are smaller than a defined threshold η. Since in every iteration the particle values are frequently updated, the halos need to be exchanged between GPUs after every particle modification. In total, this yields $2 + K \cdot 4$ halo exchange processes, where K refers to the number of iterations until it converges (typically 3-5 iterations).

Overall, this yields an execution flow as sketched in Fig. 4. The setup on the left shows a simplified multi-GPU architecture with peer-to-peer memory access, where GPUs are connected by PCIe. Each of the GPUs is assigned one subdomain. For every time step t, each GPU is conducting the computations for its respective subdomain. After the computation is completed, the halo data needs to be exchanged in order to ensure correct particle values. During these memory transfers,

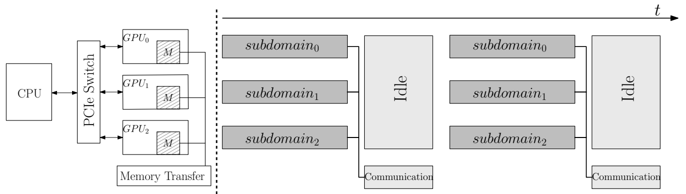
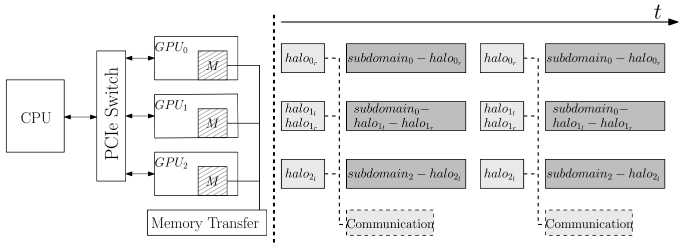


Fig. 4: Workflow of SPH on multi-GPU architectures.



t
CPU
PCIe Switch
GPU_0
GPU_1
GPU_2
M
Memory Transfer
$halo_{0_r}$
$subdomain_0 - halo_{0_r}$
$halo_{1_l}$ $halo_{1_r}$
$subdomain_0 - halo_{1_l} - halo_{1_r}$
$halo_{2_l}$
$subdomain_2 - halo_{2_l}$
Communication

Fig. 5: Proposed workflow to reduce communication time.

the GPUs are inherently rendered idle, since a synchronization barrier is necessary after exchanging data to ensure that the computations within subdomains is not conducted on outdated data. This time spent on communicating between processes further increases when the number of GPUs increases, since the synchronization needs to be performed globally among all GPUs to avoid race conditions.

Hence, in multi-GPU PCISPH simulations these memory transfers result in significant loss of efficiency.

IV. Proposed Solution: Performing Halo Exchange Parallel to Actual Computations

In any SPH simulation on multi-GPU architectures, memory needs to be exchanged between GPUs after particle data has been modified – yielding the bottleneck discussed above. In this work, we are proposing to overcome these synchronization problems by conducting the required halo exchanges (and, by this, the communication between GPUs) in parallel to the actual computations. To this end, a revised workflow is utilized which is illustrated in Fig. 5. Here, the computation is essentially divided into three separate tasks:

1) First, all computations within the halo regions are conducted. For the subdomains located in the perimeter of the domain, the computations for only one halo region need to be executed, i.e. the right halo for the leftmost subdomain (hereinafter referred to as $halo_r$) and the left halo for the rightmost subdomain (hereinafter referred to as $halo_l$). For the subdomains located in the middle of the domain, the computations for both, $halo_l$ and $halo_r$, are conducted.
2) Second, the computations for the remainder of the subdomains are performed. For the leftmost and rightmost subdomains, this yields the computations within the region $subdomain - halo_r$ and $subdomain - halo_l$, respectively. For the subdomains located in the middle of the domain, the computations within the region $subdomain - halo_r - halo_l$ are executed.
3) Finally, the halos between the subdomains are exchanged, i.e. the memory transfers are conducted. Since the interactions within the halo regions have already been computed, this can now be conducted in parallel to the second task.

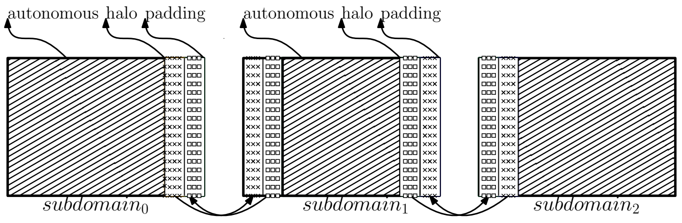


Fig. 6: Region division of subdomains.

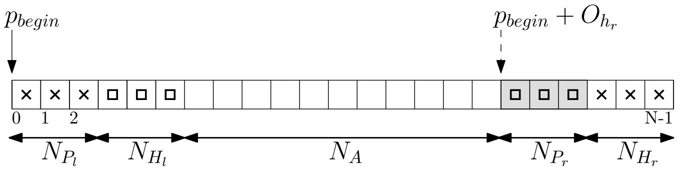


Fig. 7: Example of an array representation with respective offsets, where the right halo should be computed.

Overall, this re-arrangement of tasks reduces the synchronization problems of existing parallelization schemes for SPH. While this also could help to improve existing SPH solutions such as WCSPH, it particular overcomes the bottleneck of PCISPH solutions and, eventually, allows for a more efficient memory transfer and, hence, parallelization for this scheme. This is confirmed by experimental evaluations summarized in Section VI. Before, we however provide some more details on the implementation of the proposed scheme.

V. Implementation

In order to implement the idea proposed above, the synchronous memory transfers of existing solutions need to be replaced by a dedicated asynchronous communication strategy. By this, one process can execute other tasks while asynchronously sending or receiving data without needing to wait for the transfer to be completed. To allow for such asynchronous memory transfers, each subdomain is first divided into three separate regions as shown in Fig. 6, namely:

1) The *halo* region: Refers to the region at the perimeter of a domain.
2) The *padding* region: Refers to the region within a subdomain that is a copy of the halo of the neighboring subdomain. By this, it is ensured that the neighboring particles of the halo region are located in the same memory pool to allow fast neighbor access.
3) The *autonomous* region of the subdomain: The region which can be computed completely independent of neighboring subdomains, hence the region which is neither a halo nor a padding.

To allow overlapping of the memory transfers with the actual computation, it needs to be ensured that the regions which need to be exchanged between GPUs are already computed as soon as memory is exchanged. For that purpose, the computations are first conducted for the halo region of each subdomain. In this regard, there are three general cases to be considered:

1) For the leftmost subdomain, the computations for $halo_r$ needs be conducted.
2) For the rightmost subdomain, the computations for $halo_l$ needs be conducted.
3) For the subdomains located in the middle, the computations for $halo_l$ and $halo_r$ needs to be conducted.

Internally, in our implementation, for every subdomain the particles are stored in a single array and are kept sorted according to their position in the domain. This array contains the particles of all three regions, i.e. the autonomous, the halo, and the padding region. For the computation solely within the halo region, respective offsets are calculated. Correspondingly, the offset for the right halo (O_{h_r}) is

$$O_{h_r} = N_{P_l} + N_{H_l} + N_A, \tag{7}$$

where N_{P_l} is the total number of particles in the left padding region, N_{H_l} the number of particles in the left halo region, and N_A the number of particles in the autonomous region.

Fig. 7 exemplary illustrates an array representation for which the computations within the right halo region (highlighted in gray) should be conducted. All the particles of one subdomain are stored within one array, instead of keeping copies of the independent regions in distinct containers. This ensures that: (1) no unnecessary data needs to be copied and (2) all particles are stored in contiguous memory. This contiguous memory representation essentially results in an increased performance, considering that, for the computations inside the halo region, also the neighboring particles within the autonomous and padding region need to be accessed.

The subdomains located in the middle contain two halo regions, one on the left and one on the right. However, these two halo regions are completely independent of each other, which is ensured by the maximum influence radius for each particle (as reviewed in Section II). This allows to launch two CUDA kernels in parallel by employing separate CUDA streams – computing both, the left and the right halo concurrently.

After this computation in the halo region, a synchronization among these streams is needed, since the next step is to exchange the halos between the subdomains. By a synchronization barrier, it is ensured that computations within halo regions are already completed.

This process of exchanging halos can now be launched asynchronously to the rest of the computations. To overlap these memory transfers with the computations within the autonomous region, these two operations are launched in two separate non-blocking CUDA streams. As a result, the memory transfers can be conducted in parallel to the asynchronous computations within the autonomous region.

Overall, this yields an algorithm for PCISPH as illustrated in pseudocode in Algorithm 2.

Algorithm 2 Asynchronous Exchange of Halos for PCISPH

```
1: $\eta \leftarrow$ maximum allowed density error
2: **for** $t \in \{0, ..., T\}$ **do**
3:     **for** $GPUg_i \in \{g_0, \dots, g_{N_g}\}$ **do**
4:         UpdateBoundaryParticles(Halo)
5:         **AsyncExchangeHalo**($g_i$, $g_{i+1}$)
6:         UpdateBoundaryParticles(Autonomous)
7:         ComputeForcesHalo $F^{v,g,ext}$
8:         **AsyncExchangeHalo**($g_i$, $g_{i+1}$)
9:         ComputeForcesAutonomous $F^{v,g,ext}$
10:        **InnerIteration**                    ▷ see Algorithm 3
```

Algorithm 3 Inner Iteration for PCISPH

```
1:  function INNERITERATION
2:      while $\rho^*_{err} > \eta$ do
3:          UpdateBoundaryParticles(Halo)
4:          AsyncExchangeHalo($g_i$, $g_{i+1}$)
5:          UpdateBoundaryParticles(Autonomous)
6:          PredictDensityHalo $\rho^*(t+1)$
7:          CompPressureHalo $p(i) \mathrel{+}= f(\rho^*_{err}(t+1))$
8:          AsyncExchangeHalo($g_i$, $g_{i+1}$)
9:          PredictDensityAutonomous $\rho^*(t+1)$
10:         CompPressureAutonomous $p(i) \mathrel{+}= f(\rho^*_{err}(t+1))$
11:         PressureCorrectionHalo
12:         AsyncExchangeHalo($g_i$, $g_{i+1}$)
13:         PressureCorrectionAutonomous
14:         ComputePressureForceHalo $F_p(t)$
15:         AsyncExchangeHalo($g_i$, $g_{i+1}$)
16:         ComputePressureForceAutonomous $F_p(t)$
```

The individual functions have been divided into two separate functions (see e.g. Line 6-8). First, the computations within the halo regions are conducted, followed by the computations in the autonomous region. Asynchronously to this computation, the halos are exchanged (see e.g. Line 7). The inner iterations are shown in Algorithm 3. Here, again the functions are separated into halo and autonomous region. In total, the halos need be exchanged four times in every inner iteration (see Line 4, 8, 12, 15), which are however overlapped by actual computations

Evaluations summarized in the next section, confirm this improvement.

VI. Experimental Evaluations

To evaluate the performance of the proposed scheme, we implemented the described methods in C++ on top of an industrial PCISPH-based simulation tool. Besides that, we additionally compared the performance of the resulting method to WCSPH. This is motivated by the fact that, although PCISPH outperforms WCSPH by almost one order of magnitude, WCSPH still provides better parallelization capabilities. Conducting a comparison between both SPH schemes allows us to evaluate how close the speedup and efficiency of PCISPH parallelization gets to the speedups and efficiency which WCSPH takes for granted thus far. Correspondingly obtained results are summarized in this section.

As a testcase we have used Kleefsmans established dam-breaking test [19] considering 16 mio and 28 mio particles. All evaluations have been conducted on GPU systems composed of up to eight GPUs of type Nvidia GTX 1080 Ti, which contain 3584 CUDA cores with a memory bandwidth of 484 GB/s. The source code was compiled on Ubuntu v16.04 using gcc v4.5.3 and the CUDA Toolkit v9.1. The obtained results are shown in Table I, where the speedup (i.e. sequential execution time divided by the parallel execution time) and the efficiency (i.e. speedup divided by the number of GPUs) of the following parallel SPH methods (compared to their respective sequential counterparts) are summarized:

- A parallel version of WCSPH (whose absolute runtime is up to a magnitude larger than the runtime of the PCISPH method, but which is nevertheless considered to evaluate the speedup/efficiency possible through parallelization),
- a *standard* parallel version of PCISPH (which follows the established parallelization scheme used e.g. by WCSPH but yields the synchronization problems discussed in Section III), and
- the *proposed* parallel version of PCISPH (which, for the first time, addresses these problems in order to allow for a more efficient parallelization).

TABLE I: Results obtained by the experimental evaluation.

(a) Speedup and Efficiency using 28 million particles

	Speedup			Efficiency		
# GPUs	WCSPH	Standard PCISPH	Proposed PCISPH	WCSPH	Standard PCISPH	Proposed PCISPH
2 GPUs	1.74	1.63	1.64	0.87	0.81	0.82
4 GPUs	2.96	2.11	2.32	0.74	0.52	0.58
6 GPUs	4.03	2.4	3.00	0.67	0.40	0.50
8 GPUs	4.61	2.65	3.42	0.58	0.33	0.43

(b) Speedup and Efficiency using 16 million particles

	Speedup			Efficiency		
# GPUs	WCSPH	Standard PCISPH	Proposed PCISPH	WCSPH	Standard PCISPH	Proposed PCISPH
2 GPUs	1.77	1.54	1.56	0.89	0.77	0.78
4 GPUs	2.98	2.16	2.38	0.75	0.54	0.59
6 GPUs	3.68	2.33	2.65	0.61	0.39	0.44
8 GPUs	4.04	2.31	2.85	0.50	0.29	0.36

The results clearly confirm the discussions conducted above. WCSPH can indeed nicely be improved by parallelization. But corresponding speedups and efficiencies are never reached when the same (established) parallelization methods are employed to PCISPH. In fact, in case of e.g. 28 mio particles and eight GPUs, the efficiency drops from 0.58 (when WCSPH is applied) to 0.33 (when PCISPH is applied). This shows the effect of the problems discussed in Section III. While the alternative parallelization approach proposed in this work is not capable of completely avoiding these drawbacks, it significantly reduces the gap. In fact, following the proposed parallelization scheme increases the efficiency back to 0.43 – constituting a significant improvement. Also for all other cases, substantial improvements can be observed. Overall, the proposed parallelization scheme for PCISPH clearly outperforms the standard parallelization scheme and, by this, provides an alternative direction for exploiting HPC and GPGPU for this SPH variant.

VII. Conclusion and Future Work

In this work, we proposed a multi-GPU implementation for PCISPH simulations. To this end, we first discussed the algorithmic characteristics of PCISPH which led to severe synchronization problems that, thus far, prevented the efficient parallelization of PCISPH. Based on that, we developed an alternative approach which, as confirmed by our evaluations on top of an industrial tool, significantly reduces the gap to parallelization efficiency which is taken for granted by other SPH versions such as WCSPH. The main idea of the proposed approach rests thereby on an effective decomposition of the considered domain which allows for asynchronous memory transfers. By this, an SPH approach (namely PCISPH) which sequentially already outperforms other SPH approaches such as WCSPH by almost one order of magnitude eventually can further be improved by HPC and GPGPU parallelization – offering new prospects for optimizations. Future work includes a multi-node implementation as well as further study to completely close the scaling gap between WCSPH and PCISPH on multi-GPU architectures.

Acknowledgment

This work has been supported by the Austrian Research Promotion Agency (FFG) within the project Industrienahe Dissertationen 2016 under grant no. 860194.

References

[1] C. Chu, "Computational fluid dynamics," in *Numerical Methods for Partial Differential Equations*, 1979, pp. 149 – 175.

[2] H. K. Versteeg and W. Malalasekera, *An introduction to computational fluid dynamics: the finite volume method*. Pearson Education, 2007.

[3] G. Strang and G. J. Fix, *An analysis of the finite element method*. Prentice-hall Englewood Cliffs, NJ, 1973, vol. 212.

[4] L. Besra and M. Liu, "A review on fundamentals and applications of electrophoretic deposition (epd)," *Progress in Materials Science*, vol. 52, no. 1, pp. 1 – 61, 2007.

[5] K. Verma, L. Ayuso, and R. Wille, "Parallel simulation of electrophoretic deposition for industrial automotive applications," in *International Conference on High Performance Computing & Simulation*, 2018, pp. 1–7.

[6] J. Monaghan, "Smoothed particle hydrodynamics and its diverse applications," *Annual Review of Fluid Mechanics*, vol. 44, pp. 323–346, 2012.

[7] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible sph," in *ACM SIGGRAPH 2009*, 2009, pp. 40:1–40:6.

[8] S. Marrone, M. Antuono, A. Colagrossi, G. Colicchio, D. Le Touzé, and G. Graziani, "δ-sph model for simulating violent impact flows," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13-16, pp. 1526–1542, 2011.

[9] J. Dominguez, A. Crespo, and B. Rogers, "New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters," *Int. J. Computer Physics Communications*, vol. 184, pp. 1848–1860, 2013.

[10] E. Rustico, G. Bilotta, A. Herault, C. Negro, and G. Gallo, "Advances in multi-gpu smoothed particle hydrodynamics simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, 2014.

[11] K. Verma, K. Szewc, and R. Wille, "Advanced load balancing for SPH simulations on multi-GPU architectures," in *IEEE High Performance Extreme Computing Conference*, 2017, pp. 1–7.

[12] X. Nie, L. Chen, and T. Xiang, "Real-time incompressible fluid simulation on the gpu," *Int. J. of Computer Games Technology*, vol. 2015, 2015.

[13] J. Monaghan, "Smoothed particle hydrodynamics," *Rep. Prog. Phys*, vol. 68, pp. 1703–1759, 2005.

[14] M. Liu and G. Liu, "Smoothed particle hydrodynamics (sph): an overview and recent developments," *Arch. Comput. Methods Eng*, vol. 17, pp. 25–76, 2010.

[15] R. Gingoldand and J. Monaghan, "Smoothed particle hydrodynamics - theory and application to non-spherical star," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, 1977.

[16] A. Kolb and N. Cuntz, "Dynamic particle coupling for gpu-based fluid simulation," in *Int. Proc. of the 18th Symposium on Simulation Technique*, 2005, pp. 722–727.

[17] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on gpus," in *Proc. 5th Int. Conf. Computer Graphics*, 2007, pp. 63–70.

[18] A. Herault, G. Bilotta, and R. Dalrymple, "Sph on gpu with cuda," *Int. J. Hydraulic Research*, vol. 48, pp. 74–79, 2010.

[19] K. M. T. Kleefsman, G. Fekken, A. E. P. Veldman, B. Iwanowski, and B. Buchner, "A volume-of-fluid based simulation method for wave impact problems," *Journal of Computational Physics*, vol. 206, pp. 363–393, 2005.