Kokkos-Based Implementation of MPCD on Heterogeneous Nodes

Rene Halver[1], Christoph Junghans[2], and Godehard Sutmann[1,3(✉)]

[1] Jülich Supercomputing Centre, Institute for Advanced Simulation, Forschungszentrum Jülich, 52425 Jülich, Germany
{r.halver,g.sutmann}@fz-juelich.de
[2] Los Alamos National Laboratory, CCS-7, 87545 Los Alamos, NM, USA
junghans@lanl.gov
[3] ICAMS, Ruhr-University Bochum, 44801 Bochum, Germany

Abstract. The Kokkos based library Cabana, which has been developed in the Co-design Center for Particle Applications (CoPA), is used for the implementation of Multi-Particle Collision Dynamics (MPCD), a particle-based description of hydrodynamic interactions. It allows a performance portable implementation, which has been used to study the interplay between CPU and GPU usage on a multi-node system. As a result, we see most advantages in a homogeneous GPU usage, but we also discuss the extent to heterogeneous applications, using both CPU and GPU concurrently.

Keywords: Kokkos · Multi-particle collision dynamics · GPU-computing · particle simulations · performance portability

1 Introduction

The recent development of high-end parallel architectures shows a clear trend to a heterogeneity of compute components, pointing towards a dominance of General Purpose Graphics Processing Units (GPU) as accelerator components, compared to the Central Processing Units (CPU). According to the Top 500 list [4], more than 25% of the machines have GPU support while the overall performance share is more than 40%, i.e., heterogeneous cluster architectures have a large impact for high compute performance. Often these nodes consist of only a few multicore CPUs, while supporting 2–6 GPUs. In many applications one can observe a trend that the most powerful component of the nodes, i.e. the GPUs, is addressed, while the CPUs are used as administrating or data management components. A reason might be the additional overhead in writing/maintaining two different code versions for each architecture, as usually a CPU code cannot simply run on a GPU or vice versa.

With the advent of performance portable programming models, such as Kokkos [6] or Raja [16] it has become possible to use the same code base for different architectures, most prominently including CPUs or GPUs. It might

R. Wyrzykowski et al. (Eds.): PPAM 2022, LNCS 13827, pp. 3–13, 2023.
https://doi.org/10.1007/978-3-031-30445-3_1

be tempting to use the full capacity of a compute-node concurrently, i.e. not wasting compute resources because of the disparate character of the architecture and programming model. In this case one encounters both different performance characteristics of components and possibly a non-negligible data transfer between components. This discrepancy might be targeted by load balancing strategies which would need to take into account hardware and software specific characteristics to achieve an overall performance gain.

In the present paper we consider a stochastic particle based method for the simulation of hydrodynamic phenomena, i.e. the Multi-Particle Collision Dynamics (MPCD) [8] algorithm and its implementation with Cabana [2,14,17], a Kokkos based library. We first introduce the underlying MPCD method and then describe the Cabana library. We then present some benchmark results and finally draw conclusions from our findings and give some outlook for further research.

2 Multi-Particle Collision Dynamics

MPCD is a particle-based description for hydrodynamic interactions in an incompressible fluid. The method is based on a stochastic collision scheme in which particles, that describe the simulated fluid are rotated in velocity space while conserving linear momentum and energy (variants exist which also conserve angular momentum [8]). The method proceeds by sorting particles into a regular mesh with grid cells of size of a characteristic length scale. In order to transport momentum and energy across the system, the mesh is randomly shifted in each time step, changing the local environment of each particle stochastically. For each particle in a cell its relative velocity with respect to the center-of-mass (*com*) velocity of the cell is computed. This velocity is split into a parallel and perpendicular component with respect to a randomly oriented axis in the cell. Consequently, the perpendicular component is rotated around that axis by a fixed angle, which determines together with the particle mass and density, the time step and the cell length the diffusion and viscosity of the fluid under consideration. This procedure can be shown to mimic hydrodynamic behaviour and, in a limiting case, enters into the Navier Stokes equations [8]. Using this procedure the conservation of linear momentum and energy is guaranteed and can also be coupled to embedded particles, simulated by other methods, e.g. molecular dynamics, thereby coupling particle dynamics to a hydrodynamic medium [8,12].

From an algorithmic point of view, three main parts can be identified, i.e. (i) the local identification of particles in the underlying cell structure and the computation of *com* velocities of cells; (ii) the computation of the relative velocities of particles with respect to the *com* velocity of a cell; (iii) rotation of perpendicular velocity component of particles around a random axis. These parts will be discussed separately in Sect. 3 in more detail in the context of the Cabana implementation.

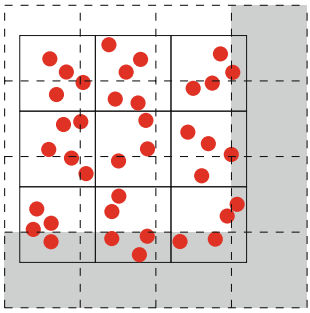


Fig. 1. Illustration of the shifted collision cell grid (black, dashed) in comparison to the static logical cell grid (black, solid). The grey cells mark the periodic images of the shifted grid. (Color figure online)

3 Implementation with Cabana

The aim of the implementation was to write a code, that is performance portable between clusters consisting of CPU and clusters with GPU nodes, which often consist of one or two CPUs and a number of GPUs ranging from two to six. Maintaining two or more codebases for all targeted architectures increases the overhead time of, e.g., design or maintenance time, and calls for solutions which allow a unified approach for various architectures.

For this reason performance portable programming models are attractive for reducing time spent with porting codes to various architectures. One of the more popular programming models in this regard is Kokkos [6], which provides an abstraction layer for data structures, called *Views*, while providing different *ExecutionSpaces* which can either be on the host (usually the CPU) or on devices, i.e. GPUs or other accelerator cards, e.g. Intel KNLs. Kokkos uses different backends to provide this performance portability, e.g. CUDA for the use of NVIDIA GPUs or ROCm for the use of AMD GPUs. Furthermore, OpenMP or PThread backends can be used among others to utilise multicore architectures of CPUs.

Within the Exascale Computing Project (ECP) [5] funded by the Department of Energy (DoE) in the USA, the Co-Design Center for Particle Applications (CoPA) [3] developed a performance portable library, based on Kokkos, with the main focus of supporting the development of particle and grid based codes on HPC systems. Cabana not only provides data structures based on Kokkos *Views* but also provides routines in order to facilitate data transfer between different processes in a distributed-memory environment, based on MPI.

Since the MPCD method is a mixture of a particle and a grid based method (due to the requirement to sort the particles into cells), the implementation of the MPCD code using Cabana was considered reasonable. In the rest of the section the main points of the implementation will be presented.

3.1 Collection of Particles in Cells

Before the *com* velocity for a cell can be calculated, it is necessary to identify the particles that reside in each collision cell. One technique to achieve this is the linked-cell list. Accordingly, all particles are checked and flagged with a cell identifier to which they belong to. In addition, a (linked) list of particles belonging to the cell is created. Listing 1.1 shows how such a list is created in Cabana. The use of Cabana simplifies the creation of such a linked cell list, as Cabana deals with the issues of creating a linked cell list in a multithreaded environment, as described e.g. in [11] or [15].

Listing 1.1. Creation of the linked cell list of the shifted collision cell grid

```
// boundaries of spacial domains
double gridMin[3], gridMax[3];
for (int d = 0; d < 3; ++d)
{
  gridMin[d] = domBorders(2*d) - (double)haloWidth
               * cellSize(d) + offset(d);
  gridMax[d] = domBorders(2*d+1) + (double)haloWidth
               * cellSize(d) + offset(d);
}
// creating the linked cell list
// r = list of particle positions
// cellSize = size of linked cells (3d)
Cabana::LinkedCellList<DeviceType>
  linkedList( r, cellSize, gridMin, gridMax );
// permute the particle AoSoA to correspond to the cells
Cabana::permute( linkedList, particles );
```

3.2 Communication of Required Information

As described in Sect. 2, it is necessary to compute the *com* velocity, i.e. the velocity in a zero momentum frame with regard to the local collision cell [7], in order to calculate the collisions within each mesh cell, which requires all velocities and masses of particles that reside within the given collision cell. The underlying parallel algorithm is based on a domain decomposition, where compute resources administrate geometrical spatial regions which are connected. Since the underlying mesh is shifted in each time step cells might be split among several domains. To compute a unique value for the *com* velocity, one can either collect all particles together with their properties on a local domain or one can compute the partial *com* velocities on each local domain and then reduce this value among those processes which share the given cell.

The first of these methods has the advantage that since all particles are collected on a single domain, the computation of the *com* velocity and the following rotation of velocities can be executed without the need of additional communication steps in between. The disadvantage is that it requires the communication of particle data in each time step, since the collision cell mesh needs to be shifted in

each time step to avoid artefacts in the computation of the hydrodynamic interactions. Listing 1.2 shows the necessary steps to prepare the particle migration between domains. Shown here is a way to try to avoid unnecessary branching while determining the target processes for particles. This is done by masking the target processes with a base-3 number, where each 'bit' indicates either a shift down(0) or up (1) or residing in the domain's boundary concerning that Cartesian direction. As an example a base-3 number of $(201)_3$ would be assigned to a particle leaving the local domain in positive x-direction and negative y-direction, while stay in the same z-region, as the local domain. This way to determine target processes should improve execution on GPU, with the tertiary operator being removed, in case that true is cast to integer one and false to integer zero.

Listing 1.2. Particle based communication with Cabana

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<ExecutionSpace>(0, nParticles),
  KOKKOS_LAMBDA (const size_t i)
  {
    int dims = 1, index = 0;
    // compute the direction of the neighbour the particle
    // needs to be moved to and use dims to compute a
    // base 3 mask:
    // (xyz)_3 with 0 (left), 1 (remains), 2 (right)
    // r = list of particle positions
    for (int d = 2; d >= 0; --d)
    {
        index += dims *
          ( 1 - ((r(i,d) < domBorders(2*d))?1:0) +
          ((r(i,d) >= domBorders(2*d+1))?1:0) );
        dims *= 3;
    }
    // tag the particle with the target neighbour rank
    export_ranks(i) = neigs(index);
  });
Kokkos::fence();

// create particles distribution object and
// migrate particles to targets
Cabana::Distributor<DeviceType> dist( mpiCart,
                      export_ranks, neighbours );
Cabana::migrate(dist, particles);
```

In contrast, the second method allows the use of a stable, halo-based communication scheme, where particles are not necessarily communicated in each time step, but only when leaving a halo region around the local domain, allowing the distributed computation of partial *com* velocities, that are reduced with a static communication scheme. The result is then sent back to the domains sharing the same cell. Listing 1.3 shows the required function calls to Cabana to do the halo exchange. This work, related to mesh administration, is implemented in *Cajita*,

which is part of Cabana. In addition, it provides methods for particle-grid interactions, e.g. interpolation of particle properties to a grid, which is, however, not used in this work. Furthermore, Cajita provides a domain-based load balancing based on a tensor decomposition scheme, provided by the ALL library [9].

Listing 1.3. Grid based halo communication with Cabana

```
// create the halo communication object based
// on the Cajita grid
auto arrHalo = Cajita::createHalo( *arrNode,
                Cajita::NodeHaloPattern<3>());
// [...] computation of com velocities
// bring the data to the halo cells
arrHalo->gather(ExecutionSpace(), *arrNode);
// collect the data from the halo cells
arrHalo->scatter(ExecutionSpace(),
     Cajita::ScatterReduce::Sum(), *arrNode);
```

For the implementation of the two different communication schemes two different kinds of communication in Cabana were used. For the former method, the particle-based one, Cabana provides a *Distributor* class, which allows the transfer of particle data between processes. This requires that particles are tagged with the target process, so that the *Distributor* object can generate a communication topology for this specific transfer. As a consequence this object needs to be recreated in every time step, since the communication pattern in each time step changes due the random shift of the collision cell grid and particle movements across domain borders.

For the second communication pattern, reducing the partial results and redistributing them, a halo-based communication on a grid is used. For this purpose, two different grids are combined, i.e. a logical collision grid which is used for communication and a linked-cell list, which sorts the particles into the shifted collision cell grid. Since the number and size of mesh cells in each grid is identical, both grids can be perfectly matched onto each other. The particles are sorted into the linked-cell list (Sect. 3.1) from where the *com* momentum of each cell is computed. For collision cells, overlapping with domain borders (Fig. 1), a halo-based communication reduces the partial results on the process which administrates the logical cell. This process redistributes the reduced sum back to each participating neighbour, where the rotations of velocities are computed for residing particles. Since the number of cells is usually far smaller than the number of particles, this leads to (i) a static communication scheme (for each iteration step the same operations on the same amount of data) and (ii) a reduced and constant amount of data that needs to be communicated.

During the development, it became apparent that the second communication scheme leads to a better performance due to the reduced amount of transferred data and the strongly reduced necessity to recreate communication patterns, due to the stable communication scheme of the halo exchange (this needs to be done

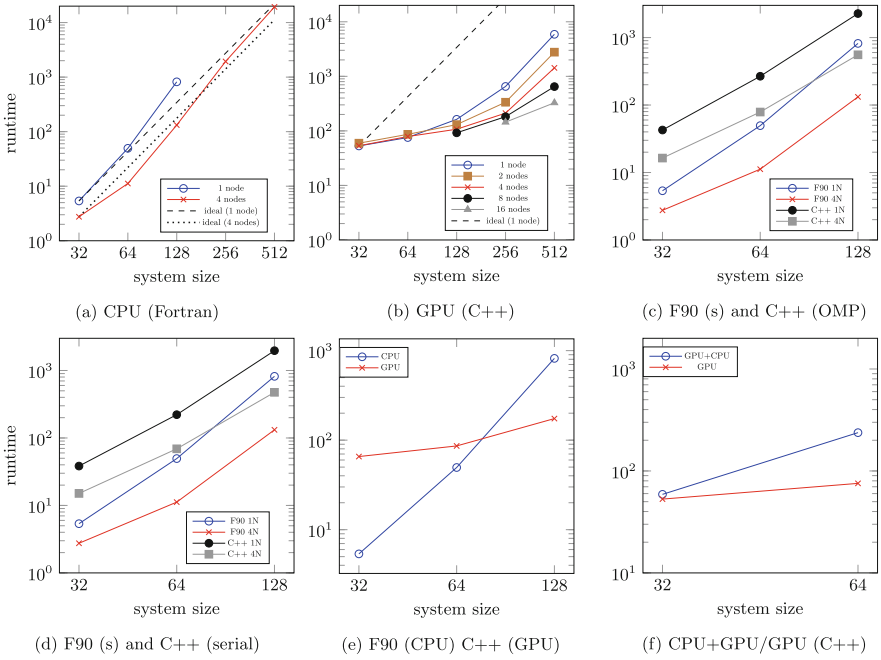
only once in the beginning or after possible load balancing steps, after which the communication pattern is static). In addition, the transfer of particles can be reduced to cases, where particles left the halo region surrounding the local domain, instead of being required in every time step.

3.3 Rotation of Velocities

To simplify the computation of the velocity rotation, the linked cell list mentioned in Sect. 3.1 is used to sort particles into the correct cell of the collision cell grid. Using the *com* velocity, gathered by one of the two previously described methods, the linked-cell list provides the particles which belong to the given cell and their velocity vector rotated.

Listing 1.4. Using the linked cell list from listing 1.1 to compute the com velocity

```
// Kokkos parallel_for iterates over
// all cells on local domain
// vcm = Kokkos::View containg the center
//       of mass velocites for each
//       collision cell
// v   = Cabana::slice containing
//       particle velocities
// m   = Cabana::slice containing
//       particles masses
Kokkos::parallel_for(Kokkos::RangePolicy<ExecutionSpace>
  (0, linkedList.totalBins()),
  KOKKOS_LAMBDA( const size_t i)
  {
    int ix, iy, iz;
    // computing the cartesian coordinates of the cell
    linkedList.ijkBinIndex(i, ix, iy, iz);
    int binOff = linkedList.binOffset(ix, iy, iz);
    // compute com velocity
    for (int d = 0; d < 4; ++d)
      vcm(ix,iy,iz,d) = 0.0;
    // computing com momentum and sum of mass
    for (int n = 0; n < linkedList.binSize(ix,iy,iz); ++n)
    {
      for (int d = 0; d < 3; ++d)
        vcm(ix,iy,iz,d) += v(binOff + n, d) *
                           m(binOff + n);
      vcm(ix,iy,iz,3) += m(binOff + n);
    }
  });
Kokkos::fence();
```



(a) CPU (Fortran) (b) GPU (C++) (c) F90 (s) and C++ (OMP)

(d) F90 (s) and C++ (serial) (e) F90 (CPU) C++ (GPU) (f) CPU+GPU/GPU (C++)

Fig. 2. Performance comparison between existing Fortran implementation and new Cabana implementation using multiple nodes.

4 Benchmarks and Discussion

For the benchmark runs simple fluid systems were used, i.e. a pure MPCD fluid in 3d periodic boundary conditions. Each cubic collision cell has an edge length of one length unit, while containing $\langle N_c \rangle = 10$ particles on average. Each system in the benchmarks is cubic with side length L (the edge length L given as the system size in the following graphs, i.e. Fig. 2), from where the total number of particles in a system is computed as $N = L^3 \langle N_c \rangle$. To check the performance of the newly implemented code, it was compared to an existing Fortran implementation of the MPCD algorithm [12,18].

The benchmarks were performed on the Juwels booster module [13] at Jülich Supercomputing Centre, consisting of GPU nodes with four NVIDIA A100 cards and two AMD EPYC 7402 processors, with 24 cores each. To maintain comparability of the benchmarks the pure CPU runs were also performed on these nodes. Since the GPU nodes are much more powerful in their computing capabilities, we performed the benchmarks for the GPU runs on node numbers from one to 16, doubling the node count each time. For the CPU, expecting longer runtimes we chose to compare single node runs with runs on four nodes, while also restricting the system size to a maximum edge length of 128 while for the GPU runs we performed the benchmarks to a maximum edge length of 512. The edge length directly influences the number of particles in the simulation, since

Table 1. Tables of runtimes for the different implementations. Empty cells indicate combinations of node numbers and system sizes, for which additional measurements did not show additional information. All runtimes presented are given in seconds.

(a) Runtimes for GPU C++ variant, using 4 GPUs on each node.

size	1 node	2 nodes	4 nodes	8 nodes	16 nodes
32	52.96	59.47	53.36		
64	75.55	86.48	78.77		
128	162.88	130.27	107.29	92.11	
256	652.25	335.12	211.51	182.18	144.60
512	5897.44	2774.35	1429.14	648.252	327.49

(b) Runtimes for CPU variants, Fortran (F90) and the C++ based variants, i.e. OpenMP-based (OMP) or serial, i.e. no hybrid parallelization, using one or four nodes (N). OMP uses 8 MPI ranks with 6 threads each on a node, the Fortran and serial version 48 MPI ranks per node. Only system sizes up to edge length 128 are presented due to the longer runtimes.

size	F90 1 N	F90 4 N	OMP 1 N	OMP 4 N	serial 1 N	serial 4 N
32	5.36	2.75	42.62	16.37	38.33	15.06
64	49.59	11.20	284.73	78.79	221.38	69.06
128	819.06	132.06	2268.80	555.42	1967.42	474.67

there are about l^3 collision cells in the system, with l being the edge length of the system, each collision cell containing ten particles on average.

As backends for Kokkos were the AMD and Ampere70 used, since these corresponded best to the available hardware. No further optimization on the basis of compiler flags was attempted yet due to time constraints, but these tests will be performed in the future. Table 1 and Fig. 2 show results for four different benchmarks: (i) C++/Kokkos implementation with GPU variant (Table 1a and Fig. 2b); (ii) C++/Kokkos variant with OpenMP (Table 1b and Fig. 2c); (iii) C++/Kokkos variant with serial backend and (iv) the previous implementation of the MPCD algorithm in Fortran (Table 1b and Fig. 2a) for comparison with the new implementation.

The original Fortran code shows a quite good scaling behaviour for all studied cases (edge lengths $L \in [32, 512]$), as can be seen in Fig. 2a. In comparison to that the scaling behaviour of the GPU variant of the C++ implementation shows for the smaller system sizes a super-linear scaling behaviour, before reaching linear behaviour at system sizes 256 and 512, indicating that smaller sizes not fully utilise the GPU (Fig. 2b).

When comparing the performance of the Fortran implementation (Fig. 2a) and the CPU based variants of the C++ version, i.e. OpenMP based or serial, it can be seen that Fortran achieves much better results (Figs. 2c, 2d). An expla-

nation for this behaviour still needs to be analysed in more depth. But first results point towards a different level of optimization (which is not the main focus of this article). In contrast, the GPU variant is able to outperform the Fortran implementation given sufficiently large system sizes, as can be seen in Fig. 2e, comparing the benchmark results on a single node, respectively. Here only the results for system sizes 32 and 64 are shown, since the measurement strongly hint that for larger system sizes the gap between hybrid execution and pure GPU execution will only widen.

Furthermore, it was tested on a single node if the combination of GPU and CPU could result in a better performance than only GPU computations. Due to the obtained performance of the CPU-based C++ variants, the results indicate at this stage no performance gain for hybrid execution (Fig. 2f). In case of a performance improvement of the CPU-based variants, this result might change for smaller system sizes. Note that for small systems load balancing GPU and CPU ranks can improve the overall performance for hybrid execution significantly, but not sufficiently in order to outperform either pure CPU or GPU. This does not lead to a recommendation of a hybrid execution model at this stage.

5 Conclusion and Outlook

Considering the benchmark results of the new implementation of the MPCD code the following conclusions can be drawn:

(i) It is possible to implement a scalable MPCD algorithm with Cabana, that for large enough systems is faster on GPUs than the existing Fortran implementation. The CPU variant of the Cabana implementation needs to be improved upon to bring the performance closer to the one of the Fortran code.
(ii) Load balancing between CPU and GPU can support hybrid execution, but was not found to increase performance beyond the one of pure CPU or GPU usage.
(iii) The porting effort from a pure CPU variant to a multi-architecture variant was significantly decreased by using Cabana, which offers an architecture independent development and code implementation which provides a unified and transparent view for the programmer. Porting effort is therefore dramatically reduced by maintaining performance (which was not the focus here, but which is demonstrated for other use cases [1,6,10]).
(iv) The implementation of the MPCD algorithm allows further investigation of coupled simulations of MPCD fluids with embedded Molecular Dynamics (MD) systems, e.g. polymer chains. For this, an implementation based on a unified formulation of MD and MPCD, as described, e.g., in [8,12], is required. Since the ratio of MD- to MPCD particles is often small, this could profit from a hybrid implementation and execution model, which invites to further investigations, including execution models for modular supercomputing.

References

1. Artigues, V., Kormann, K., Rampp, M., Reuter, K.: Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. Concurr. Comput. Pract. Exp. **32**(11), e5640 (2020). https://doi.org/10.1002/cpe.5640
2. Cabana. https://github.com/ECP-copa/Cabana
3. Co-Design Center for Particle Applications. https://www.exascaleproject.org/research-project/particle-based-applications/
4. Dongarra, J., Luszczek, P.: TOP500, pp. 2055–2057. Springer, US, Boston, MA (2011). https://doi.org/10.1007/978-0-387-09766-4_157
5. Exascale Computing Project. https://www.exascaleproject.org/
6. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
7. Goldstein, H., Poole, C., Safko, J.: Classical Mechanics. Addison Wesley, San Francisco (2002)
8. Gompper, G., Ihle, T., Kroll, D.M., Winkler, R.G.: Multi-Particle Collision Dynamics: A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids. In: Advanced Computer Simulation Approaches for Soft Matter Sciences III, pp. 1–87. Springer, Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-87706-6_1
9. Halver, R., Schulz, S., Sutmann, G.: ALL - A loadbalancing library, C++/Fortran library. https://gitlab.version.fz-juelich.de/SLMS/loadbalancing/-/releases
10. Halver, R., Meinke, J.H., Sutmann, G.: Kokkos implementation of an Ewald coulomb solver and analysis of performance portability. J. Parallel Distrib. Comput. **138**, 48–54 (2020). https://doi.org/10.1016/j.jpdc.2019.12.003
11. Halver, R., Sutmann, G.: Multi-threaded construction of neighbour lists for particle systems in OpenMP. In: Parallel Processing and Applied Mathematics 11th International Conference, PPAM 2015, Krakow, Poland, 6–9 September 2015. Revised Selected Papers, Part II. 11th International Conference on Parallel Processing and Applied Mathematics, Krakow (Poland), 6 Sep 2015–9 Sep 2015 (2015). https://juser.fz-juelich.de/record/279249
12. Huang, C., Winkler, R., Sutmann, G., Gompper, G.: Semidilute polymer solutions at equilibrium and under shear flow. Macromolecules **43**, 10107–10116 (2010)
13. Juwels. https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html
14. Mniszewski, S.M., et al.: Enabling particle applications for exascale computing platforms. Int. J. High Perform. Comput. Appl. **35**(6), 572–597 (2021). https://doi.org/10.1177/10943420211022829
15. Ohno, K., Nitta, T., Nakai, H.: SPH-based fluid simulation on GPU using verlet list and subdivided cell-linked list. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR), pp. 132–138 (2017). https://doi.org/10.1109/CANDAR.2017.104
16. RAJA Performance Portability Layer. https://github.com/LLNL/RAJA
17. Slattery, S., et al.: Cabana: a performance portable library for particle-based simulations. J. Open Source Softw. **7**(72), 4115 (2022). https://doi.org/10.21105/joss.04115
18. Sutmann, G.: MP2C (2022). https://fz-juelich.de/en/ias/jsc/about-us/structure/simulation-and-data-labs/sdl-molecular-systems/mp2c