Computer Physics Communications 241 (2019) 155–177

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers ☆

Pietro Incardona [a,b], Antonio Leo [a], Yaroslav Zaluzhnyi [a], Rajesh Ramaswamy [d,1], Ivo F. Sbalzarini [a,b,c,*]

[a] Chair of Scientific Computing for Systems Biology, Faculty of Computer Science, TU Dresden, Germany
[b] MOSAIC Group, Center for Systems Biology Dresden, Germany
[c] Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
[d] Max Planck Institute for the Physics of Complex Systems, Dresden, Germany

ARTICLE INFO

Article history:
Received 23 April 2018
Received in revised form 14 February 2019
Accepted 13 March 2019
Available online 28 March 2019

Keywords:
Simulation software
Parallel computing
Particle methods
Scalable simulation
Software library
High-performance computing

ABSTRACT

Scalable and efficient numerical simulations continue to gain importance, as computation is firmly established as the third pillar of discovery, alongside theory and experiment. Meanwhile, the performance of computing hardware grows through increasingly heterogeneous parallelism, enabling simulations of ever more complex models. However, efficiently implementing scalable codes on heterogeneous, distributed hardware systems becomes the bottleneck. This bottleneck can be alleviated by intermediate software layers that provide higher-level abstractions closer to the problem domain, reducing development times and allowing computational scientists to focus. Here, we present OpenFPM, an open and scalable framework that provides an abstraction layer for numerical simulations using particles and/or meshes. OpenFPM provides transparent and scalable infrastructure for shared-memory and distributed-memory implementations of particles-only and hybrid particle-mesh simulations of both discrete and continuous models, as well as non-simulation codes. This infrastructure is complemented with frequently used numerical routines, as well as interfaces to third-party libraries. We present the architecture and design of OpenFPM, detail the underlying abstractions, and benchmark the framework in applications ranging from Smoothed-Particle Hydrodynamics (SPH) to Molecular Dynamics (MD), Discrete Element Methods (DEM), Vortex Methods, stencil codes (finite differences), and high-dimensional Monte Carlo sampling (CMA-ES), comparing it to the current state of the art and to existing software frameworks.

Program summary

Program Title: OpenFPM
Program Files doi: http://dx.doi.org/10.17632/4yrp8nbm7c.1
Licensing provisions: GPLv3
Programming language: C++
Nature of problem: Writing numerical simulation programs that use meshes, particles, or any combination of the two typically requires long development times, in particular if the code is to scale efficiently on parallel distributed-memory computers. The long development times incur high financial and project-time costs and often lead to sub-optimal program performance as shortcuts are taken. Yet, a large portion of the functionality is common across programs and could be automated or provided as reusable software components, leading to large savings in project costs and potentially improved software performance.
Solution method: OpenFPM provides a scalable, highly efficient software platform for numerical simulations using meshes, particles, or any combination of the two on parallel computers. It is based on a well-known set of abstract data types and operators that suffice to express any such simulation, regardless of the application domain. OpenFPM provides reusable, tested, and internally parallelized software components that reduce development times and make parallel computing accessible to computational scientists without extensive knowledge in parallel programming.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author at: Chair of Scientific Computing for Systems Biology, Faculty of Computer Science, TU Dresden, Germany.
E-mail address: ivo.sbalzarini@tu-dresden.de (I.F. Sbalzarini).

1 Now at: EMD Serono, Inc., Boston.

https://doi.org/10.1016/j.cpc.2019.03.007

Additional comments including restrictions and unusual features: OpenFPM is a software library based on which users can implement their simulation codes at a fraction of the development cost. All parallelization and memory handling is transparently done by the library. As its main innovation, OpenFPM makes use of C++ Template Meta Programming in order to enable simulations in arbitrary-dimensional spaces, distribution of arbitrary user-defined C++ objects, and compile-time code optimization and targeting for specific hardware platforms. OpenFPM-based simulations can directly output VTK files for visualization of results and HDF5 files for data archiving.

1. Introduction

Computer simulations enable the study of complex models beyond what is possible analytically. The primary goal of a simulation therefore is the scientific or engineering result obtained from the computer experiment. The majority of time, however, is usually spent developing, testing, and optimizing the simulation codes. This is mainly due to the "knowledge gap", stating that the increasingly specialized knowledge required to efficiently use modern supercomputing platforms is found in only a small group of people [1]. The knowledge gap can be reduced by abstracting the algorithmic implementation of the simulation from the computer system platform.

Such abstraction has a long tradition in computational science. It is typically provided as *programming language extensions*, *high-level programming languages*, *software libraries*, or as a *framework* combining programming language extensions and libraries. For high-performance computing (HPC), programming language extensions include OpenACC [2] and OpenMP [3] that provide a directive-based parallel programming model, CUDA [4] and OpenCL [5] for GPGPU and accelerator programming, co-array Fortran (CAF) [6], High-Performance Fortran (HPF) [7], and Unified Parallel C (UPC) [8]. Examples of high-level programming languages for parallel computing include Linda [9], providing a model for coordination and communication between parallel processes, Vectoral [10] for direct vector-processor programming, and Julia [11] designed for high-performance numerical analysis. Examples of software libraries for parallel HPC include implementations of the Message Passing Interface (MPI) [12] standard like OpenMPI [13] and MPICH [14], HPX [15] as a runtime system for parallel and distributed applications, the DASH implementation of the PGAS model providing general-purpose distributed data structures [16], and Charm++ [17] as an example of a framework for distributed parallel programming. Beyond HPC, a number of languages, libraries, and problem-solving environments [18] for simulation exist that focus on sequential processing, including the equation-based simulation language Modelica [19], a Matlab-based compiler [20], and the scientific computing environment FALCON [20]. Such languages and libraries can greatly reduce code-development overhead and render hardware platforms more accessible to a wide user base.

However, the abstractions on which these libraries and languages are based cannot be universal and concise at the same time. Fine abstractions, like those in MPI [21] can be universal (i.e., every parallel algorithm can be implemented with MPI), but may remain hard to use. Coarse abstractions, like the ParMetis library for parallel graph partitioning [22], are easy to use, but provide only limited flexibility (i.e., ParMetis cannot do parallel FFTs). The knowledge gap reduction that is achievable with general-purpose abstractions, like the ones above, is therefore limited by the trade-off between generality and ease of use.

Further reduction of the knowledge gap is possible using abstractions that are specific to a certain application domain. This has successfully been exploited by numerical simulation libraries like OpenFOAM [23] for finite-volume simulations, DUNE [24] and Trilinos [25] for finite-element simulations, DualSPHysics [26] and others [27,28] for Smoothed-Particle Hydrodynamics (SPH) simulations, NAMD [29] and LAMMPS [30] for Molecular Dynamics (MD) and Dissipative Particle Dynamics (DPD) simulations, LibGeoDecomp [31] for cell-based decomposition codes, and AMReX [32] for Adaptive Mesh Refinement (AMR). Examples of domain-specific programming languages for parallel numerical simulations include DOLFIN [33], the programming language of the FEniCS framework for finite-element simulations [34], and Liszt [35], a domain-specific language for mesh stencil codes. Among numerical simulation frameworks, particle methods are particularly appealing from a software-engineering viewpoint, because they can be used to simulate models of all four kinds: discrete, continuous, deterministic, stochastic. For particle methods, mainly three frameworks for distributed parallel computing exist: POOMA [36], FDPS [37], and the Parallel Particle Mesh (PPM) library [38,39] with its domain-specific Parallel Particle Mesh Language (PPML) [40,41]. While these have successfully provided abstractions for rapid development of scalable parallel implementations of particle and particle-mesh methods, two of them seem to be discontinued (POOMA, PPM) and the third (FDPS) seems specifically focused on N-body problems. However, all these languages and libraries have successfully demonstrated the benefits of domain-specific abstraction, and helped close the knowledge gap in scientific high-performance computing.

Based on this past success, we here present OpenFPM, an open-source C++ framework for parallel particles-only and hybrid particle-mesh codes. OpenFPM is intended as a successor to the discontinued PPM Library [38,39] using advanced methods from scientific software engineering, such as template meta-programming (TMP). OpenFPM combines the most general formulation of particle methods with classical mesh-based approaches. A particle is defined as a point in an arbitrary-dimensional space that carries an arbitrary number of arbitrary data structures. A mesh is a regular division of the space into polyhedra (Cartesian at the time of writing) with support for local refinement. OpenFPM uses C++ TMP to transparently handle simulation domains of any dimension, and to allow particles to carry any C++ object as a property, including objects of user-defined classes. OpenFPM also provides its own memory allocators and runtime dynamic load balancing in order to transparently distribute the data and the work among the processors, and to dynamically adapt to changes in local mesh resolution or particle density during a simulation. The functionality of OpenFPM therefore goes beyond that of the PPM Library, and it relaxes PPM's most salient limitations, including the limitation to 2D and 3D simulations and the limitation to primitive types as particle properties. It also extends the capabilities by adding transparent dynamic load balancing, support for accelerator hardware, and automatic memory layout optimization. Using OpenFPM is further facilitated by a complete and up-to-date documentation, as well as a series of tutorial videos and example codes. It is actively supported in the long term with new functionality continuously being added. OpenFPM implements the same abstractions as the PPM Library [1], rendering it easy to understand for experienced PPM users and for novice developers alike.

2. Particle methods

Particle methods provide a unifying algorithmic framework for simulating both discrete and continuous models. When simulating discrete models, particles naturally correspond to the modeled entities, e.g. atoms in molecular dynamics or cars in road traffic simulation. When simulating continuous models or numerically solving differential equations, particles correspond to (Lagrangian) tracer points or mathematical collocation points, e.g., as in SPH [42] or Particle Strength Exchange (PSE) [43–45]. Naturally, particle interactions and dynamics can be either deterministic or stochastic. In all cases, a particle p is described by its position $\boldsymbol{x}_p \in \mathbb{R}^n$ and properties $\boldsymbol{w}_p$ where different elements/properties can be of different data types. Particles can do two things: they can interact with each other, and they can evolve. Interactions are limited to be pairwise. Three-particle and higher-order interactions are emulated as sequences of pairwise interactions. In addition, interactions are restricted to be additive, which ensures that the overall result is independent of the particle indexing order. After all interactions are computed, particles evolve their positions and properties according to pre-defined rules. These rules can be given by the model, such as update rules in a cellular automaton, or they may result from discretization of continuous differential operators, such as in time-integration schemes.

In the most general case, where each particle interacts with every other particle, the positions and properties hence evolve as:

$$\frac{\mathrm{d}\boldsymbol{x}_p}{\mathrm{d}t} = \sum_{q=1}^{N(t)} \boldsymbol{K}(\boldsymbol{x}_p, \boldsymbol{x}_q, \boldsymbol{w}_p, \boldsymbol{w}_q) \tag{1}$$

$$\frac{\mathrm{d}\boldsymbol{w}_p}{\mathrm{d}t} = \sum_{q=1}^{N(t)} \boldsymbol{F}(\boldsymbol{x}_p, \boldsymbol{x}_q, \boldsymbol{w}_p, \boldsymbol{w}_q) \tag{2}$$

with the total number of particles $N(t)$ at time t (continuous or discrete), and the interaction kernels $\boldsymbol{K}$ and $\boldsymbol{F}$ that encapsulate the model being simulated and the numerical method used. An important aspect of particle methods is that the total number of particles can adaptively change during a simulation. It is possible to locally add particles or remove them. This may represent discrete model dynamics, or can be used to implement adaptive-resolution solvers for continuous models.

In the above equations, the evolution of each particle depends on interactions with all N (other) particles. This leads to a nominal computational cost that scales as $O(N^2)$ with more efficient approximation algorithms available [46,47]. In many applications, however, the kernels $\boldsymbol{K}$ and $\boldsymbol{F}$ are local or compact, such that particles only need to interact within a finite radius. Together with efficient data structures, such as Cell Lists [48] or Verlet Lists [49], this reduces the computational cost to $O(N)$ *on average*. In other cases, the kernels can be decomposed into short- and long-range components and only the short-range component is directly evaluated on the particles, whereas the long-range interactions are computed on a uniform Cartesian background mesh [50]. Examples of such hybrid particle-mesh methods include the Ewald method for including electrostatic interactions in MD simulations [51] and remeshed vortex methods for solving the incompressible Navier–Stokes equations [52]. Hybrid particle-mesh methods allow each computational step to be performed in the better-suited formulation. When simulating continuous models, moment-conserving particle-mesh and mesh-particle interpolation are used to translate between the two discretizations [53].

3. The OpenFPM library

OpenFPM is an open-source software library to implement scalable particle and hybrid particle-mesh simulations on shared-memory and distributed-memory parallel computer systems. OpenFPM is written in C++ and provides the same abstractions as the PPM Library [38,39]. These abstractions have been designed to be as coarse-grained as possible while still separating computation from communication [1]. OpenFPM provides a scalable infrastructure to implement custom particle and particle-mesh codes. It provides methods for domain decomposition, dynamic load balancing, communication abstractions, transparent handling of cross-processor interactions, checkpoint/restart facilities, asymmetric and symmetric (i.e., "action–reaction law") particle interactions, as well as iterators for particles and meshes. The distributed data structures of OpenFPM are implemented atop efficient single-core data structures, which in turn rely on compile-time memory layout and runtime memory allocators using RTTI (run-time type information) virtual functions (see Fig. 1). This infrastructure is complemented by a set of frequently used numerical solvers, including interfaces to the solvers in PetSc [54] and in Eigen [55].

OpenFPM provides flexibility through templated parametric data structures. A parameter in a data structure can, e.g., define the dimensionality of the space or the floating-point precision used. It can also add or modify the code of the data structure implementation, e.g., in order to change the internal memory layout. This enables hardware-targeting of data structures at compile time. In addition, OpenFPM enforces strong unit testing for strict code quality assurance. Its design follows modern programming techniques like Template Meta Programming (TMP) and the familiar GNU configure/make installation process. OpenFPM's release cycles follow the software engineering method of Continuous Integration and include static code analysis, automated performance testing, memory coherence and memory leak testing, test coverage analysis, and tested installation support for multiple operating systems and compilers. The full documentation of OpenFPM is automatically updated after every change to the source code. Updates are deployed online as source code tested for Windows (Cygwin), macOS, and Linux, as binaries for Linux, as virtual-machine disk images with completely pre-installed OpenFPM, and as Docker containers for virtualized environments. This ensures code quality, simplifies installation, and renders the framework fully open, reproducible, and accessible.

3.1. OpenFPM abstractions

OpenFPM provides transparently distributed data structures for particles and meshes in n-dimensional computational domains. Particle sets in OpenFPM are objects storing the positions $\boldsymbol{x}_p$ and properties $\boldsymbol{w}_p$ of all particles p in the set. Multiple particle sets can be used concurrently. At the time of writing, meshes are implemented as regular Cartesian grids that do not require storing the position of each mesh node. This defines the data abstractions of OpenFPM: particle sets and meshes.

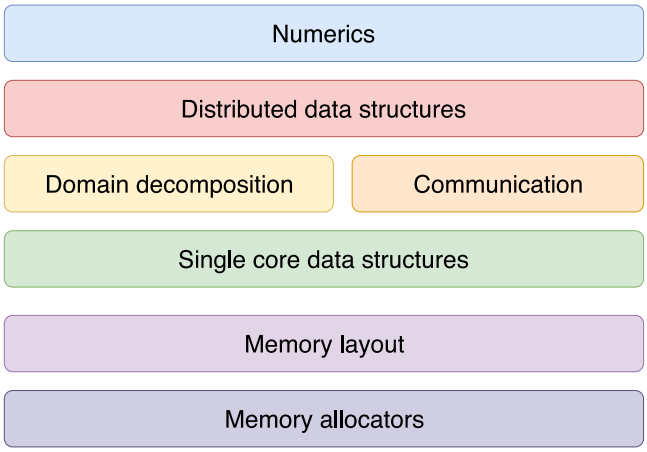


Fig. 1. Diagram of the OpenFPM software stack. The top-level modules provide scalable numerical algorithms implemented using the transparently distributed data structure of the next-lower level. The distributed data structures in turn are implemented based on efficient single-core data structures through a domain-decomposition and inter-processor-communication layer. The memory layout of the single-core data structures is parametrically decided at compile time and managed at runtime by the lowermost layers.

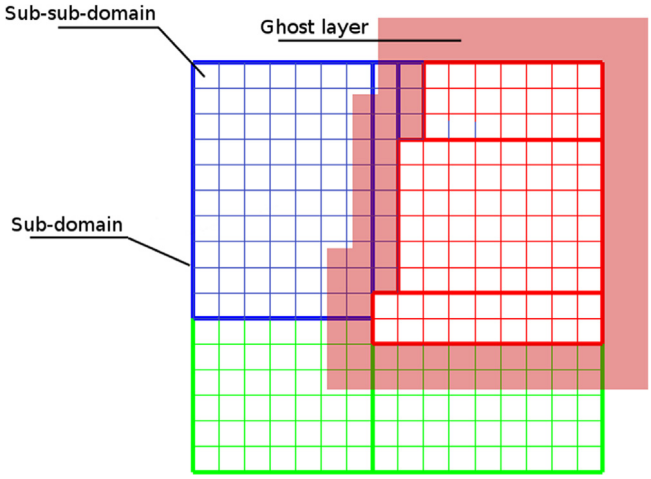


Fig. 2. Domain decomposition in OpenFPM. The computational domain is decomposed into Cartesian sub–sub-domains (small squares) that are subsequently assigned to processors (colors). After assignment, cuboidal blocks of sub–sub-domains are merged to form the larger sub-domains (bold lines). One processor may have more than one subdomain. In order to locally provide all data required for the computations, sub-domain borders at processor boundaries are extended by a ghost layer (shaded area, shown exemplarily for the red processor) for particles. For mesh nodes, all sub-domain boundaries have ghost layers, also within a processor, in order to allow mesh solvers to independently iterate over sub-domains. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

These data abstractions are distributed across multiple computers or memory address spaces in a way that is transparent to the user. This is done by decomposing the n-dimensional simulation domain into sub-domains that are then assigned processors (Fig. 2). Each processor only stores the particles and mesh nodes inside its subdomains. Each processor also only computes the interactions and values of its particles and mesh nodes, hence parallelizing both data and work. In order for all computations to be local, sub-domains are extended by a *ghost layer* (halo layer) around inter-processor boundaries (Fig. 2). The width of the ghost layers is given by the particle interaction radius or the radius of the mesh stencil and can spatially vary.

Inter-processor communication in OpenFPM is done by communication abstractions called *Mappings*. Mappings only communicate, but do not compute. Different types of mappings are provided to distribute particle and/or mesh data according to a certain domain decomposition and to manage ghost layers. Once the ghost layers are populated, all computation can be done locally, hence cleanly separating communication and computation. This makes the communication overhead explicit to the user and enables optimization and predictive performance modeling.

3.2. Domain decomposition

The domain decomposition process in OpenFPM is divided into three steps: *decomposition*, *distribution*, and *sub-domain creation*, as summarized in Algorithm 1. The decomposition step divides the computational domain into Cartesian sub–sub-domains (small squares in Fig. 2). The number of sub–sub-domains generated is at least as large as the number of processors, but typically much larger. The sub–sub-domain size is either specified by the user program or chosen automatically by OpenFPM. In the latter case, OpenFPM chooses the largest possible size that guarantees at least 64 sub–sub-domains per processor.

Algorithm 1 Domain decomposition.

```
1: procedure DomainDec
2:     Decomposition:
3:         Divide the bounding box of the computational domain into regular Cartesian sub–sub-domains
4:     Distribution:
5:         Create a graph where each sub–sub-domain is a vertex and edges indicate potential communication if two
6:             sub–sub-domains are assigned to different processors
7:         Partition the graph using ParMetis [22] in order to define processors boundaries
8:     Sub-domain creation:
9:         Merge the sub–sub-domains locally on each processor to form larger sub-domains with less intra-processor mesh ghost
10:            layers
```

In the second step, the sub–sub-domains are assigned to processors (colors in Fig. 2). Because their number is larger than the number of processors, there is no trivial assignment. Instead, the additional degrees of freedom can be used to improve load balance and to reduce communication overhead. An optimal mapping would ensure that each processor receives the same amount of computational work (in terms of wall-clock time), while the total volume of inter-processor communication is minimized. This problem can be modeled as a graph-partitioning problem where each sub–sub-domain is a vertex of the graph, and an undirected edge between two vertices indicates an exchange of data between the respective sub–sub-domains through their ghost layers. In order to account for varying particle density and different sub–sub-domain sizes, we assign to each vertex a computational cost c_i that is proportional to the total amount of compute operations required in sub–sub-domain i. We also weight the edges $e_{i,j}$ between sub–sub-domains i and j proportional to the amount of data that needs to be exchanged between them. In an optimal assignment, the sum of the vertex weights is the same for all processors, while the sum of the weights of all edges crossing a processor boundary is minimal. While the optimal solution cannot efficiently be computed, several publicly available libraries compute approximate sub-optimal solutions to this problem. These libraries include Scotch [56], ParMetis [22], and Zoltan [57]. OpenFPM uses ParMetis because of the efficiency of the implemented algorithms. The approximate solution to the graph-partitioning problem computed by ParMetis then defines the assignment of sub–sub-domains to processors. Alternatively, the user can choose to distribute sub–sub-domains along a Hilbert-type space-filling curve.

In the third step, the sub–sub-domains assigned to a processor are merged into larger sub-domains (bold lines in Fig. 2) wherever possible, in order to reduce the total ghost-layer volume. Sub–sub-domains on the same processor do not require a ghost layer between them. Nevertheless, such ghost layers would be created for meshes (not for particles) because mesh data structures are allocated sub-domain-wise in OpenFPM. The merging step is therefore important for meshes and reduces the total mesh ghost-layer volume. The goal of sub-domain creation therefore is to merge sub–sub-domains such that the minimum number of sub-domains with smallest surface-to-volume ratio is created on each processor.

The sub-domain creation algorithm runs in parallel on each processor and starts from the first (by indexing order) sub–sub-domain on that processor, which is used as a seed. From there, it extends the sub-domain boundaries uniformly in all directions. For example, in 2D, the box is enlarged by shifting the border by one sub–sub-domain in directions $+$X, $+$Y, $-$X, $-$Y. This procedure is iterated until the sub-domain border reaches an inter-processor boundary, or it is not possible to merge any more sub–sub-domains in any direction. A sub-domain is then created from all merged sub–sub-domains. The process then chooses the next (by indexing order) unassigned sub–sub-domain at the border of the just-created sub-domain and repeats until all sub–sub-domains have been assigned. Despite not finding the optimal solution, i.e., the one with the smallest number of minimum-surface sub-domains, this greedy heuristic is perfectly parallel and finds a sub-optimal solution in linear time (linear in the number of sub–sub-domains per processor).

Out of the total time taken for domain decomposition, 94% to 99% is consumed by ParMetis in the *distribution* step. This fraction was stable across all of our benchmarks. The *decomposition* step always took less than 1% of the total time and the *sub-domain creation* step less than 5% of the total time. The overall runtime performance of the OpenFPM domain decomposition is therefore dominated by the time required by ParMetis, which is why we do not further benchmark it here.

3.3. Parametric data structures

OpenFPM uses parametric data structures and C++ TMP in order to adaptively generate optimized implementation code at compile-time. This is done for three reasons: First, it enables the internal implementation of data structures to adapt to the hardware platform for which they are being compiled, optimizing, e.g., the memory layout. Second, it avoids slow runtime polymorphism since dependencies can be resolved at compile-time and the corresponding code injected, leading to significant speed-ups. Third, it renders the interface and semantics of the data structures independent of simulation parameters, such as the dimensionality of the computational domain or the floating-point precision used. TMP is used for compile-time code generation with code-generation logic extending beyond simple compile-time substitutions of template parameters. TMP uses C++ template meta-functions to compute implementation code at compile time. Therefore, OpenFPM does for example not have a struct/class "particle", but simply specifies the list of particle properties as variadic templates from which all code (e.g., for the actual particle class, for serialization, communication, and I/O) is automatically generated at compile time without requiring further user code. This renders OpenFPM data structures fully generic, e.g., to different target hardware platforms, with minimal or no maintenance code.

An example of such a parametric data structure is shown in Fig. 3 for a particle set stored as a distributed vector (`vector_dist`). The parameters of the data structure are specified in angle brackets as a variadic C++ template. The example in Fig. 3a defines a distributed vector in 2D (first parameter is 2) with single-precision floating point numbers (the second parameter is "float"). The third parameter is the list (`aggregate`) of particle properties. In the example here, each particle stores a scalar float, an array of 3 floats, a double-precision `Point` object in a 3D space, and an OpenFPM `vector` object (equivalent to `std::vector`) of floats. Parametric data structures can

(a)

```
vector_dist<2,float, aggregate<float,
                               float[3],
                               Point<2,float >,
                               openfpm::vector<float>>
```

(b)

```
vector_dist<2,float, aggregate<float,
                               float[3],
                               Point<2,float >,
                               openfpm::vector<float>>,
                memory_traits_lin,
                CartDecomposition<2,float>,HeapMemory,
                                 ParMetisDistribution<2,float>
                HeapMemory
                >
```

Fig. 3. Data structures in OpenFPM are parametric, which allows to target them to different space dimensionality, data types, and hardware platforms at compile time. An example is shown for a particle set stored as a distributed vector. This includes mandatory parameters (a), such as the space dimension and the data types of the particle properties, but may also include optional parameters (b), such as the internal memory layout or the type of domain decomposition used for this data structure. This renders all data structures generic and independent of their actual implementation, with optimized code automatically generated by the compiler.

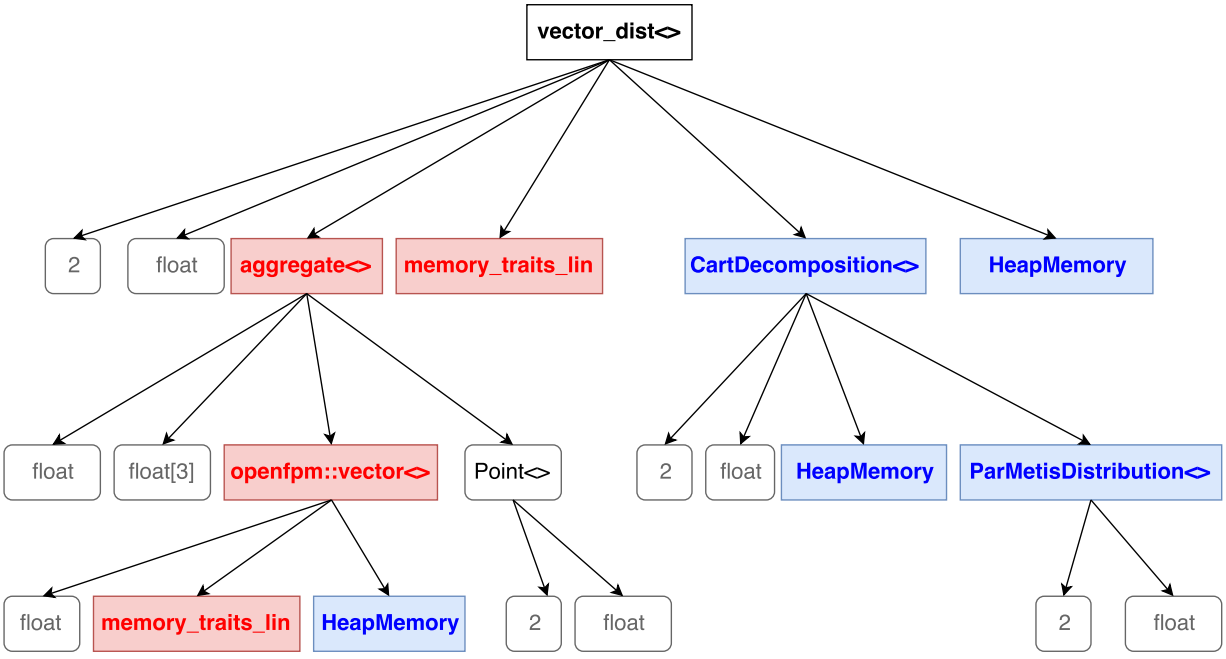


Fig. 4. Tree expansion of the parametric data structure from Fig. 3b at compile-time. Introspection of the tree nodes allows the compiler to automatically generate implementations of all nodes and inline the code. Child nodes shown in blue are simply generated and inlined, whereas child nodes shown in red alter the implementation of their parents after self-introspection (non-local code injection). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

be arbitrarily nested and contain further parametric data structures within, as for example illustrated for the parametric `Point` object here.

There are additional, optional parameters that may be specified. In Fig. 3b, some optional parameters of the `vector_dist` data structure are shown in addition to the mandatory parameters already present in Fig. 3a. This includes parameters that define the memory layout of the data structure (here `memory_traits_lin` for linear contiguous memory layout), the type of domain decomposition used to distribute this data structure (here decomposition into Cartesian blocks in 2D with float precision), the type of load balancer to be used on this data structure (here the distributed graph partitioning provided by the ParMetis library [22] in 2D), and the type of memory where the data inside this data structure are going to be allocated (here `HeapMemory`; this could also be stack memory or GPU device memory, for example). This defines fully parametric data structures, both with respect to the application and to the target machine hardware.

At compile-time, the actual implementations of the classes for these data structures and of the methods required for serializing, communicating, accessing, and iterating over these data structures are automatically generated by the TMP engine of the C+ compiler. Therefore, the compiler expands the template parameters of a data structure into a tree, as shown in Fig. 4 for the example from Fig. 3b. Each node in this tree contains the rules (in the form of C++ template meta-functions) that specify how to generate the internal source code and call the child nodes. There are two types of child nodes: those that are simply called by the parent (shown in blue in Fig. 4), and those that may alter the implementation of their parent after self-introspection (shown in red in Fig. 4). The blue child nodes are inlined, such that all compiler optimizations apply. A parent node that has one or more red child nodes can alter its own implementation after introspection of the respective child nodes. This could, e.g., involve analyzing whether a child node implements a certain method or defines specific attributes that can be used to transform the parent code to automatically adjust to the child node's interface. The compiler processes the tree recursively until the internal implementations of all nodes have been generated.

As an example, consider data stored in an array of aggregates. The parent node `vector_dist` analyzes at compile-time whether any data type in the aggregate is a native object, like `int`, `double`, `float`, `float[3]`, etc. If this is not the case, the parent node checks at compile-time whether any C++ class contained in the aggregate defines a function `pack()` to serialize its own data or a function `hasPointers()` to indicate that the class points to data in another object. Depending on the outcome of these checks, different code

for the parent node is generated. If at least one of the classes in the aggregate has `pack()`, the parent node generates code to call this pack function for the specific variables contained in the aggregate. If none of the aggregate classes defines a `pack()` function, and no pointers are present, the code is optimized to perform a single memory copy of the full array of aggregates. In case there are pointers, but no `pack()` function is defined, code to display an error message at run-time is generated. If all of the classes in the aggregate define neither a `pack()` function nor `hasPointers()`, simple memory-copy code is generated, but a warning message is displayed to make the user aware that this is not guaranteed to be safe.

Compile-time code injection, analysis, and introspection rely on the template engine of the C++ compiler. TMP is used for conditional code injection/inlining. While injected/inlined code is implementation-dependent, the TMP meta-code required for analysis is independent of the specific data structure implementation and is encapsulated into the modular and re-usable parametric structures of OpenFPM.

Parametric data structures are also used by OpenFPM to manage memory and to control memory allocation patterns. Parameters are, e.g., available to control whether code is generated to allocate the contents of a data structure on the heap (see Fig. 3b), on device memory such as on GPUs, or in pre-allocated external memory (e.g., transparent memory hoards). The code for the actual implementation and memory layout of the data structure is automatically injected at compile-time. The same is also used to control memory layout (e.g., `memory_traits_lin` in Fig. 3b) or to force a data structure to have a certain implementation (i.e., array of structs vs. struct of arrays).

In conclusion, the internal implementations of all OpenFPM data structures are automatically and adaptively generated at compile-time through a multi-level and modular source-code generation process relying on TMP. This opens the possibility to aggressively optimize the code, increase its flexibility and versatility, and to help the compiler exploit compile-time information to produce more scalable code. In addition, it also renders the code more compact, significantly reducing development and maintenance overheads.

3.4. Mappings

Mappings are OpenFPM communication abstractions [1]. They are provided in order to transparently distribute data according to a previously determined domain decomposition and to manage ghost layers. Each OpenFPM data structure can have its own domain decomposition and provides a method `map()` to distribute data according to this decomposition. The `map()` method is typically used when particles have migrated across processor boundaries (local map), after distributed reading of data from a file (global map), or after the load balancer has altered the domain decomposition (global map). In a *local mapping*, processors only communicate with their neighbors (in the given domain decomposition and application connectivity). In this case, the `map()` function internally uses non-blocking point-to-point communications. This avoids the need for an explicit communication schedule, e.g. using graph coloring, as required in synchronous protocols [38]. For the *global mappings*, OpenFPM uses concepts from dynamic sparse data exchange (DSDE), specifically the non-blocking consensus (NBX) protocol [58], in order to limit non-blocking communication to only those links that are actually needed, i.e., where there actually is data to communicate. Since this depends on the simulation state, and may differ from time step to time step, it can only be decided at runtime, hence the use of NBX.

Each OpenFPM data structure also provides methods to manage ghost layers, called *ghost mappings*. OpenFPM provides two types of ghost mappings: `ghost_get()` and `ghost_put()`. Ghost layers are populated with copies of the particles and/or mesh nodes from the overlapping regions of the neighboring processors using the `ghost_get()` mapping. Calling `ghost_get()` automatically populates the ghost layers of all processors without the user needing to know where the data came from, or how it has been communicated. This again uses an optimized internal communication schedule, as described above. The second ghost-related mapping is `ghost_put()`. This mapping is used to send data from the ghost layer back to the source processor that stores the corresponding particle or mesh node. This is for example required when evaluating symmetric particle interactions that also alter the values of a ghost. In this case, the ghost contributions need to be sent back to the source processor where they are added to the local interaction results. The `ghost_put()` mapping provides three different ways of merging the sent-back ghost contributions with the results on the source processor: sum the values on the source processor, replace the value on the source processor with the maximum of all incoming contributions, or merge all contributions into a list. The first is typically used to send symmetric interaction results back for aggregation. The second could be used for collision detection, while the third is typically the case when managing contact lists in discrete-particle simulations. In addition to these three pre-defined `ghost_put()` merging operations, the user may also implement own operators as C++ template functors.

All mappings are internally implemented using non-blocking MPI communication. In all mappings it is also possible to only map a subset of the properties of the affected particles or mesh nodes. This is done by specifying the list of properties that are to be mapped as an optional template parameter to the mapping methods, causing the compiler to automatically implement versions of the mappings optimized to the specified subsets.

3.5. Dynamic load balancing

Domain decomposition leads to an initially load-balanced and communication-minimized distribution of work and data across processors. During the course of a simulation, however, particle movement, dynamic resolution refinement, and particle evolution may lead to progressively worse load balance and increasing communication overhead. OpenFPM therefore provides runtime load re-balancing. While this can be done in a number of ways, including using multi-grid approaches [59], space-filling curves [60], and adaptive distributed resampling [61], we choose for simplicity to use the same method as in the initial domain decomposition, where the task is formulated as a graph-partitioning problem that is approximately solved using ParMetis [22]. Load re-balancing is done on the level of sub–sub-domains and is followed by another sub-domain creation step, but avoids *de novo* domain decomposition. In addition, the current decomposition graph is used as a soft constraint for the new graph decomposition in order to minimize load migration. An additional migration cost m_i is assigned to each sub–sub-domain, corresponding to the amount of data that would need to be communicated if that sub–sub-domain changed processor. This cost term is included into the graph-decomposition problem in order to obtain re-balanced processor assignments that trade off the migration cost of transferring sub–sub-domains between processors against the gain in load balance. The data-migration cost is linearly discounted over the number of simulation time steps since the last re-balancing. The time-points when re-balancing is beneficial, i.e., when the cost of the previous re-balancing is amortized by the expected gain in load balance, are automatically determined using the Stop-At-Rise (SAR) heuristic [62] or can be specified by the user program.

Table 1
Memory bandwidth per core on the benchmark machine when using different numbers of cores all on the same processor socket.

#Cores/socket:	1	2	4	8	12
Memory GB/s per core:	14.7	10.8	10.0	8.3	5.0

3.6. Iterators

Because of the non-trivial distribution of data across processors, looping over particles or mesh nodes in a cache-friendly manner requires additional information. In order to hide this complexity from the user, OpenFPM provides transparent iterators for particles and meshes.

On a mesh, iterators are multi-dimensional (i.e., multi-indices) and follow the natural topology of the mesh. On particles, iterators are one-dimensional. Since the indexing order of the particles is inconsequential for the simulation result (see Section 2), the user can select iterators that automatically create cache-friendly loops, which may re-sort or re-order the particles in a more memory-efficient way. For this, OpenFPM uses a space-filling Hilbert curve along the Cartesian cells of the particle cell-list on each processor. The particles are then linearly arranged along the Hilbert curve.

Separate iterators are provided for interior particles and interior mesh nodes and for ghost particles and ghost mesh nodes, which allows, e.g., overlapping communication with computation by iterating over interior elements while the ghosts are mapped in the background (all OpenFPM mappings use non-blocking MPI calls, see Section 3.4), before iterating also over the ghosts. Examples of iterators for both particles and meshes are shown in the code examples in Section 4.

3.7. File I/O, check-point restart

The distributed data structures provided by OpenFPM can be saved to file using the portable binary HDF5 file format [63]. This is done by calling the OpenFPM method `save()`, irrespective of how the data are distributed across processors. Files written in this way can be read back using the OpenFPM method `load()`. Since HDF5 is a parallel file format, reading can be done on any number of processors, independent from the number of processors used to save the file, and the domain decompositions at the time of saving and loading may be different. OpenFPM HDF5 files can therefore also directly be used to restart a simulation from a previously saved state (check-point restart), including restarts on different numbers of processors.

The HDF5 format of OpenFPM internally serializes all data structures, which allows arbitrary C++ objects and nested parametric data structures to be transparently supported. For this, OpenFPM provides an encapsulated serialization/de-serialization sub-system that is used by each processor locally in order to serialize the local pieces of a distributed data structure into a so-called "chunk". The chunks from all processors are then saved contiguously into a parallel HDF5 file. Meta-data is automatically added by the serializer to allow de-serialization on other numbers of processors and on other domain decompositions.

When reading an OpenFPM HDF5 file, chunks are read in parallel by individual processors and, after distributed de-serialization, mapped onto the new domain decomposition. This map-after-read strategy is preferred over partitioned reads, as it causes data to be read/written in large contiguous blocks rather than in multiple smaller random reads, producing less load in the I/O sub-system.

In addition to HDF5 output, distributed data can also be saved to files in VTK format [64] using the OpenFPM method `write()`. This enables direct visualization of the particle and mesh data, e.g., in ParaView [65], an open-source scientific data visualization software natively reading VTK files. While OpenFPM HDF5 files cannot directly be visualized in ParaView, they can easily be converted to VTK files using `load()` to reload the HDF5 file into distributed data structures followed by `write()` to export to VTK. VTK output and ParaView were used to generate all visualizations in Section 4 of this paper.

4. Results

We demonstrate the use, performance, and scalability of OpenFPM in a number of test cases from different application domains. We compare with a respective state-of-the-art code in each application domain, demonstrating that a generic framework like OpenFPM can reach and, in some cases, even exceed the performance of application-specific codes that matured over many years. All tests are performed on the computer cluster of the *Centre for Information Services and High Performance Computing (ZiH)* of TU Dresden. Each node of the cluster is equipped with two 2.5 GHz 12-core Intel Xeon E5-2680v3 CPUs (total 24 cores per node) sharing 64 GB of RAM. The peak bandwidth of the memory sub-system is 68 GB/s per socket. The cluster interconnect is an InfiniBand network with 40 Gb/s bandwidth. The machine is running RedHat Enterprise Linux (RHEL) Server release 6.9 (Santiago) as operating system. For all tests, OpenFPM was compiled using GCC g++ version 7.1.0 and linked against the OpenMPI 3.0.0 [13] implementation of the MPI standard version 3 [12]. We use process-socket-affinity binding in all benchmarks in order to provide reproducible measurements.

On the benchmark machine, each processor (socket) has its own, independent memory bus. The cores within each processor, however, share the memory bandwidth. The results of a concurrent memory-read benchmark are shown in Table 1. The per-core memory bandwidth reduces from 14.7 GB/s when using only 1 core to 5.0 GB/s when using all 12 cores of a processor in parallel. This synthetic benchmark shows that when using 8 cores per socket, the memory bus of the machine saturates at about 97.6% of its theoretical peak bandwidth.

4.1. Molecular dynamics

We first consider a classical Molecular Dynamics (MD) application simulating a Lennard-Jones fluid. In this simulation, particles represent atoms that interact according to the pairwise Lennard-Jones potential:

$$V_{\mathrm{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{3}$$

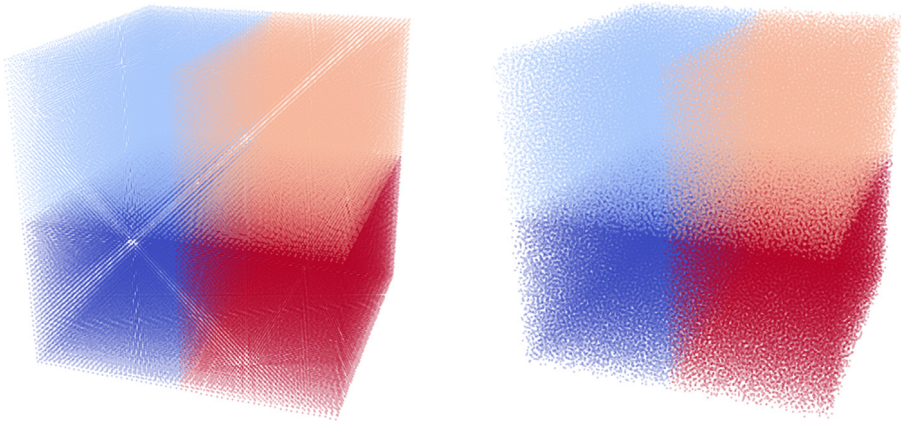


Fig. 5. Visualization of the particle configurations at the start of the simulation (left) and after 5000 time steps (right) for the Lennard-Jones molecular dynamics test case. The system is thermally equilibrated after 1000 time steps. In this example, a decomposition into four sub-domains is used, indicated by different colors. Each particle is plotted as a dot with the color of the respective sub-domain. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

as a function of the distance r between the two interacting particles. The parameters σ and ϵ define the zero-crossing and the well depth of the potential, respectively. We implement the simulation in OpenFPM using OpenFPM's implementation of Verlet lists [49] for the particles to efficiently find their interaction partners. We exploit symmetry in the interactions, i.e., we compute every interaction pair only once. This also requires changing the values of ghost particles, for which we use OpenFPM's `ghost_put()` mapping (see Section 3.4).

We compare the OpenFPM-based implementation with LAMMPS [30], a well-established and highly optimized parallel MD code. We start the simulation with 216,000 particles initialized on a regular Cartesian 60^3 mesh. This initial particle configuration is shown in the left panel of Fig. 5 for a Cartesian domain decomposition with four sub-domains indicated by different colors. We first equilibrate the system and then simulate its time dynamics using the symplectic velocity-Verlet time-stepping scheme [49] with step size $\delta t = 0.01$. For the Lennard-Jones potential, we use $\sigma = 0.1$ and $\epsilon = 1$. Periodic boundary conditions are imposed in all three coordinate directions. We run the simulation for 5000 time steps (equilibration finishes after 1000 time steps). The final configuration of particles is shown in the right panel of Fig. 5. The time courses of the kinetic, potential, and total energies of the system as computed by LAMMPS and OpenFPM are identical (data not shown) and the total energy is conserved to machine precision, hence validating the simulation.

The OpenFPM-based simulation can be implemented in less than 40 lines of C++ code (not counting comments), as shown in Listing 4.1. In the example listing, lines 10–15 define the pairwise Lennard-Jones interaction between the particles. The main program starts by initializing OpenFPM in line 19 and defining the size of the simulation domain in line 28 (here: the unit cube), the boundary conditions in line 29 (here: periodic in all three directions), and the size of the ghost layers in line 30 (here: given by the interaction cut-off radius `r_cut`). The particle interaction object is instantiated in line 33 based on the definition of the interaction from lines 10 to 15. Line 36 declares a distributed vector to store the particles. The particle position consists of 3 doubles (first two template parameters), and the particle properties are an aggregate containing two 3D points/vectors of doubles (here: one for the velocity and one for the force). Lines 6 and 7 define that the first property (i.e, property 0) is the velocity and the second (i.e, property 1) is the force. The particles are then initialized on a regular Cartesian lattice in line 37, where the number of mesh points (`sz`) has been defined in line 27 (here: 60 in each direction). Lines 41–43 define convenient aliases for the particle position, velocity, and force that can be used to simplify notation in expressions like the one in line 58. Line 50 creates the symmetric particle cell lists for fast neighbor access and uses them in line 51 to compute the initial forces using symmetric particle interactions (omitting the `_sym` specifier would compute the interactions without exploiting symmetry). The `_in` specifier tells OpenFPM to only compute interactions for internal (i.e., non-ghost) particles. Lines 54–73 contain the simulation time loop using the two-step velocity-Verlet time integrator (lines 58, 59, 72). Lines 63 and 64 contain communication operations. Line 63 performs a mapping to migrate particles that have moved across processor boundaries, and line 64 performs a ghost-get mapping for only the particle positions (empty properties list <>). Line 76 finalizes the OpenFPM Library at the end of the program.

Table 2 compares the performance of the OpenFPM-based implementation using Verlet lists with LAMMPS for a strong scaling, i.e., for distributing the fixed number of 216,000 particles across an increasing number of processors. The absolute wall-clock time per time step is below 1 second even on a single core. On 1536 cores (using all 12 cores on 128 nodes), a simulation time step is completed in 0.5 ms. Despite the fact that OpenFPM is a general-purpose particle-mesh library and is not limited to MD, its performance is almost as good as that of the highly optimized LAMMPS.

In addition, we show in Table 3 how the runtime of the OpenFPM client changes when using different distributions of the same total number of cores across different numbers of shared-memory sockets. Taking, e.g., the 8-core case from Table 2, corresponding to the 4×2 case in Table 3. The runtime does not significantly increase when going from 4 cores per socket to 8 cores per socket, indicating that memory bandwidth is not the most important bottleneck for this code. Indeed, the size of the problem suggests that it could fit into L3 cache, making cache latency the bottleneck.

4.2. Smoothed-particle hydrodynamics

Smoothed-Particle Hydrodynamics (SPH) is a widely used method for simulating continuous models of fluid dynamics. Due to its simplicity and flexibility in modeling complex fluid properties and free fluid surfaces, it is preferentially used to model multi-phase flows and fluid–structure interaction [66,67], as well as for multi-resolution simulations [28].

Listing 4.1: C++ code for Lennard-Jones molecular dynamics using OpenFPM

```
///// define parameters
double sigma12, sigma6, epsilon = 1.0, sigma = 0.1;   // parameters of the potential
double dt = 0.0005, r_cut = 3.0*sigma;                // parameters of the simulation
double r_cut2;

constexpr int velocity_prop = 0;  // velocity is the first particle property
constexpr int force_prop = 1;     // force is the second particle property

///// Define Lennard-Jones interaction
DEFINE_INTERACTION_3D(ln_force)
    Point<3,double> r = xp - xq;
    double rn = norm2(r);
    if (rn > r_cut2) return 0.0;
    return 24.0*epsilon*(2.0*sigma12/(rn*rn*rn*rn*rn*rn*rn)-sigma6/(rn*rn*rn*rn))*r;
END_INTERACTION

int main(int argc, char* argv[]) {
    ///// Initialize OpenFPM
    openfpm_init(&argc,&argv);

    ///// Initialize constants
    sigma6 = pow(sigma,6), sigma12 = pow(sigma,12);
    r_cut2 = r_cut*r_cut;

    ///// Define initialization grid, simulation box, boundary conditions,
    ///// and ghost layer
    size_t sz[3] = {60,60,60};
    Box<3,float> box({0.0,0.0,0.0},{1.0,1.0,1.0});
    size_t bc[3]={PERIODIC,PERIODIC,PERIODIC};
    Ghost<3,float> ghost(r_cut);

    ///// Create Lennard-Jones potential object
    ln_force lennard_jones;

    ////// Define particles and initialize them on a grid
    vector_dist<3,double,aggregate<Point<3,double>,Point<3,double>>> particles(0,box,bc,ghost);
    Init_grid(sz,particles);

    ///// Define aliases for the particle force, velocity, and position
    ///// to simplify notation
    auto force = getV<force_prop>(particles);
    auto velocity = getV<velocity_prop>(particles);
    auto position = getV<PROP_POS>(particles);  // PROP_POS is defined in OpenFPM

    ///// initialize all particle velocities to zero
    velocity = 0;

    ///// Generate the cell lists and compute the initial forces using the Lennard-Jones
    ///// potential evaluated with exploiting symmetry
    auto NN = particles.getCellListSym(r_cut);
    force = applyKernel_in_sym(particles,NN,lennard_jones);

    ///// Time loop
    for (size_t i = 0; i < 10000 ; i++) {
        ///// 1st step of velocity-Verlet time integration
        ///// v(t + 1/2*dt) = v(t) + 1/2*force(t)*dt
        ///// x(t + dt) = x(t) + v(t + 1/2*dt)
        velocity = velocity + 0.5*dt*force;
        position = position + velocity*dt;

        ///// communicate particles that have crossed processor boundaries and
        ///// update the ghost layers for all properties (empty props list)
        particles.map();
        particles.ghost_get<>();

        ///// Update the cell lists after mapping the particles
        particles.updateCellListSym(NN);
        ///// Calculate the forces at t + dt
        force = applyKernel_in_sym(particles,NN,lennard_jones);

        ///// 2nd step of velocity-Verlet time integration
        ///// v(t+dt) = v(t + 1/2*dt) + 1/2*force(t+dt)*dt
        velocity = velocity + 0.5*dt*force;
    }

    ///// Finalize OpenFPM and deallocate all memory
    openfpm_finalize();
}
```

Table 2
Molecular dynamics benchmark results. We report wall-clock execution times (mean ± standard deviation over 10 independent runs) and parallel efficiencies of the OpenFPM client compared with LAMMPS [30] for a strong scaling from 1 to 1536 processor cores (12 cores per node) simulating 216,000 Lennard-Jones particles in the unit cube over 5000 time steps (see Fig. 5).

#cores	OpenFPM (seconds)	LAMMPS (seconds)	OpenFPM (Efficiency)	LAMMPS (Efficiency)
1	1010.69 ± 1.58	976.10 ± 3.30	100.0%	100.0%
4	262.55 ± 0.80	257.00 ± 6.48	96.2%	94.9%
8	143.81 ± 0.26	137.10 ± 0.31	87.8%	89.0%
16	77.10 ± 0.40	73.00 ± 0.46	81.9%	83.6%
24	52.70 ± 0.27	49.89 ± 0.17	79.9%	81.5%
48	29.70 ± 0.12	28.98 ± 0.22	70.9%	70.2%
96	15.16 ± 0.11	15.64 ± 0.60	69.4%	65.0%
192	8.07 ± 0.16	8.22 ± 0.32	65.2%	61.8%
384	4.73 ± 0.16	4.66 ± 0.17	55.6%	54.5%
768	3.15 ± 0.09	3.37 ± 1.10	41.8%	37.7%
1536	2.20 ± 0.24	1.93 ± 0.77	29.9%	32.9%

Table 3
Average runtime of the OpenFPM Lennard-Jones client using different numbers of cores per socket for the same total of 8 cores.

#cores per socket × #sockets	OpenFPM (seconds)
8 × 1	147.4
4 × 2	143.8
2 × 4	133.5
1 × 8	128.0

We use OpenFPM to implement a weakly compressible SPH solver for the Navier–Stokes equations, where each particle p has a velocity $\boldsymbol{v}_p$, a pressure P_p, and a density ρ_p. The evolution of these particle properties is governed by [42]:

$$\frac{d\boldsymbol{v}_p}{dt} = -\sum_{q\in\mathcal{N}(p)} m_q \left(\frac{P_p + P_q}{\rho_p \rho_q} + \Pi_{pq}\right) \nabla W(\boldsymbol{x}_q - \boldsymbol{x}_p) + \boldsymbol{g} \tag{4}$$

$$\frac{d\rho_p}{dt} = \sum_{q\in\mathcal{N}(p)} m_q \boldsymbol{v}_{pq} \cdot \nabla W(\boldsymbol{x}_q - \boldsymbol{x}_p) \tag{5}$$

$$P_p = b\left[\left(\frac{\rho_p}{\rho_0}\right)^{\gamma} - 1\right] \tag{6}$$

$$b = \frac{1}{\gamma} c_{\text{sound}}^2 |\boldsymbol{g}| h_{\text{swl}} \rho_0\,, \tag{7}$$

where m_q is the mass of particle q, h_{swl} is the maximum height of the fluid, $\gamma = 7$, and $c_{\text{sound}} = 20\,\text{m/s}$ [42]. Here, $\mathcal{N}(p)$ is the set of all particles within a cutoff radius of $2\sqrt{3}h$ from p, where h is the distance between nearest neighbors. $W(\boldsymbol{x})$ is the classical cubic SPH kernel [42] and $\boldsymbol{g}$ is the gravitational acceleration. The relative velocity between particles p and q is $\boldsymbol{v}_{pq} = \boldsymbol{v}_p - \boldsymbol{v}_q$, $\nabla W(\boldsymbol{x}_q - \boldsymbol{x}_p)$ is the analytical gradient of the kernel W centered at particle p and evaluated at the location of particle q. Eq. (7) is the equation of state that links the pressure P_p with the density ρ_p, where ρ_0 is the density of the fluid at $P = 0$. Π_{pq} is the viscosity term defined as:

$$\Pi_{pq} = \begin{cases} -\frac{\alpha \bar{c}_{pq} \mu_{pq}}{\bar{\rho}_{pq}} & \boldsymbol{v}_{pq} \cdot \boldsymbol{r}_{pq} < 0 \\ 0 & \boldsymbol{v}_{pq} \cdot \boldsymbol{r}_{pq} > 0, \end{cases} \tag{8}$$

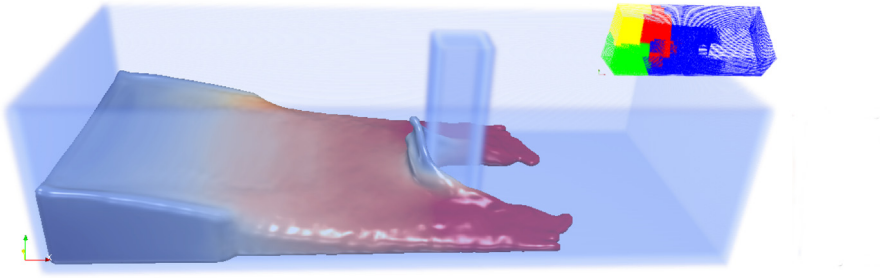
where $\boldsymbol{r}_{pq} = \boldsymbol{r}_p - \boldsymbol{r}_q$ is the vector between the two particles and $r_{pq} = |\boldsymbol{r}_{pq}|$ its magnitude. The constants are defined as: $\mu_{pq} = \frac{h \boldsymbol{v}_{pq} \cdot \boldsymbol{r}_{pq}}{r_{pq}^2 + \eta^2}$ and $\bar{c}_{pq} = c_{\text{sound}} \sqrt{|\boldsymbol{g}| h_{\text{swl}}}$.

We use the OpenFPM-based implementation to simulate a water column impacting onto a fixed obstacle. This "dam break" scenario is a standard test case for SPH simulation codes. A visualization of the OpenFPM result at three different time points is shown in Fig. 6. We compare the results and performance with those obtained using the popular open-source SPH code DualSPHysics [26]. The publicly available version of DualSPHysics only supports shared-memory multi-core platforms and GPGPUs, which is why we limit comparisons to these cases.

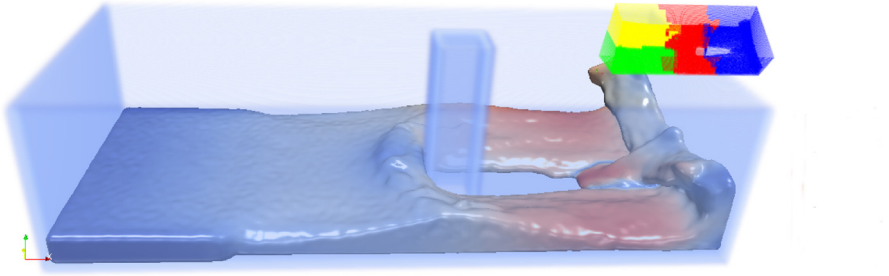
The present SPH implementation based on OpenFPM uses the same algorithms as DualSPHysics [26], with identical initialization, boundary conditions, treatment of the viscosity term, and Verlet time-stepping [49] with dynamic step size (computed identically in both codes according to the DualSPHysics paper [26]). The results are therefore directly comparable. In this test case, particles are not homogeneously distributed across the domain, and they significantly move during the simulation. Therefore, this provides a good showcase for the dynamic load balancing capability of OpenFPM.

We validate our simulation by calculating and comparing the velocity and pressure profiles at multiple points between OpenFPM and DualSPHysics [26]. We find that all pressure and velocity profiles are identical (not shown). We measure the performance of the OpenFPM-based implementation in comparison with the DualSPHysics code running on the 24 cores of a single cluster node. We simulate the dam-break case with 171,496 particles until a simulated physical time of 1.5 seconds. The OpenFPM code completes the entire simulation in about 500 seconds, whereas DualSPHysics requires about 950 seconds. The roughly two-fold better performance of OpenFPM is possibly attributed to the use of symmetry when evaluating the interactions and the use of optimized Verlet lists, which do not seem to be exploited in DualSPHysics [26].

(a) t = 0 s



(b) t = 0.43 s



(c) t = 0.95 s

Fig. 6. Visualization of the SPH dam-break simulation. We show the fluid particles at times 0, 0.43, and 0.95 s of simulated time, starting from a column of fluid in the left corner of the domain as shown. We use the OpenFPM SPH client to solve the weakly compressible Navier–Stokes equations with an equation of state for pressure, as given in Eqs. (4)–(7). The figure shows a density iso-surface indicating the fluid surface with color indicating the fluid velocity magnitude in units of meters/second. The small insets show the distribution of the domain onto 4 processors with different processors shown by different colors. The dynamic load balancing of OpenFPM automatically adjusts the domain decomposition to the evolution of the simulation in order to maintain scalability. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Since DualSPHysics is mainly optimized for use on GPGPUs, we also compare the OpenFPM-based implementation in distributed-memory mode with DualSPHysics running on a GPGPU. The benchmark is done with 15 million SPH particles using a nVidia GeForce GTX1080 GPU. The OpenFPM code reached the same performance when running on around 270 CPU cores of the benchmark machine and was faster when using more cores. This shows that OpenFPM can reach GPU performance on moderate numbers of CPU cores without requiring specialized CUDA code.

Since this test case is particularly demanding for load-balancing, we also use it to profile OpenFPM with respect to the fraction of time spent computing, communicating, and load-balancing. The results are shown in Table 4 for different numbers of particles on 1536 processors, hence testing the scalability of the code to large numbers of particles. The small insets in Fig. 6 show how the domain decomposition of OpenFPM adapts to the evolving particle distribution by dynamic load re-balancing (see Section 3.5). In this example, the load distribution strongly changes over time, due to the large bulk motion of the particles. Therefore, the dynamic load-balancing routines of OpenFPM consume anywhere between 5% and 25% of the total execution time, but their absolute runtime is independent of the number of particles. The overall performance of load balancing is limited by the performance of ParMetis [22]. OpenFPM thus reduces the degradation of parallel performance due to communication and load imbalance to the extent possible. Therefore, the relative fraction of communication and load-balancing decreases for increasing number of particles, whereas the average imbalance remains roughly constant due to the dynamic load balancing. Since load balancing and communication are not required when running on a single core, the percentage of time spent computing (second column in Table 4) can directly be interpreted as the parallel efficiency of the code on 1536 cores, which is, as expected, increasing with problem size.

4.3. Finite-difference reaction–diffusion code

As a third showcase, we consider a purely mesh-based application, namely a finite-difference code to numerically solve a reaction–diffusion system. Reaction–diffusion systems are widely studied due to their ability to form steady-state concentration patterns,

Table 4
Percentage of the total runtime spent on different tasks by OpenFPM for the SPH dam-break simulation on 1536 cores using different numbers of particles (1st column). The computation time is the average wall-clock time across processors spent on local computations, while the load imbalance is given by the difference between the maximum wall-clock time across processors and the average. Communication is the time taken by all mappings together, and DLB (dynamic load balancing) is the time taken by all load re-balancing steps to assign sub-domains to processors. The last column gives the total runtime of the simulation until a simulated time of 1.5 s. Due to the dynamic time step size, the total number of time steps performed ranges between 20,000 and 100,000, thus providing sufficient statistics for the percentages reported.

#particles	Computation (%)	Imbalance (%)	DLB (%)	Communication (%)	Time (s)
0.46M	19.2%	15.1%	25.7%	40.0%	148.40
1.20M	32.3%	30.4%	10.5%	26.8%	346.83
4.00M	50.0%	22.4%	10.6%	17.0%	1424.64
10.78M	64.0%	21.2%	6.81 %	8.09%	4666.78
14.63M	65.5%	21.5%	5.38%	7.71%	7035.01

including Turing patterns [68]. A particularly known example is the Gray–Scott system [69–72], which produces a rich variety of patterns in different parameter regimes. It is described by the following set of partial differential equations:

$$\begin{aligned} \frac{\partial u}{\partial t} &= D_u \Delta u - uv^2 + F(1-u) \\ \frac{\partial v}{\partial t} &= D_v \Delta v + uv^2 - (F+k)v\,, \end{aligned} \tag{9}$$

where D_u and D_v are the diffusion constants of the two species with concentrations u and v, respectively. The parameters F and k determine the type of pattern that is formed.

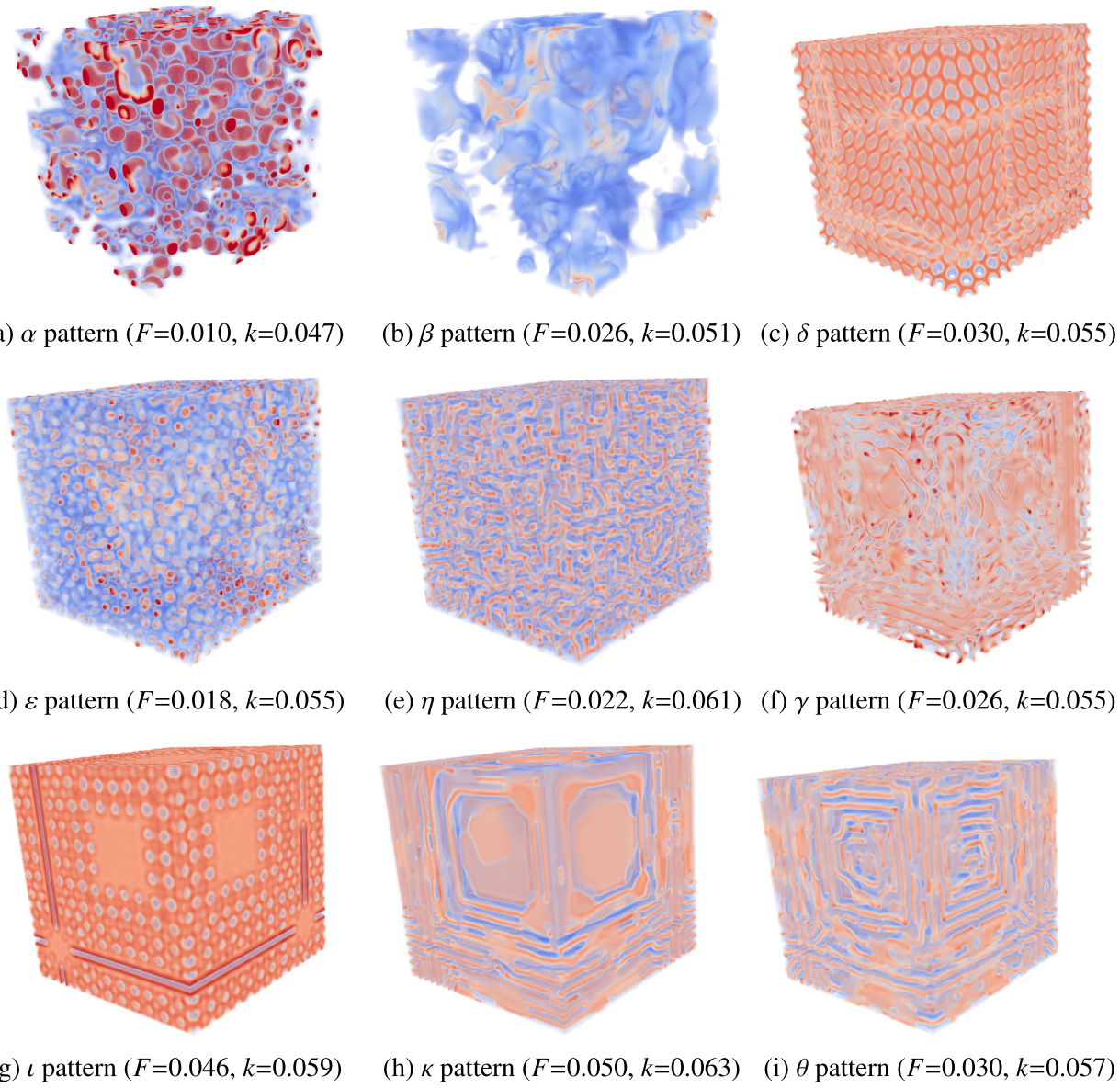
We implement an OpenFPM-based numerical solver for these equations using second-order centered finite-differences on a regular Cartesian mesh in 3D with periodic boundary conditions in all directions. We compare the performance of the OpenFPM-based implementation with that of an efficient AMReX-based solver [32]. Even though AMReX is capable of multi-resolution adaptive mesh refinement, where a direct comparison would be unfair, it uses a patch-based implementation, which, when restricted to one level, is extremely fast even in the uniform-resolution case. Indeed, the way one-level patches are implemented in AMReX is very similar to how OpenFPM meshes are implemented. AMReX therefore is the closest we could find to OpenFPM, which is why we use it as a benchmark also in the present uniform-resolution case. However, AMReX requires the user to tune the maximum mesh size for data distribution [32]. If we choose the maximum mesh size too large, AMReX does not have enough granularity to parallelize. If it is chosen too small, scalability is impaired by a larger ghost-layer communication overhead. In our benchmarks, we manually tune this parameter in order to ensure that the number of sub-meshes is always larger than the number of processor cores used with a small over all ghost-layer volume. The actual values used are given in the last columns of Table 5 and Table 6. OpenFPM does not require the user to set such a parameter, as the domain decomposition is determined automatically. For both AMReX and OpenFPM, we use MPI-only parallelism in order to compare the results.

For the benchmarks, we use the following dimensionless parameter values: $D_u = 2 \cdot 10^{-5}$, $D_v = 10^{-5}$, varying k and F as given in the legends of Fig. 7 to produce different patterns. To validate the simulation, we reproduce the nine patterns classified by Pearson [73], with visualizations shown in Fig. 7.

An OpenFPM source-code example of applying a simple 5-point finite-difference stencil to a regular Cartesian mesh is shown in Listing 4.3. The stencil is defined in line 2 as an OpenFPM grid key array with relative mesh coordinates. Here, the stencil object is called `star_stencil_2D` and consists of 5 points. In line 5, a mesh iterator is created for this stencil as applied to the mesh object `Old`. Lines 7–24 then loop over all mesh nodes and apply the stencil. The expression of the stencil (lines 18–20) is simplified by first defining aliases for the shifted nodes in lines 10–14, albeit this is not necessary.

Listing 4.3: OpenFPM code example for stencil operations on a regular Cartesian mesh

```
1  ///// finite-difference stencil definition
2  static grid_key_dx<2> star_stencil_2D[5] = {{0,0},{-1,0},{+1,0},{0,-1},{0,+1}};
3
4  ///// create an iterator for the stencil on the mesh "Old"
5  auto it = Old.getDomainIteratorStencil(star_stencil_2D);
6
7  while (it.isNext()) {
8      ///// define aliases for center, minus-x, plus-x, minus-y, plus-y.
9      ///// The template parameter is the stencil element.
10     auto Cp = it.getStencilGrid<0>();
11     auto mx = it.getStencilGrid<1>();
12     auto px = it.getStencilGrid<2>();
13     auto my = it.getStencilGrid<3>();
14     auto py = it.getStencilGrid<4>();
15
16     ///// apply the stencil to field U on mesh "Old" and store
17     ///// the result in the field U on mesh "New"
18     New.get<U>(Cp) = Old.get<U>(Cp) +
19         (Old.get<U>(my)+Old.get<U>(py)+Old.get<U>(mx)+Old.get<U>(px) -
20         4.0*Old.get<U>(Cp));
21
22     ///// Move to the next mesh node
23     ++it;
24 }
```



(a) α pattern (F=0.010, k=0.047) (b) β pattern (F=0.026, k=0.051) (c) δ pattern (F=0.030, k=0.055)

(d) ε pattern (F=0.018, k=0.055) (e) η pattern (F=0.022, k=0.061) (f) γ pattern (F=0.026, k=0.055)

(g) ι pattern (F=0.046, k=0.059) (h) κ pattern (F=0.050, k=0.063) (i) θ pattern (F=0.030, k=0.057)

Fig. 7. Visualizations of the OpenFPM simulations of nine steady-state patterns produced by the Gray–Scott reaction-diffusion system in 3D [73] for different values of the parameters F and k.

The performance of OpenFPM compared to AMReX is shown in Table 5 and Table 6 and in Fig. 8 for two different problem sizes. OpenFPM scales slightly better than AMReX, with wall-clock times in the same range. For the small problem of 256^3 mesh nodes (Table 5), we use up to 24 cores within one cluster node. Therefore, this benchmark shows how the codes scale in shared memory. The large problem of 784^3 mesh nodes (Table 6) uses up to 256 cores, hence spanning multiple computer nodes and showing how the codes scale in distributed memory. Both AMReX and OpenFPM use mixed C++/Fortran code for this benchmark, with all stencil iterations implemented in Fortran. Because Fortran provides native support for multi-dimensional arrays, it produces more efficient assembly code than C++. In our tests, a fully C++ version was about 20% slower than the hybrid C++/Fortran. In addition, we show in Table 7 how the runtime of the OpenFPM client changes when using different distributions of the same total number of cores across different numbers of sockets. The memory bus saturates when using more than 4 cores per socket, which is consistent with the code profiling results suggesting that memory access is the main bottleneck for this code.

4.4. Vortex methods

As a fourth showcase, and in order to show how OpenFPM handles hybrid particle-mesh problems, we consider a full vortex-in-cell [52] code, a hybrid particle-mesh method to numerically solve the incompressible Navier–Stokes equations in vorticity form with periodic boundary conditions. These equations are:

$$\begin{aligned} \frac{D\boldsymbol{\omega}}{Dt} &= (\boldsymbol{\omega}\cdot\nabla)\boldsymbol{u} + \nu\Delta\boldsymbol{\omega} \\ \Delta\boldsymbol{\psi} &= -\nabla\times\boldsymbol{u} = -\boldsymbol{\omega}\,, \end{aligned} \tag{10}$$

Table 5
Performance of the OpenFPM finite-difference code compared with AMReX [32]. Total times are given in seconds as mean±standard deviation over 10 independent runs for a fixed problem size of 256^3 mesh nodes over 5000 time steps (strong scaling). The mesh-size parameters used for AMReX are given in the last column.

#cores	OpenFPM (seconds)	AMReX (seconds)	AMReX param
1	393.1 ± 1.3	388.5 ± 1.5	256
2	207.5 ± 1.3	265.0 ± 0.8	128
4	105.8 ± 1.3	144.8 ± 0.3	128
8	65.1 ± 2.1	106.6 ± 2.6	128
12	65.6 ± 2.6	90.9 ± 5.0	64
16	57.6 ± 1.9	173.6 ± 3.6	64
20	56.8 ± 2.0	66.0 ± 1.7	64
24	60.5 ± 0.3	60.9 ± 4.0	64

Table 6
Performance of the OpenFPM finite-difference code compared with AMReX [32]. Total times are given in seconds as mean±standard deviation over 5 independent runs for a fixed problem size of 784^3 mesh nodes over 100 time steps (strong scaling). The mesh-size parameters used for AMReX are given in the last column.

#cores	OpenFPM (seconds)	AMReX (seconds)	AMReX param
1	199.7 ± 0.4	205.5 ± 1.4	740
4	52.5 ± 0.3	56.1 ± 0.3	370
8	32.0 ± 0.1	39.7 ± 0.1	370
16	15.2 ± 0.1	28.1 ± 0.1	294
32	13.8 ± 0.1	8.4 ± 0.1	233
64	4.3 ± 0.1	6.8 ± 0.1	185
128	3.1 ± 0.1	4.3 ± 0.2	147
256	2.3 ± 0.1	2.7 ± 0.1	117

Table 7
Average runtime of the OpenFPM finite-difference client using different numbers of cores per socket for the same total of 8 cores for the large problem from Table 6.

#cores per socket × #sockets	OpenFPM (seconds)
8 × 1	61.0
4 × 2	32.0
2 × 4	25.5
1 × 8	24.2

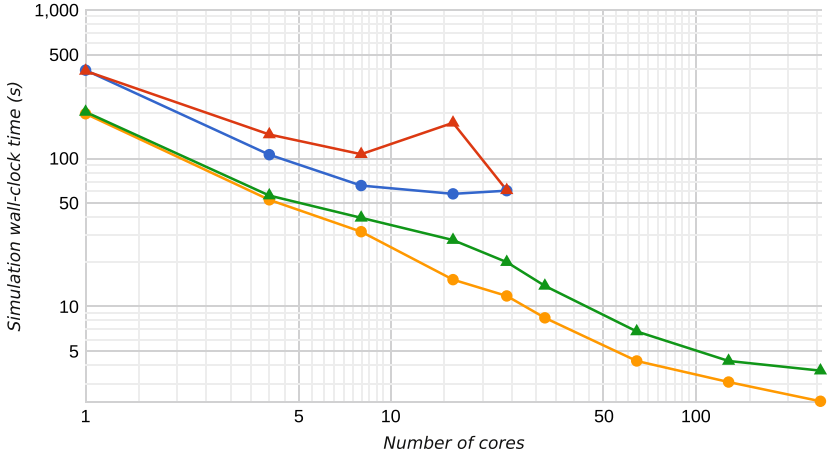


Fig. 8. Strong scaling of the OpenFPM finite-difference client (circles) in comparison with AMReX [32] (triangles). Shown is the wall-clock time in seconds to complete 5000 time steps of the Gray–Scott finite-difference code (7-point stencil) on a 256^3 uniform Cartesian mesh (see Table 5; OpenFPM: blue circles, AMReX: red triangles) and to complete 100 time steps of the same problem on a 784^3 mesh (see Table 6; OpenFPM: yellow circles, AMReX: green triangles) using different numbers of cores. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

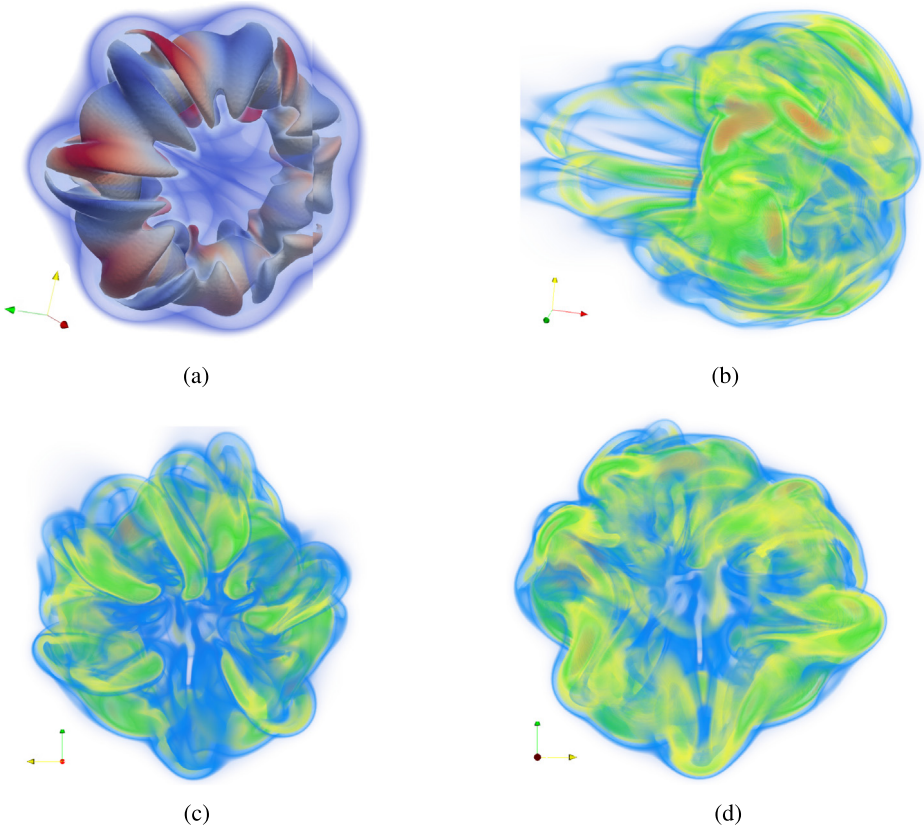


Fig. 9. Visualization of the OpenFPM simulation of a vortex ring at Re=3750 using a hybrid particle-mesh Vortex Method (Algorithm 2) to solve the incompressible Navier–Stokes equations with 256 million particles on 3072 processors. Results are visualized for $t = 195.5$ when the ring is just about to become turbulent. (a) The iso-surfaces of vorticity highlight the tubular dipole structures in the vortex ring. Color corresponds to the x-component of the vorticity with red indicating positive signs and blue negative signs. (b)–(d) Three different views of a volume rendering of four vorticity bands: orange is $\|\boldsymbol{\omega}\|^2 = 2.3 \ldots 3.239$, green is $\|\boldsymbol{\omega}\|^2 = 1.16 \ldots 1.372$, yellow is $\|\boldsymbol{\omega}\|^2 = 0.7 \ldots 0.815$, and blue is $\|\boldsymbol{\omega}\|^2 = 0.3 \ldots 0.413$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

with $\boldsymbol{\omega}$ the vorticity, $\boldsymbol{\psi}$ the vector stream function, ν the kinematic viscosity, and $\boldsymbol{u}$ the velocity of the fluid. The operator $\frac{D}{Dt}$ denotes a Lagrangian (material) time derivative [52]. We numerically solve these equations using an OpenFPM-based implementation of the classical vortex-in-cell method as given in Algorithm 2 with two-stage Runge–Kutta time stepping. Particle-mesh and mesh-particle interpolations use the moment-conserving M_4' interpolation kernel [74].

We run a simulation that reproduces previous results of a self-propelling vortex ring [75]. The vortex ring is initialized on a mesh of size $1600 \times 400 \times 400$ using

$$\boldsymbol{\omega}_0 = \frac{\Gamma}{\pi\sigma^2} \mathrm{e}^{-s/\sigma}, \tag{11}$$

where $s^2 = (z - z_c)^2 + [(x - x_c)^2 + (y - y_c)^2 - R^2]$, with ring radius $R = 1$, thickness $\sigma = R/3.531$, and the domain $(0 \ldots 5.57,\ 0 \ldots 5.57,\ 0 \ldots 22.0)$ with periodic boundary conditions in all directions. We set the ring's circulation to $\Gamma = 1$, and the center of the torus defining the initial vortex ring to $x_c = y_c = z_c = 2.785$.

A Runge–Kutta time-stepping scheme of order 2 is used with fixed step size $\delta t = 0.0025$. This time-stepping method is chosen for two reasons: (1) to demonstrate that distributed multi-stage schemes are easily realized in OpenFPM, (2) for comparability of results, because the PPM code [38] we benchmark against did use the same. The time step size is chosen empirically so as to guarantee numerical stability. Like in the benchmark PPM code, all differential operators are discretized using second-order symmetric finite differences on the mesh. We use 256 million particles distributed across 3072 processors to simulate the behavior of the vortex ring at Reynolds number $\mathrm{Re} = 3750$ until final time $t = 225.5$. VTK files are written by OpenFPM and directly visualized using ParaView [65]. We observe the same patterns and structures for the ring as in Ref. [75], see Fig. 9.

The performance and scalability of the OpenFPM code are limited by the linear system solver required for computing the velocity from the vorticity on the mesh, i.e., by solving the Poisson equation. In this benchmark, OpenFPM internally uses a solver provided by the PetSc library [54]. We benchmark the parallel scalability of the PetSc solver and of the overall code in a weak scaling starting from a $109 \times 28 \times 28$ mesh on 1 processor up to $1207 \times 317 \times 317$ mesh nodes on 1536 processors. We separately time the efficiency of the PetSc solver and of the OpenFPM parts of the code (particle-mesh/mesh-particle interpolation, remeshing, time integration, right-hand side evaluation). The results are shown in Fig. 10. Within a cluster node (1. . .24 cores), the decay in efficiency can be explained by the shared memory bandwidth (see Table 1). PetSc shows another marked drop in efficiency when transitioning from one cluster node to two nodes (48 cores). After that, the efficiency remains stable until 768 cores, when it starts to slowly drop again.

To put these results into perspective, we compare the particle-mesh interpolation part of the code with the corresponding part of a PPM-based hybrid particle-mesh vortex code previously used [38]. We only compare this part of the code in order to exclude

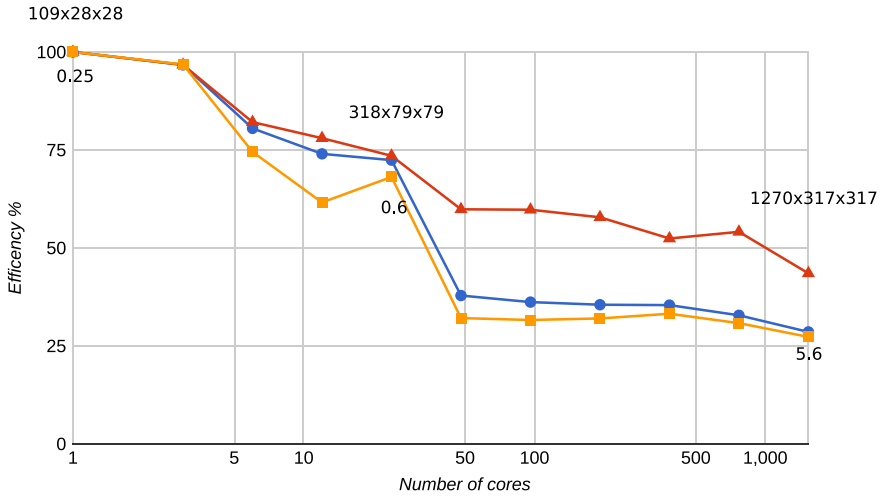


Fig. 10. Parallel efficiency of the OpenFPM-based hybrid particle-mesh vortex code for a scaled-size problem (weak scaling). The problem size scales from $109 \times 28 \times 28$ mesh nodes on 1 processor core to $1207 \times 317 \times 317$ mesh nodes on 1536 cores (24 cores per node). We separately show the parallel efficiency of the PetSc Poisson solver (yellow squares), the OpenFPM parts of the code (red triangles) and the resulting overall scalability (blue circles). For three points, the problem sizes and the overall wall-clock time per time step in seconds are indicated next to the symbols. We note that the computational complexity of the Poisson solver is not linear with problem size.

differences between PetSc and the own internal solvers of PPM. Interpolating two million particles to a $129 \times 65 \times 65$ mesh using the M_4' interpolation kernel [74] takes 0.078 s in OpenFPM on a single core, and it takes exactly the same amount of time in PPM on the same computer. Performing a weak scaling starting from a 128^3 mesh on 1 processor, the OpenFPM particle-mesh interpolation reaches a parallel efficiency of 75% on 128 cores (16 nodes using 8 cores of each node). This is comparable with the scalability of PPM on the same test problem (see Fig. 13 of Ref. [38]).

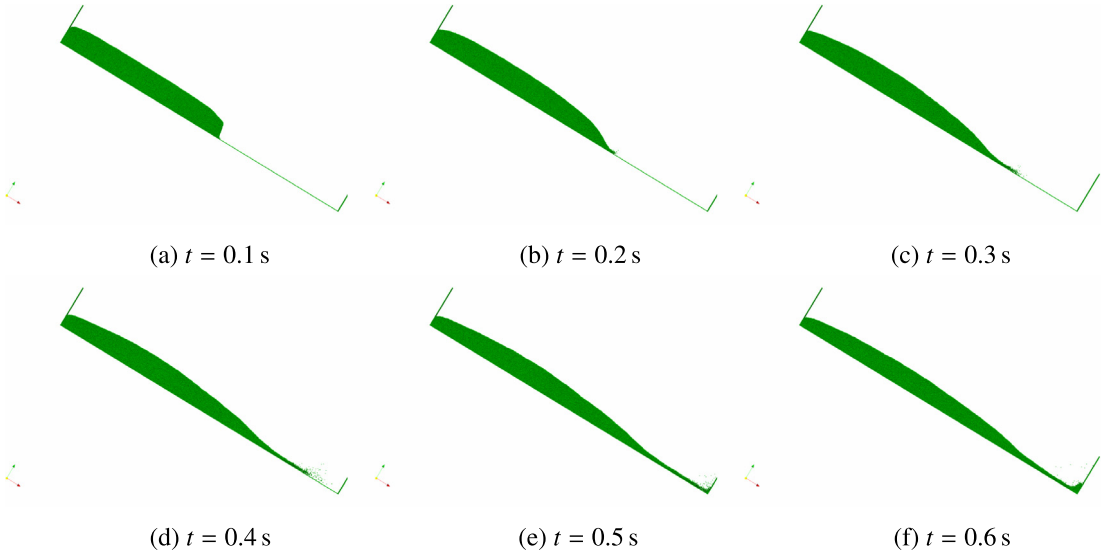
Algorithm 2 Vortex-in-Cell method with two-stage Runge–Kutta (RK) time integration.

procedure VortexMethod
  initialize the vortex ring on the mesh
  do a Helmholtz–Hodge projection to make the vorticity divergence-free
  initialize particles at the mesh nodes
  while $t < t_{\text{end}}$ **do**
    calculate velocity $\boldsymbol{u}$ from the vorticity $\boldsymbol{\omega}$ on the mesh (Poisson equation solver)
    calculate the right-hand side of Eq. (10) on the mesh and interpolate to particles
    interpolate velocity $\boldsymbol{u}$ to particles
    1st RK stage: move particles according to the velocity; save old position in $\boldsymbol{x}_{\text{old}}$
    interpolate vorticity $\boldsymbol{\omega}$ from particles to mesh
    calculate velocity $\boldsymbol{u}$ from the vorticity $\boldsymbol{\omega}$ on the mesh (Poisson equation solver)
    calculate the right-hand side of Eq. (10) on the mesh and interpolate to particles
    interpolate velocity $\boldsymbol{u}$ to particles
    2nd RK stage: move particles according to the velocity starting from $\boldsymbol{x}_{\text{old}}$
    interpolate the vorticity $\boldsymbol{\omega}$ from particles to mesh
    create new particles at mesh nodes (remeshing)

4.5. Discrete element methods

Discrete element methods (DEM) are important for the study of granular materials, in particular for determining effective macroscopic dynamics for which the governing equations are not known. They simulate each grain of the material explicitly, with all collisions fully resolved. Force and torque balance over the grains then governs their Newtonian mechanics. The main difference to MD is that forces are only exerted by direct contact, and that contact sites experience elastic deformation. DEM is therefore essentially a collision-event method. In order to correctly integrate the dynamics over time, lists of contact sites between particles need to be managed. Since these lists are of varying length, both in time and in space, and collisions involving ghost particles need to be properly accounted for in the lists of the respective source particles, parallelizing DEM is not trivial. Previously, DEM has been parallelized onto distributed-memory machines using the PPM Library [38], enabling DEM simulations of 122 million elastic spheres distributed over 192 processors [76].

We implement the same DEM simulation using OpenFPM in order to directly compare performance with the previous PPM implementation. For comparability, we pre-allocate the contact list to their maximum length, at the expense of a higher communication overhead. We note that this is not required, as more dynamic list management can be implemented using OpenFPM, but it is how the benchmark PPM code does it.



(a) $t = 0.1$ s (b) $t = 0.2$ s (c) $t = 0.3$ s

(d) $t = 0.4$ s (e) $t = 0.5$ s (f) $t = 0.6$ s

Fig. 11. Visualization of the Discrete Element Method (DEM) simulation of an avalanche of spheres down an inclined plane (inclination angle: 31.2 degrees) at different simulated times. While the simulation is 3D, we visualize the same 2D cross-section here as in Ref. [76] in order to allow direct visual comparison.

We implement the classical Silbert grain model [77], including Hertzian contact forces and elastic deformation of the grains, as previously considered [76]. All particles have the same radius R, mass m, and moment of inertia I. Each particle p is represented by the location of its center of mass $\boldsymbol{r}_p$. When two particles p and q are in contact with each other, the radial elastic contact deformation is given by:

$$\delta_{pq} = 2R - r_{pq}\,, \tag{12}$$

with $\boldsymbol{r}_{pq} = \boldsymbol{r}_p - \boldsymbol{r}_q$ the vector between the two particle centers and $r_{pq} = \|\boldsymbol{r}_{pq}\|_2$ its length. The evolution of the tangential elastic deformation $\boldsymbol{u}_{t_{pq}}$ is integrated over the time duration of a contact using the explicit Euler scheme as:

$$\boldsymbol{u}_{t_{pq}} = \boldsymbol{u}_{t_{pq}} + \boldsymbol{v}_{t_{pq}}\delta t\,, \tag{13}$$

where δt is the simulation time step and $\boldsymbol{v}_{t_{pq}} + \boldsymbol{v}_{n_{pq}} = \boldsymbol{v}_p - \boldsymbol{v}_q = \boldsymbol{v}_{pq}$ are the tangential and radial components of the relative velocity between the two colliding particles. For each pair of particles that are in contact with each other, the normal and tangential forces are [77]:

$$\boldsymbol{F}_{n_{pq}} = \sqrt{\frac{\delta_{pq}}{2R}}\ \left(k_n\delta_{pq}\boldsymbol{n}_{pq} - \gamma_n m_{\text{eff}}\boldsymbol{v}_{n_{ij}}\right)\,, \tag{14}$$

$$\boldsymbol{F}_{t_{pq}} = \sqrt{\frac{\delta_{pq}}{2R}}\ \left(-k_t\boldsymbol{u}_{t_{pq}} - \gamma_t m_{\text{eff}}\boldsymbol{v}_{t_{pq}}\right)\,, \tag{15}$$

where $k_{n,t}$ are the elastic constants in normal and tangential direction, respectively, and $\gamma_{n,t}$ the corresponding friction constants. The effective collision mass is given by $m_{\text{eff}} = \frac{m}{2}$. In addition, the tangential deformation is rescaled to enforce Coulomb's law (with $\mu = 1$) as described [76,77]. The total resultant force $\boldsymbol{F}_p^{\text{tot}}$ and torque $\boldsymbol{T}_p^{\text{tot}}$ on particle p are then computed by summing the contributions over all current collision partners q and the gravitational force vector. We integrate the equations of motion using the second-order accurate leapfrog scheme, as:

$$\boldsymbol{v}_p^{n+1} = \boldsymbol{v}_p^n + \frac{\delta t}{m}\boldsymbol{F}_p^{\text{tot}}\,,\qquad \boldsymbol{r}_p^{n+1} = \boldsymbol{r}_p^n + \delta t\,\boldsymbol{v}_p^{n+1}\,,\qquad \boldsymbol{\omega}_p^{n+1} = \boldsymbol{\omega}_p^n + \frac{\delta t}{I}\boldsymbol{T}_p^{\text{tot}}\,, \tag{16}$$

where $\boldsymbol{r}_p^n$, $\boldsymbol{v}_p^n$, and $\boldsymbol{\omega}_p^n$ are the center-of-mass position, velocity, and rotational/angular velocity of particle p at time step n, all stored as particle properties.

We simulate an avalanche down an inclined plane, which has previously been used as a benchmark case for distributed-memory parallel DEM simulations using the PPM Library [76]. The simulation, visualized in Fig. 11, uses 108,908 particles radii R uniformly at random between 1.00 mm and 1.12 mm, and inertial moment I and mass m computed for each particle according to the density of sand of 1700 kg/m^3. We use $k_n = 7849$ N/m, $k_t = 2243$ N/m, $\gamma_n = \gamma_t = 34010\,\text{s}^{-1}$ (notice the typo in $\gamma_{n,t}$ in Ref. [76]). The size of the simulation domain is 1.5 m $\times$ 0.01 m $\times$ 0.21 m and has fixed-boundary walls in x-direction and in negative z-direction, a free-space boundary in positive z-direction, and periodic boundaries in y-direction. Initially, all particles are placed on a Cartesian lattice inside a box of size 0.74 m $\times$ 0.01 m $\times$ 0.2 m and equilibrated (i.e., let to settle). After equilibration, the simulation box is inclined by 31.2 degrees by rotating the gravity vector accordingly. The simulation time step is $\delta t = 10^{-6}$ s.

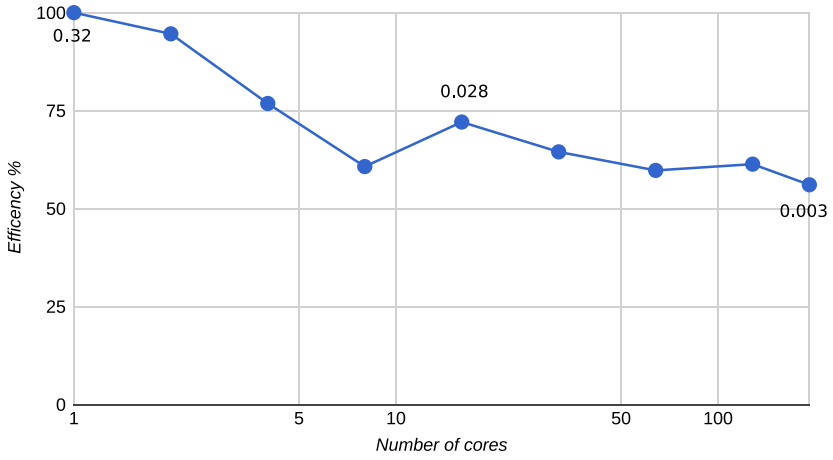


Fig. 12. Strong scaling of the OpenFPM DEM client using a fixed problem size of 677,310 particles distributed onto up to 192 cores using 8 cores on each cluster node. The numbers near the symbols indicate the absolute wall-clock times per time step in seconds.

We compare the performance of the OpenFPM DEM client with the legacy PPM code [76] using the same test problem. In Fig. 12, we plot the parallel efficiency of the OpenFPM DEM simulation for a strong scaling on up to 192 processors. OpenFPM completes one time step with 677,310 particles on one core in 0.32 s, whereas the PPM-based code needs 0.33 s per time step for 635,780 particles. On 192 cores, OpenFPM completes a time step of the same problem in 3 ms with a parallel efficiency of 56%. In comparison, the PPM DEM client needs 3.67 ms per time step on 192 cores with a parallel efficiency of 47% [76]. The literature results are compatible with our present benchmark, as the PPM code was previously tested on a Cray XT-3 machine, whose AMD Opteron 2.6 GHz processors are about 3 times slower than the 2.5 GHz Intel Xeon E5-2680v3 of the present benchmark machine, indicating similar effective performance for both codes.

4.6. Particle-swarm covariance-matrix-adaptation evolution strategy (PS-CMA-ES)

One of the main advantages of OpenFPM over other simulation frameworks is that OpenFPM can transparently handle spaces of arbitrary dimension. This enables simulations in higher-dimensional spaces, such as the four-dimensional spaces used in lattice quantum chromodynamics [78,79], and it also enables parallelization of non-simulation applications that require high-dimensional spaces, including image analysis algorithms [80] and Monte-Carlo sampling strategies [81].

A particular Monte-Carlo sampler used for stochastic real-valued optimization is the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [82,83]. The goal is to find a (local) optimum of a (non-convex) function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. In practical applications, the dimensionality n of the domain is 10 to 50. CMA-ES has previously been parallelized by running multiple instances concurrently that exchange information akin to a particle-swarm optimizer. The resulting particle-swarm CMA-ES (PS-CMA-ES) has been shown to outperform standard sequential CMA-ES on multi-funnel functions [84], and an efficient Fortran implementation of it is available, pCMAlib [85].

Here, we implement PS-CMA-ES using OpenFPM in order to demonstrate how OpenFPM transparently handles high-dimensional spaces and also extends to non-simulation applications, such as sampling and computational optimization. In our implementation, each OpenFPM particle corresponds to one CMA-ES instance, hence implementing PS-CMA-ES through particle interactions across processors. To validate the OpenFPM implementation, we use the multi-modal test function f_{15} from the IEEE CEC2005 set of standard optimization test functions [86]. In order to directly compare with pCMAlib, we limit the total number of function evaluations allowed to 5×10^5 and run both implementations 25 times each. We compare the success rate, i.e., the fraction of the 25 runs that found the true global optimum, and the success performance, i.e., the average best function value found across all 25 runs, in 10, 30, and 50 dimensions [84,85]. The results from the OpenFPM-based implementation are identical with those from pCMAlib when using the same pseudo-random number sequence (not shown).

We also compare the runtime performance and parallel scalability of the OpenFPM-based implementation with those of the highly optimized Fortran pCMAlib. The results are shown in Fig. 13 for dimension 50. For dimensions 10 and 30, the results are analogous and not shown. Since the total number of function evaluations is kept constant at 5×10^5, irrespective of the number of cores used, this amounts to a strong scaling. However, the number of swarm particles is always chosen equal to the number of cores, as this is a hard requirement of pCMAlib, while OpenFPM would not require this. In all cases, the OpenFPM implementation is about one third faster than pCMAlib.

Implementing arbitrary-dimensional codes using OpenFPM is straightforward, as the dimensionality is a template parameter in all data structures. While, of course, the memory requirement for mesh data structures grows exponentially with dimension, the size of particle data structures scales linearly. The code example in Listing 4.6 illustrates how the PS-CMA-ES data structures in 50 dimensions are defined in OpenFPM. All iterators and mappings work transparently. This example illustrates that OpenFPM naturally extends to problems in higher-dimensional spaces, which the PPM Library [38] could not.

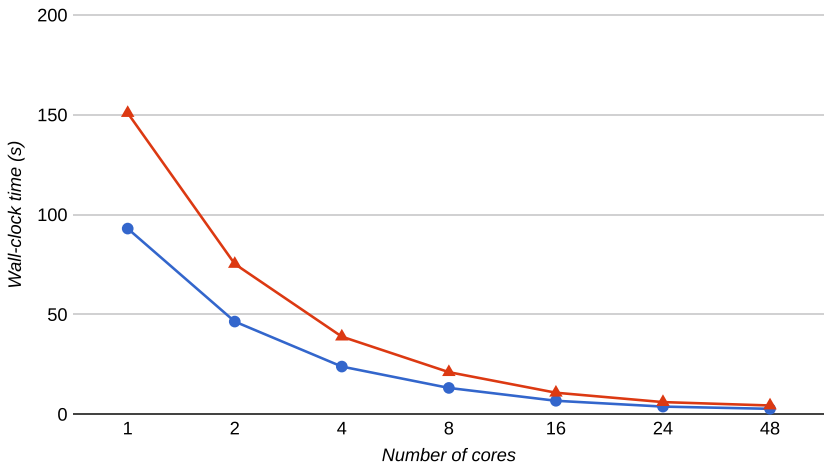


Fig. 13. Strong scaling of the OpenFPM PS-CMA-ES client (blue circles) in comparison with the Fortran pCMAlib (red triangles) scaling from 1 to 48 cores for IEEE CEC2005 test function f_{15} in dimension 50. Shown is the minimum (over 25 independent repetitions) total wall-clock time in seconds for 5×10^5 function evaluations.

Listing 4.6: OpenFPM code example for high-dimensional spaces

```
constexpr int dim = 50;       // define the dimensionality

///// Define the optimization domain as (-5:5)^dim
Box<dim,double> domain;
for (size_t i = 0; i < dim; i++) {
        domain.setLow(i,-5.0);
        domain.setHigh(i,5.0);
}

///// Define non-periodic boundary conditions (free space)
size_t bc[dim];
for (size_t i = 0; i < dim; i++) {bc[i] = NON_PERIODIC;};

///// There are no ghost layers needed for this problem
Ghost<dim,double> g(0.0);

///// allocate 8 particles
vector_dist<dim,double,aggregate<double,double[dim]>> particles(8,domain,bc,g);

///// get an iterator over particles and loop over all of them
auto it = particles.getDomainIterator();
while (it.isNext()) {
   .......   // do PS-CMA-ES here
    ++it;
}
```

5. Conclusions

We have presented OpenFPM, an open-source framework for particle and particle-mesh codes on parallel computers. OpenFPM implements abstract data structures and operators for particles-only and hybrid particle-mesh methods [1]. Any method that can be expressed in terms of these abstractions can be implemented using OpenFPM. This includes simulations of both discrete and continuous models, as well as non-simulation applications such as computational optimization [84] and image analysis [80]. The same abstractions were already implemented in the discontinued PPM Library [38], which has enabled particle-mesh simulations of unprecedented scalability and performance over the past 15 years [1]. OpenFPM extends this to a modern software-engineering framework using C++ Template Meta-Programming (TMP), continuous integration, and rigorous unit testing. The parallelization infrastructure provided by OpenFPM includes runtime load (re-)balancing, parallel and distributed HDF5 file I/O, checkpoint-restart on different numbers of processors, transparent iterators for particles and mesh nodes, and adaptive domain decompositions. This infrastructure is supplemented with frequently used numerical solvers and a range of convenience functions, including direct VTK file output for visualization of simulation results using the open-source software ParaView [65].

We have described the architectural principles of OpenFPM and provided an overview of its functionality. The main innovation of OpenFPM is the extensive use of TMP for compile-time code generation, which is still not widely used in scientific programming. While several existing frameworks use templates, TMP goes one step further in that it implements code-generation logic that goes beyond simple substitutions of templated type parameters. Due to this, for example, OpenFPM does not need declare a class “particle”, but simply specifies the list of particle properties as variadic templates from which the code for the data structures is automatically generated at compile time. This is different from traditional use of templates, where an array is created over a templated type at design time. In particular, TMP enables OpenFPM particles to have arbitrarily complex properties (in fact, any C++ object) while serialization for communication and file I/O remains automatic by analyzing the variadic template at compile time and automatically generating the required implementations. Moreover, analysis of the variadic template allows the compiler to tune the memory layout at compile

time, depending on the data structures actually used by the client and on the target hardware platform, and it enables simulations in arbitrary-dimensional spaces. We consider TMP a viable tool for scientific computing, which could further increase the versatility of simulation frameworks, as well as improve their runtime performance, while keeping code maintenance costs low.

We presented six test cases that represent the main classes of applications: continuous particle methods (SPH), discrete particle methods (molecular dynamics), high-dimensional problems (PS-CMA-ES), mesh codes (finite differences), hybrid particle-mesh codes (vortex methods), and event-driven codes (DEM). In all cases, the number of lines of code we needed to write in order to implement a fully scalable OpenFPM client was significantly less than what would have been necessary in a stand-alone application. The total numbers of lines of C++ code for the benchmarks presented here were: 192 for molecular dynamics, 550 for SPH, 140 (+37 lines of Fortran) for Gray–Scott, 500 for the vortex method, 430 for the DEM, and 950 for PS-CMA-ES. This includes all code needed for file I/O, visualization, and parameter file parsing and gives a good indication of the developer efficiency achievable with OpenFPM. All example clients are available in the OpenFPM source code repository.

We have benchmarked OpenFPM on up to 3072 processor cores, simulating systems with millions of degrees of freedom. Despite the automatic and transparent parallelization in OpenFPM, code performance and scalability in all examples was comparable to or better than those of state-of-the-art application-specific codes. For molecular dynamics, wall-clock times per time step were between 0.5 ms and 1 s, almost reaching the performance and scalability of the highly optimized LAMMPS code [30]. For SPH, OpenFPM outperformed the popular DualSPHysics CPU code [26] by about a factor of two, surpassing GPU performance when using 270 CPU cores or more. Solving a finite-difference system on a regular Cartesian mesh, OpenFPM outperformed the highly optimized AMReX code [32], both in terms of scalability and performance. When using Vortex Methods [52] to simulate incompressible fluid flow, OpenFPM was able to compute vortex-ring dynamics at Re=3750 using 256 million particles on 3072 processor cores and achieved state-of-the-art parallel efficiencies in all benchmarks. Using DEM to simulate a granular avalanche down an inclined plane illustrated OpenFPM's capability to handle complex particle properties, such as contact lists, outperforming the previous PPM code [38,76]. Finally, we illustrated the use of OpenFPM in high-dimensional problems by implementing PS-CMA-ES and comparing with the pCMAlib Fortran library [85]. This benchmark has shown the simplicity with which OpenFPM handles different space dimensions, while maintaining performance and scalability. Taken together, OpenFPM offers state-of-the-art performance and scalability at a reduced code development overhead. It overcomes the main limitations of the PPM Library [38] by extending to spaces of arbitrary dimension and allowing particles to carry arbitrary data types (C++ objects) as particle properties. It also adds automatic dynamic load (re-)balancing, transparent internal memory management and re-alignment, parallel checkpoint-restart, visualization file output, and custom distributed template expressions.

OpenFPM is going to be supported and developed in the long term. In the future, we plan to add the following functionalities to OpenFPM: transparent support for Discretization-Corrected Particle-Strength Exchange (DC-PSE) operators for the consistent discretization of differential operators on arbitrary particle distributions [45], an efficient distributed multi-grid solver for the general Poisson equation, 3D rendering capabilities for real-time *in-situ* visualization of a running simulation on screens and in virtual-reality environments, a compiler and development environment for application-specific language front-ends to OpenFPM [40,41,87], static (compile-time) and dynamic (runtime) code analysis [87] and optimization in order to reduce communication overhead to the required minimum, as well as support for adaptive-resolution particle representations [48,88,89] and GPU calculations [90]. In addition, we will further improve performance and scalability, e.g., by optimizing the domain decomposition and sub-domain merging implementations and by using space-filling curves, such as Morton curves, to constrain processor assignment.

The source code of OpenFPM and all benchmark clients presented here, virtual machines of various operating systems with a complete OpenFPM environment pre-installed, virtualized Docker containers, documentation, example applications, and tutorial videos are freely available from http://openfpm.mpi-cbg.de. We hope that the flexibility, free availability, performance, quality of documentation, and long-term support of OpenFPM will make it a standard platform for particles-only and hybrid particle-mesh simulations of discrete and continuous models on parallel computer hardware, as well as for non-simulation applications, such as evolutionary optimization strategies and particle-based image-analysis methods [80,91].

Acknowledgments

We thank all members of the MOSAIC Group for the many fruitful discussions. We particularly thank the early adopters and test users of OpenFPM whose feedback has helped improve the library throughout: Prof. Nikolaus Adams and Dr. Stefan Adami (both TU Munich, Germany), Prof. Marco Ellero (Swansea University, UK), Prof. Bernhard Peters (University of Luxembourg, Luxembourg), Prof. Bonnefoy (École des Mines Saint-Étienne, France), and Dr. Yaser Afshar (University of Michigan, Ann Arbor, USA). We thank the Centre for Information Services and High Performance Computing (ZiH) at TU Dresden for generous allocation of computer time. This project was supported in parts by the Deutsche Forschungsgemeinschaft (DFG), Germany under project "OpenPME" and by the German Federal Ministry of Research and Education (BMBF), Germany under grant number 031L0044.

References

[1] I.F. Sbalzarini, Intl. J. Distr. Syst. Technol. 1 (2) (2010) 40–56.
[2] OpenACC web page [cited 2018]. URL https://www.openacc.org/.
[3] OpenMP web page [cited 2018]. URL http://www.openmp.org.
[4] J. Nickolls, I. Buck, M. Garland, K. Skadron, Queue 6 (2) (2008) 40–53, http://dx.doi.org/10.1145/1365490.1365500.
[5] J. Stone, D. Gohara, G. Shi, Comput. Sci. Eng. 12 (3) (2010) 66.
[6] R.W. Numrich, J. Reid, SIGPLAN Fortran Forum 17 (2) (1998) 1–31, http://dx.doi.org/10.1145/289918.289920.
[7] C.H. Koelbel, The High Performance FORTRAN Handbook, MIT Press, 1993.
[8] T. El-Ghazawi, L. Smith, Proc. 2006 ACM/IEEE Conference on Supercomputing, SC '06, ACM, New York, NY, USA, 2006, p. 27, http://dx.doi.org/10.1145/1188455.1188483.
[9] N. Carriero, D. Gelernter, Commun. ACM 32 (4) (1989) 444–458.
[10] A.A. Wray, A Manual of the Vectoral Language, Internal Report, NASA Ames Research Center, Moffett Field, California, 1988.
[11] J. Bezanson, S. Karpinski, V.B. Shah, A. Edelman, Julia: A Fast Dynamic Language for Technical Computing, Tech. Rep., 2012, arXiv:1209.5145.
[12] Message Passing Interface Forum, MPI: A Message-Passing Interface StandArd, Version 3.0, High-Performance Computing Center Stuttgart, 2012.

[13] E. Gabriel, E.F. Graham, G. Bosilca, A. Thara, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall, Proceedings, 11th European PVM/MPI Users' Group Meeting, 2004, pp. 97–104.
[14] W. Gropp, Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, London, UK, UK, 2002, p. 7.
[15] H. Kaiser, M. Brodowicz, T. Sterling, Proceedings of the 2009 International Conference on Parallel Processing Workshops, ICPPW '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 394–401, http://dx.doi.org/10.1109/ICPPW.2009.14.
[16] K. Fürlinger, T. Fuchs, R. Kowalewski, 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016, pp. 983–990. arXiv:1610.01482.
[17] L.V. Kale, S. Krishnan, Proc. 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, 1993, pp. 91–108.
[18] E. Gallopoulos, E. Houstis, J.R. Rice, IEEE Comput. Sci. Eng. (1994).
[19] Modelica web pge [cited 2018]. URL https://www.modelica.org/.
[20] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, D. Padua, in: C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 269–288.
[21] Message Passing Interface Forum, MPI: A Message-Passing Interface StandArd, Version 2.1, High-Performance Computing Center Stuttgart, 2008.
[22] G. Karypis, V. Kumar, J. Parallel Distrib. Comput. 48 (1) (1998) 71–95, http://dx.doi.org/10.1006/jpdc.1997.1403.
[23] H. Jasak, A. Jemcov, Z. Tuković, Proc. Intl. Workshop on Coupled Methods in Numerical Dynamics, Dubrovnik, Croatia, 2007, pp. 1–20.
[24] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, O. Sander, Arch. Numer. Softw. 4 (100) (2016) 13–29.
[25] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, K.S. Stanley, ACM Trans. Math. Software 31 (3) (2005) 397–423.
[26] A.J.C. Crespo, J.M. Domínguez, B.D. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro, O. García-Feal, Comput. Phys. Comm. 187 (2015) 204–216.
[27] X. Guo, S.J. Lind, B.D. Rogers, P.K. Stansby, M. Ashworth, Proc. 8th International SPHERIC Workshop, 2013.
[28] Z. Ji, L. Fu, X. Hu, N. Adams, Proc. SPHERIC'2017, 2017.
[29] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, J. Comput. Chem. 26 (2005) 1781–1802.
[30] S. Plimpton, J. Comput. Phys. 117 (1995) 1–19.
[31] A. Schäfer, D. Fey, European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2008, pp. 285–294.
[32] AMReX web page [cited 2018]. URL https://amrex-codes.github.io/amrex/.
[33] A. Logg, G.N. Wells, ACM Trans. Math. Software 37 (2) (2011) 20, arXiv:1103.6248.
[34] M. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, G.N. Wells, Arch. Numer. Softw. (2015).
[35] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, 2011 Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC), 2011, PP. 1–12. http://dx.doi.org/10.1145/2063384.2063396.
[36] J. Reynders, J. Cummings, M. Tholburn, P. Hinker, S. Atlas, S. Banerjee, M. Srikant, W. Humphrey, S. Karmesin, K. Keahey, in: A. Bode, M. Gerndt, R. Hackenberg, H. Hellwagner (Eds.), Proceedings. First International Workshop on High-Level Programming Models and Supportive Environments, Tech. Univ. Munchen; Res. Centre Julich; Central Inst. Appl. Math.; 10th IEEE Int. Parallel Process. Symposium; IEEE Comput. Soc. Tech. Committee on Parallel Process.; ACM SIGARCH, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1996, pp. 41–49.
[37] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, J. Makino, Publ. Astron. Soc. Japan 68 (4) (2016) 54.
[38] I.F. Sbalzarini, J.H. Walther, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, P. Koumoutsakos, J. Comput. Phys. 215 (2) (2006) 566–588.
[39] O. Awile, O. Demirel, I.F. Sbalzarini, Proc. ICNAAM, Numerical Analysis and Applied Mathematics, International Conference, AIP, 2010, pp. 1313–1316.
[40] O. Awile, M. Mitrović, S. Reboux, I.F. Sbalzarini, Proc. III Intl. Conf. Particle-based Methods (PARTICLES), Stuttgart, Germany, 2013, p. 52.
[41] S. Karol, T. Nett, P. Incardona, N. Khouzami, J. Castrillon, I.F. Sbalzarini, International Conference on Particle-based Methods – Fundamentals and Applications, Hanover, Germany, 2017, pp. 1–12.
[42] J.J. Monaghan, Annu. Rev. Astron. Astrophys. 30 (1992) 543–574.
[43] P. Degond, S. Mas-Gallic, Math. Comp. 53 (188) (1989) 509–525.
[44] J.D. Eldredge, A. Leonard, T. Colonius, J. Comput. Phys. 180 (2002) 686–709.
[45] B. Schrader, S. Reboux, I.F. Sbalzarini, J. Comput. Phys. 229 (2010) 4159–4182.
[46] J. Barnes, P. Hut, Nature 324 (1986) 446–449.
[47] L. Greengard, V. Rokhlin, J. Comput. Phys. 73 (1987) 325–348.
[48] O. Awile, F. Büyükkeçeci, S. Reboux, I.F. Sbalzarini, Comput. Phys. Comm. 183 (2012) 1073–1081.
[49] L. Verlet, Phys. Rev. 159 (1) (1967) 98–103.
[50] R.W. Hockney, J.W. Eastwood, Computer Simulation using Particles, Institute of Physics Publishing, 1988.
[51] P.P. Ewald, Ann. Phys. (1921).
[52] G.-H. Cottet, P. Koumoutsakos, Vortex Methods – Theory and Practice, Cambridge University Press, New York, 2000.
[53] G.-H. Cottet, J.-M. Etancelin, F. Pérignon, C. Picard, ESAIM Math. Model. Numer. Anal. 48 (4) (2014) 1029–1060.
[54] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSC web page, 2016, URL http://www.mcs.anl.gov/petsc.
[55] G. Guennebaud, B. Jacob, et al., Eigen v3, 2010, http://eigen.tuxfamily.org.
[56] F. Pellegrini, J. Roman, Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1996, Springer-Verlag, London, UK, UK, 1996, pp. 493–498.
[57] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, L.A. Riesen, Proc. of 21st International Parallel and Distributed Processing Symposium, IPDPS'07, IEEE, 2007, pp. 1–11.
[58] T. Hoefler, C. Siebert, A. Lumsdaine, Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10, ACM, 2010, pp. 159–168.
[59] M. Furuichi, D. Nishiura, Comput. Phys. Comm. 219 (2017) 135–148.
[60] S. Tsuzuki, T. Aoki, Proc. 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA'16, IEEE, IEEE Press, Piscataway, NJ, USA, 2016, pp. 1–8.
[61] O. Demirel, I. Smal, W.J. Niessen, E. Meijering, I.F. Sbalzarini, Proc. ICASSP, IEEE Intl. Conf. Acoustics, Speech, and Signal Processing, IEEE, IEEE, Florence, Italy, 2014, pp. 1635–1639.
[62] B. Moon, J. Saltz, Proceedings of the IEEE Scalable High-Performance Computing Conference, IEEE, 1994, pp. 176–183.
[63] Hdf5 web page [cited 2018]. URL http://www.hdfgroup.org/HDF5/.
[64] L.B. Schroeder Will, Martin Ken, The Visualization Toolkit, fourth ed., Kitware, 2006.
[65] A. Utkarsh, The ParaView Guide: A Parallel Visualization Application, ACM Press, 2015.
[66] X.Y. Hu, N.A. Adams, J. Comput. Phys. 213 (2) (2006) 844–861, http://dx.doi.org/10.1016/j.jcp.2005.09.001, URL http://www.sciencedirect.com/science/article/pii/S0021999105004195.
[67] S. Adami, X.Y. Hu, N.A. Adams, J. Comput. Phys. 231 (21) (2012) 7057–7075, http://dx.doi.org/10.1016/j.jcp.2012.05.005, URL http://www.sciencedirect.com/science/article/pii/S002199911200229X.
[68] A.M. Turing, Phil. Trans. R. Soc. London B 237 (1952) 37–72.
[69] P. Gray, S.K. Scott, Chem. Eng. Sci. 38 (1) (1983) 29–43.
[70] P. Gray, S.K. Scott, Chem. Eng. Sci. 39 (6) (1984) 1087–1097.
[71] P. Gray, S.K. Scott, J. Phys. Chem. 89 (1) (1985) 22–32.
[72] J. Lee, A. Ishihara, J.A. Theriot, K. Jacobson, Nature 362 (1993) 167–171.

[73] J.E. Pearson, Science 261 (5118) (1993) 189–192.
[74] J.J. Monaghan, J. Comput. Phys. 60 (1985) 253–262.
[75] M. Bergdorf, P. Koumoutsakos, A. Leonard, J. Fluid Mech. 581 (2007) 495–505, URL http://www.cse-lab.ethz.ch/wp-content/papercite-data/pdf/bergdorf2007a.pdf.
[76] J.H. Walther, I.F. Sbalzarini, Eng. Comput. 26 (6) (2009) 688–697.
[77] L.E. Silbert, D. Ertaş, G.S. Grest, T.C. Halsey, D. Levine, S.J. Plimpton, Phys. Rev. E 64 (2001) 051302.
[78] K.G. Wilson, Phys. Rev. D 10 (1974) 2445–2459, http://dx.doi.org/10.1103/PhysRevD.10.2445.
[79] C. Bonati, G. Cossu, M. D'Elia, P. Incardona, Comput. Phys. Comm. 183 (4) (2012) 853–863, http://dx.doi.org/10.1016/j.cpc.2011.12.011, URL http://www.sciencedirect.com/science/article/pii/S0010465511003997.
[80] Y. Afshar, I.F. Sbalzarini, PLoS One 11 (4) (2016) e0152528, http://dx.doi.org/10.1371/journal.pone.0152528.
[81] C.L. Müller, I.F. Sbalzarini, Proc. IEEE Congress on Evolutionary Computation (CEC), Barcelona, Spain, 2010, pp. 2594–2601.
[82] N. Hansen, S.D. Muller, P. Koumoutsakos, Evol. Comput. 11 (1) (2003) 1–18.
[83] N. Hansen, The CMA Evolution Strategy: A Tutorial, 2007, URL http://www.inf.ethz.ch/personal/hansenn/cmatutorial.pdf.
[84] C.L. Müller, B. Baumgartner, I.F. Sbalzarini, Proc. IEEE Congress on Evolutionary Computation (CEC), IEEE, Trondheim, Norway, 2009, pp. 2685–2692.
[85] C.L. Müller, B. Baumgartner, G. Ofenbeck, B. Schrader, I.F. Sbalzarini, Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, ACM, New York, NY, USA, 2009, pp. 1411–1418.
[86] C.L. Müller, I.F. Sbalzarini, Proc. EvoStar, in: Leture Notes in Computer Science, vol. 6624, Springer, Torino, Italy, 2011, pp. 294–303.
[87] S. Karol, T. Nett, J. Castrillon, I.F. Sbalzarini, ACM Trans. Math. Software 44 (3) (2018) 34.
[88] S. Reboux, B. Schrader, I.F. Sbalzarini, J. Comput. Phys. 231 (2012) 3623–3646.
[89] B.L. Cheeseman, U. Günther, K. Gonciarz, M. Susik, I.F. Sbalzarini, Nat. Commun. 9 (2018) 5160.
[90] F. Büyükkeçeci, O. Awile, I.F. Sbalzarini, Parallel Comput. 39 (2013) 94–111.
[91] J. Cardinale, G. Paul, I.F. Sbalzarini, IEEE Trans. Image Process. 21 (8) (2012) 3531–3545.