Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2014)
J. Bender, C. Duriez, F. Jaillet, and G. Zachmann (Editors)

A Parallel Architecture for IISPH Fluids

Felix Thaler Barbara Solenthaler Markus Gross

Department of Computer Science, ETH Zurich, Switzerland

Abstract

We present an architecture for parallel computation of incompressible IISPH simulations on distributed memory systems. We use orthogonal recursive bisection for domain decomposition and present a stable and fast converging load balancing controller. The neighbor search data structure is derived such that it optimally fits into the parallel pipeline. We further show how symmetry aspects of the simulation can be integrated into the architecture. Simultaneous communication and computation are used to minimize parallelization overhead. The seamless integration of these parallel concepts into IISPH results in near linear scaling for large-scale simulations.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing;

1. Introduction

The visual quality of a particle-based fluid simulation depends on the spatial resolution which is used to represent the fluid. High-resolution simulations result in more complex surface and flow structures and dissipation artifacts are decreased. Recent work therefore aimed at increasing the performance of high-resolution, incompressible solvers, for example by improving the pressure computation [SP09, ICS*13, MM13] or by leveraging spatial adaptivity [APKG07, SG11, HS13]. Today's solvers such as the implicit incompressible SPH method (IISPH) [ICS*13] are capable of simulating several million particles resulting in stunning fluid effects and thus demonstrate that particle-based models have emerged to be a competitive technique for fluid animation [IOS*14].

Despite the prominent advantages of the IISPH method, the solver is much more complex compared to the standard, compressible SPH model used for example in [MCG03]. Hence, the inclusion of parallelization strategies to further improve the performance is not straightforward, especially if distributed memory platforms are considered. Furthermore, existing data structures for parallel execution of standard SPH on distributed systems such as in [FE08] cannot be directly applied to the incompressible IISPH method.

This paper aims at filling this gap by introducing a seamless integration of a parallel architecture into IISPH. Our implementation is targeted at distributed memory platforms using the standard message passing interface MPI and orthogonal recursive bisection ORB for domain decomposition. We use simultaneous communication and computation to minimize parallelization overhead. A robust load balancing controller is introduced that shows fast convergence, and we further demonstrate how neighbor search and symmetry aspects can be embedded into the parallel pipeline. Our results show near linear parallel scaling for IISPH simulations with up to 22 million particles.

2. Related Work

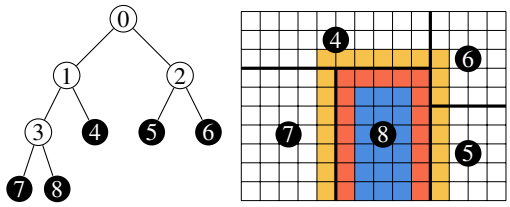
SPH Methods. Compressible SPH models [Mon92] have been used for interactive simulations of fluids for graphics applications in [MCG03]. Since then, many methods have been presented targeted at improving the performance. Various works addressed the problem of efficient pressure computation in incompressible particle methods, e.g. [SP09, BLS12, ICS*13, MM13]. To our knowledge, the state-of-the-art solver IISPH [ICS*13] is one of the fastest solvers for incompressible SPH. Thus, our work is using IISPH and shows how parallel architectures can be seamlessly integrated. Other strategies to increase the performance include adaptive methods [APKG07, OK12] and multi-scale models [SG11, HS13]. They follow the idea to allocate computing resources to interesting regions of the fluid only. A complete overview of SPH in Computer Graphics is given in the state-of-the-art report in [IOS*14].

GPU Implementations. In [HKK07] a single GPU implementation using OpenGL is proposed. All simulation

data including the neighbor search data structure is stored in texture memory. The SPH algorithm follows closely the one introduced in [MCG03]. A newer method proposed in [GSSP10] uses NVIDIA's CUDA for a more specialized GPU implementation. An efficient usage strategy of shared memory is introduced, which transfers particle data from global memory to shared memory during the computation of particle interactions. Several different optimization strategies for GPU-based SPH solvers are discussed in [DCGG11]. This includes the reduction of the cell size used for neighbor lookup to reduce code divergence, minimizing global memory lookup by storing all particle data of the current cell in shared memory, as well as grouping of neighbor cells to simplify neighbor search. Common to all these publications is that standard, compressible SPH solvers are considered but no predictor-corrector schemes for incompressible fluids as used in this work.

Multi-core CPU Implementations. A multi-core CPU implementation using fine-grained domain decomposition and a master-slave communication model was proposed in [HWT11]. Good scaling is reported, but no more than eight cores are considered. In [IABT11] a parallel implementation of predictive-corrective incompressible SPH (PCISPH) [SP09] is introduced. For neighbor search a parallel compact hashing implementation is proposed, but still a small amount of serial code is used to build the data structure. To get better locality of data accesses, particles are sorted by their Z-indices every 100th time step. With 130K particles, a speedup of about 10 was achieved on a system with 24 cores using PCISPH. General optimization strategies for SPH simulations on modern CPUs are again discussed in [DCGG11]. The proposed optimizations cover – besides parallelization – cache locality, SIMD vectorization and optimization of symmetric force computations.

Distributed Memory Implementations. For distributed memory platforms different parallelization strategies are needed, as only memory of the current processor is directly accessible. For short-range interactions as found in SPH domain decomposition methods are well suited. In [FE07] such a system for mixed discrete element and SPH simulations is proposed. Orthogonal recursive bisection (ORB) [Fox88] is used for domain decomposition. For communication, a master-slave model is introduced, where the master manages the data transfers between worker nodes. Load balancing is carried out by a hierarchical proportional-integral (PI) controller. Neighbor search is performed by hierarchical clustering, supporting adaptive particle sizes. A speedup of about 11 is achieved for a one million particle simulation using 16 processors. In [FE08], the same approach is described in more detail. The measurements show a convergence to a balanced state of the PI-controller-based load balancer in about 3000 time steps. Our work is closely related to [FE08]. In contrast, we apply ORB to the more complex IISPH solver with constant particle sizes [ICS*13] and show how load balancing and communication concepts can be seamlessly integrated into pressure projection schemes. Our load balancing controller differs from [FE08] as a faster convergence is achieved and the controller is easier to set up.



(a) ORB tree (b) Domain decomposition

Figure 1: *ORB tree and domain decomposition. (b) shows the cell types for process 8 used for communication: inner (blue), outer (red) and neighbor (yellow) cells.*

Probably the fastest SPH solver available today was introduced in [DCVB*13], making efficient use of GPU clusters. Here, a much simpler domain decomposition algorithm is used, i.e., the domain is just split along one axis. Still, this is probably the first work which enables SPH simulations with more than one billion particles to be computed within a reasonable amount of time. A classical weakly compressible SPH algorithm (WCSPH) [Mon92, BT07] is implemented.

3. Parallel Architecture

3.1. Recursive Domain Decomposition

We use Orthogonal Recursive Bisection (ORB) [Fox88] for spatial domain decomposition to distribute the particles to a constant number of computation nodes. The simulation domain is split recursively, and the subdomains corresponding to the leaf nodes represent the processes (see Figure 1). This domain decomposition scheme might also be seen as a kd-tree based splitting where leaf nodes contain several particles.

The division boundary on each level of the tree is axis-aligned and each process manages therefore a box-shaped subdomain. The division boundaries are placed such that all processes have approximately equal load. We follow [FE08] where the boundary position is determined by sampling the particles on a uniform grid. In this step, data from the neighbor search data structure can be reused. Distribution functions are then used to compute normalized cumulative particle density functions for the three spatial dimensions. When the best boundary position for each axis is found, the cost of the communication used on this axis can be estimated by counting the particles in the cells adjacent to the new division boundary candidate. The division boundary with the smallest number of adjacent particles is finally chosen to reduce communication costs during attribute synchronization.

Algorithm 1: Loop with nonblocking communication.

```
start synchronization of attribute b;
foreach inner particle i, neighbors j do
    compute attribute a_i depending on b_j;
wait for synchronization of attribute b;
foreach outer particle i, neighbors j do
    compute attribute a_i depending on b_j;
```

3.2. Communication

Efficient communication is an important part of the algorithm. As we focus on simulations with constant particle sizes (and thus constant support radii), only particles with a distance to a process' subdomain smaller than the support radius may be influenced by the particles owned by that process. In combination with a cell-based neighbor search algorithm we can classify the cells into three categories as shown in Figure 1, (b): *Inner* cells (blue) do never interact with particles from other processes' sub-domains. *Outer* cells (red) are adjacent to a sub-domain of any other process, and *neighbor* cells (yellow) are inside a sub-domain owned by a foreign process. Only particles in the own inner, outer and neighbor cells must be known by each process, other particles are therefore not stored locally.

To compute a particle attribute a, the loop traverses only cells owned by the respective process, i.e., inner and outer cells but not neighbor cells. If the computation of a depends on some other particle attribute b of neighbor particles, a synchronization of b between the processes is needed. For this, each process needs to send his outer particle data to and receive neighbor particle data from its neighbor processes. To avoid idle time, computation and communication is done simultaneously. As the inner cells are totally independent of the neighbors' data, it is possible to compute the inner cells' data during the communication. This is done by splitting the loop into two loops, one traversing the inner and one traversing the outer cells. This is outlined in Algorithm 1.

3.3. Load Balancing Controller

For an optimized performance idle states of processes should be avoided and thus work load needs to be continuously redistributed. In [FE07] and [FE08], boundaries of subdomains are shifted on each level of the ORB tree and thus particles are reassigned to neighboring domains. A hierarchical proportional-integral (PI) controller is used, taking as input the difference of waiting times of the processes on each level of the tree, and returning the position of the boundary division (rounded to cell boundaries of the neighbor search structure). As shown in [FE08], the discretized version of the analytical controller equation can be approximated as

$$\mathbf{x}_{inner}^{l+1} = \mathbf{x}_{inner}^{l} + P(\Delta w_{inner}^{l} - \Delta w_{inner}^{l-1}) + I\Delta t \Delta w_{inner}^{l}, \quad (1)$$

where l denotes the time step index, $\mathbf{x}_{inner}$ is the position of the boundary of the inner node, and Δw_{inner} is given as $w_{left} - w_{right}$, which is the difference of the waiting times of the right and left children. P and I are parameters of the proportional integral controller. Typically, conservative values need to be used to prevent instability. This leads, however, to slow convergence, which is also reflected in [FE08] where a well balanced situation is not found before the first 3000 time steps.

To improve the convergence we propose a robust heuristic controller. As in [FE08], the controller is hierarchically organized and uses the same controller in- and outputs. Division boundaries are moved by a linear factor of the waiting time difference of the child nodes. Again, the results are rounded to the nearest cell boundary. To minimize the probability of overshoots and instabilities, the correction of the splitting position is limited to a given maximum value. We compute the position of the boundary as

$$\mathbf{x}_{inner}^{l+1} = \mathbf{x}_{inner}^{l} + r, \quad (2)$$

where

$$r = \begin{cases} K\Delta w_{inner}^{l} & \left|K\Delta w_{inner}^{l}\right| \leq r_{max} \\ r_{max} & K\Delta w_{inner}^{l} > r_{max} \\ -r_{max} & K\Delta w_{inner}^{l} < -r_{max}. \end{cases} \quad (3)$$

The two controller parameters $K > 0$ and $r_{max} > 0$ define the speed and the maximum correction at one load balancing step. The parameters depend on the frequency of the load balancing steps. However, default parameters with r_{max} limited to one cell and K equals to one lead to good results. To minimize load balancing overhead but still get a balanced tree fast, we perform load balancing every 10th to 50th time steps. This frequency is scene dependent and in our implementation defined empirically by the user. In the future the need for load balancing might be recognized automatically depending on the current load imbalance. It is also advantageous to fully rebuild the ORB tree when topology has significantly changed, reducing the amount of transferred particle data during synchronization.

The computation of the new split positions is very efficient, as no knowledge is needed about the current particle distribution. Only the waiting times of the processes are used. The time intensive part of load balancing is indeed the communication, namely the redistribution of the particles. In fact this leads to minimal overhead in load balanced situations and is thus quite favorable.

3.4. Neighbor Search

We employ a neighbor search algorithm that is ideally fitted for use in conjunction with ORB. It is a combination of compact hashing as proposed in [IABT11] and any index sort method, preferably Z-index sort. Similar to index sort algorithms, the particles are always sorted by cell indices. While standard methods use an array of all cells in the domain with pointers to the respective particles, we only store

full cells. To allow for fast neighbor search nevertheless, an additional hash table is used, mapping cell indices to cells in the array of full cells. While the hashing and memory requirements are similar to compact hashing, the sorted particle storage makes it better suited for distributed memory implementations where cell-based communication is used, as the particle data for a cell is stored in contiguous memory.

A distinct data storage for fluid and boundary particles (representing solids and domain boundaries [AIS*12]) is used. This simplifies communication as only fluid attributes need to be synchronized. For neighbor search two data structures are kept, one for fluid and one for boundary particles.

3.5. Symmetry Optimizations

As in SPH forces are symmetric between two neighboring particles [DCGG11], we integrate symmetry optimizations into our parallel solver. The cell-based neighbor search can be modified accordingly by looping over 14 neighbor cells instead of 27 in three dimensions. For outer cells, i.e., cells adjacent to the subdomain boundary, stencils must be adapted accordingly to ensure that all cells of neighbor processes are employed in the computation.

Note that calculations for the current cell in the loop are not yet finished when moving on to the next cell as neighbor cells still contribute. In other words, calculations on a given cell in the loop can influence cells that have already been visited before. There are two possibilities to solve this problem: First, the order of cell traversal could be changed. This limits the traversal to simple linear loops as used with linear index mapping. Second, dependencies on values computed in the current loop are prevented. This allows arbitrary loop sequences but requires that some loops are split into two separate traversals. We make use of the loop splitting in the parallel fluid implementation discussed in Section 4.

4. Implementation of Parallel IISPH

In implicit incompressible SPH (IISPH) a density invariance condition is employed and the pressure field is computed iteratively with relaxed Jacobi. The algorithm can be split into three steps: Prediction of advection using non-pressure forces, pressure solve and time integration. Our parallel implementation follows closely the algorithm outlined in [ICS*13]. For more details on the algorithm we refer to the aforementioned paper. For each of the three steps we introduce the parallel implementation in the following.

The prediction of the advection is outlined in Algorithm 2. In contrast to [ICS*13], the density must be computed in a separate loop due to the dependency of the viscous and surface tension forces on neighbor particles' densities ρ. The second loop, where the intermediate velocities $\mathbf{v}^{adv}$ and displacements $\mathbf{d}_{ii}$ are evaluated, is split in an inner and outer part. During the computations on the inner cells the density

Algorithm 2: Parallel IISPH: Prediction of advection.

foreach *particle i* **do**
  compute density ρ_i;
`// divided into inner and outer loop`
start synchronization of density;
foreach *inner particle i* **do**
  compute $\mathbf{v}_i^{adv}$ and $\mathbf{d}_{ii}$;
wait for synchronization of density;
foreach *outer particle i* **do**
  compute $\mathbf{v}_i^{adv}$ and $\mathbf{d}_{ii}$;
`// divided into inner and outer loop`
start synchronization of $\mathbf{v}^{adv}$ and $\mathbf{d}_{ii}$;
foreach *inner particle i* **do**
  compute ρ_i^{adv} and a_{ii};
wait for synchronization of $\mathbf{v}^{adv}$ and $\mathbf{d}_{ii}$;
foreach *outer particle i* **do**
  compute ρ_i^{adv} and a_{ii};

Algorithm 3: Parallel IISPH: Pressure solve.

$l \leftarrow 0$;
while $\rho_{avg}^l - \rho_0 > \eta \vee l < 2$ **do**
  `// divided into inner and outer loop`
  start synchronization of pressure;
  foreach *inner particle i* **do**
    compute $\sum_j \mathbf{d}_{ij} p_j^l$;
  wait for synchronization of pressure;
  foreach *outer particle i* **do**
    compute $\sum_j \mathbf{d}_{ij} p_j^l$;
  `// divided into inner and outer loop`
  start synchronization of $\sum_j \mathbf{d}_{ij} p_j^l$;
  foreach *inner particle i* **do**
    compute intermediate sum $\sum_{j \neq i} a_{ij} p_j^l$;
  wait for synchronization of $\sum_j \mathbf{d}_{ij} p_j^l$;
  foreach *outer particle i* **do**
    compute intermediate sum $\sum_{j \neq i} a_{ij} p_j^l$;
  `// used due to symmetry optimization`
  foreach *particle i* **do**
    compute p_i^{l+1};
  compute global ρ_{adv}^{l+1};
  $l \leftarrow l + 1$;

is synchronized with neighbor nodes. Analogously, the next loop is split into an inner and outer part to compute the predicted density ρ^{adv} and coefficients a_{ii}. The pressure solve is shown in Algorithm 3. The original algorithm of [ICS*13] consists of two loops over all particles; one for summing up $\mathbf{d}_{ij} p_j^l$, which is the movement of particle i caused by the neighboring pressure value p_j, and the second for computing the pressure p_i^{l+1}. We source out some of the computations

Algorithm 4: Parallel IISPH: Time integration.

```
start synchronization of pressure;
foreach inner particle i do
    compute pressure forces;
wait for synchronization of pressure;
foreach outer particle i do
    compute pressure forces;
// used due to symmetry optimization
foreach particle i do
    integrate;
redistribute particles to nodes;
update neighbor search and communication structures;
```

into separate loops due to optimization of symmetric calculations and better spatial locality of variables in memory. Further, due to the dependencies on previous loops' neighbor data, we split up the first two loops of Algorithm 3 in an inner and outer part. And last, the time integration and particle redistribution is computed as shown in Algorithm 4.

5. Results and Discussion

Scaling and Performance. For a dam break scenario with different resolutions ranging from 0.4M to 22M particles we measured the scaling performance using eight cluster nodes, each of them with four sockets and 2.5GHz quad core processors, leading to a maximum number of 128 cores. The results for parallel IISPH are shown in Figure 2. The performance of parallel IISPH scales nearly linearly up to 64 processes. The best scaling is reached with 22M particles. With smaller particle counts it can be observed that communication overheads dominate, i.e., synchronization of the outer particles' attributes need more time than looping over the inner particles. The same effect is observed if too many processes are used, i.e., with a small ratio of inner to outer particle count per process.

The average computation time per time step for different particle numbers is shown in Figure 3. For 64 processes and 22M particles, for example, the computation time per time step is approximately 10s.

Load Balancing. The resulting domain decomposition of one simulation step of the dam break example is shown in Figure 4. Each color corresponds to one process. The proposed heuristic-based controller allows a fast convergence to a well balanced tree: In this example, convergence was reached after 50-100 simulation steps. In Figure 5 the differences to the average waiting time of all MPI processes are shown. The dam break simulation was run on two computing nodes with 48 cores each, leading to a total of 96 cores. The influence of the periodical tree rebuilds (every 50th load balancing step / 1000th time step) can be seen clearly, as the waiting time differences

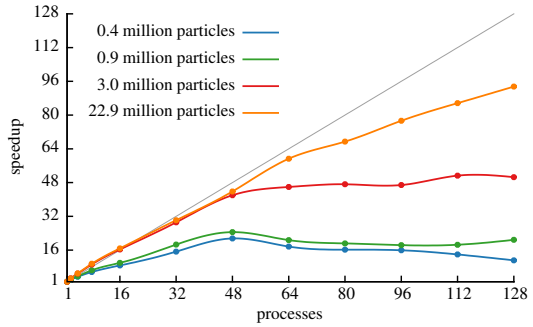


Figure 2: *Scaling of our parallel IISPH implementation for different numbers of processes and particle counts. IISPH shows a nearly linear scaling up to 64 processes.*

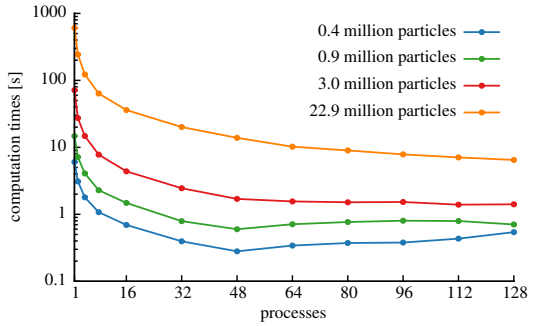


Figure 3: *Average computation time per time step in parallel IISPH for different numbers of processes.*

are increased in the same frequency as well. The used tree construction algorithm is able to distribute the particles evenly, but cannot guarantee fair communication costs, hence some load balancing steps are needed to go back to a fully balanced state.

Further it may be noticed that load balancing seems to work better in the later stages of the simulation. This is due to the very regularly positioned particles of the initial state: The perfectly cell-aligned particles may lead to very abrupt changes of waiting times when division boundaries are moved by only one cell. This makes it very difficult for the controller to reach a well balanced state, while in later stages of the simulation the particles are distributed much more randomly, so small modifications on the boundary positions lead to small changes of the waiting times, too.

Note that we omit a comparison to the PI-controller of [FE07, FE08] as no controller parameters could be found that worked well in general. Only a slow balancing could be achieved without loosing stability. The slow convergence (3000 steps are stated in [FE08]) is especially problematic for simulations with large changes in the particle distribution (such as in the dam break scenario) as they need periodic tree rebuilds to keep the communication minimal.

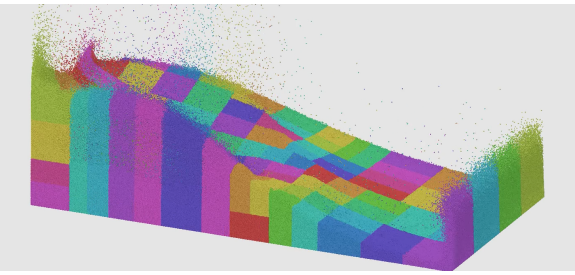


Figure 4: *Domain decomposition for one simulation step of the dam break example. Each color represents one process.*

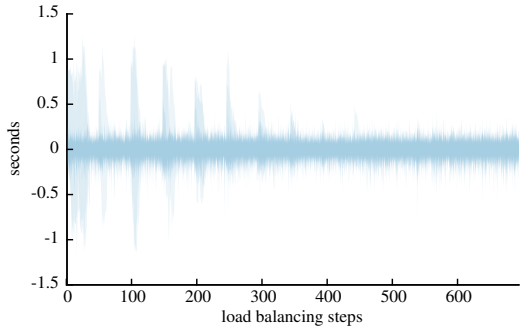


Figure 5: *Waiting time differences of all 96 processes to the average. ORB rebuilds are well visible as periodical outliers.*

6. Conclusion

We have presented a parallel framework suitable for state-of-the-art incompressible SPH schemes such as IISPH. We have shown that these modern SPH algorithms can be efficiently parallelized to use the full power of current distributed memory hardware. To leverage the parallelism of cluster computers, ORB was used for domain decomposition. A novel heuristic load balancing controller for ORB was introduced, which gives faster convergence to a balanced state than previous solutions and thus leads to lower simulation times. With the use of simultaneous communication and computation, the overhead of communication between the compute nodes was minimized. Using our implementation, good scaling was achieved on large-scale simulations.

References

[AIS*12] AKINCI N., IHMSEN M., SOLENTHALER B., AKINCI G., TESCHNER M.: Versatile rigid-fluid coupling for incompressible SPH. *ACM Transactions on Graphics (Proceedings SIGGRAPH) 30*, 4 (2012), 72:1–72:8. 4

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L.: Adaptively sampled particle fluids. In *ACM Transactions on Graphics (Proc. SIGGRAPH)* (2007), vol. 26, pp. 48:1–48:7. 1

[BLS12] BODIN K., LACOURSIRE C., SERVIN M.: Constraint fluids. *IEEE Transactions on Visualization and Computer Graphics 18*, 3 (2012), 516–526. 1

[BT07] BECKER M., TESCHNER M.: Weakly compressible SPH for free surface flows. In *Proc. of Symposium on Computer Animation* (2007), pp. 209–217. 2

[DCGG11] DOMINGUEZ J. M., CRESPO A. J. C., GOMEZ-GESTEIRA M.: *Optimization strategies for parallel CPU and GPU implementations of a meshfree particle method*. arXiv e-print 1110.3711, 2011. 2, 4

[DCVB*13] DOMINGUEZ J., CRESPO A., VALDEZ-BALDERAS D., ROGERS B., GOMEZ-GESTEIRA M.: New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications 184*, 8 (2013), 1848–1860. 2

[FE07] FLEISSNER F., EBERHARD P.: Load balanced parallel simulation of particle-fluid dem-sph systems with moving boundaries. *In Proceedings of Parallel Computing: Architectures, Algorithms and Applications* (2007), 37–44. 2, 3, 5

[FE08] FLEISSNER F., EBERHARD P.: Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. *International Journal for Numerical Methods in Engineering 74*, 4 (2008), 531–553. 1, 2, 3, 5

[Fox88] FOX G. C.: A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures*, no. 13. 1988, pp. 37–61. 2

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive SPH simulation and rendering on the GPU. In *Proc. of Symposium on Computer Animation* (2010), pp. 55–64. 2

[HKK07] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Smoothed particle hydrodynamics on GPUs. In *Proceedings of Computer Graphics International* (2007), pp. 63–70. 1

[HS13] HORVATH C. J., SOLENTHALER B.: Mass preserving multi-scale SPH. Pixar Technical Memo 13-04, Pixar Animation Studios, 2013. 1

[HWT11] HOLMES D. W., WILLIAMS J. R., TILKE P.: A framework for parallel computational physics algorithms on multi-core: SPH in parallel. *Advances in Engineering Software 42*, 11 (2011), 999–1008. 2

[IABT11] IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A parallel SPH implementation on multi-core CPUs. *Computer Graphics Forum 30*, 1 (2011), 99–112. 2, 3

[ICS*13] IHMSEN M., CORNELIS J., SOLENTHALER B., HORVATH C., TESCHNER M.: Implicit incompressible SPH. *IEEE Trans. on Visualization and Computer Graphics* (2013). 1, 2, 4

[IOS*14] IHMSEN M., ORTHMANN J., SOLENTHALER B., KOLB A., TESCHNER M.: SPH Fluids in Computer Graphics. In *Eurographics 2014 - STARs* (2014), pp. 21–42. 1

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *Proc. of Symposium on Computer Animation* (2003), pp. 154–159. 1, 2

[MM13] MACKLIN M., MUELLER M.: Position based fluids. *ACM Trans. on Graphics (Proc. SIGGRAPH) 32* (2013), 1–5. 1

[Mon92] MONAGHAN J.: Smoothed particle hydrodynamics. *Ann. Rev. Astron. Astrophys. 30* (1992), 543–574. 1, 2

[OK12] ORTHMANN J., KOLB A.: Temporal blending for adaptive SPH. *Computer Graph. Forum 31*, 8 (2012), 2436–2449. 1

[SG11] SOLENTHALER B., GROSS M.: Two-scale particle simulation. *ACM Transactions on Graphics (Proceedings SIGGRAPH) 30*, 4 (2011), 72:1–72:8. 1

[SP09] SOLENTHALER B., PAJAROLA R.: Predictive-corrective incompressible SPH. *ACM Transactions on Graphics (Proceedings SIGGRAPH) 28* (2009), 40:1–40:6. 1, 2