# The OpenPME Problem Solving Environment for Numerical Simulations

Nesrine Khouzami[1], Lars Schütze[1], Pietro Incardona[1,2,3], Landfried Kraatz[1], Tina Subic[1,2,3], Jeronimo Castrillon[1], and Ivo F. Sbalzarini[1,2,3,4]

[1] Technische Universität Dresden, Faculty of Computer Science, Dresden, Germany
firstname.lastname@tu-dresden.de
[2] Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
[3] Center for Systems Biology Dresden, Dresden, Germany
[4] Cluster of Excellence Physics of Life, TU Dresden, Dresden, Germany

**Abstract.** We introduce OpenPME, the Open Particle-Mesh Environment, a problem solving environment that provides a Domain Specific Language (DSL) for numerical simulations in scientific computing. It is built atop a domain metamodel that is general enough to cover the main types of numerical simulations: simulations using particles, meshes, and hybrid combinations of particles and meshes. Using model-to-model transformations, OpenPME generates code against the state-of-the-art C++ parallel computing library OpenFPM. This effectively lowers the programming barrier and enables users to implement scalable simulation codes for high-performance computing (HPC) systems using high-level abstractions. Plenty of recent research has shown that higher-level abstractions and problem solving environments are well suited to alleviate low-level implementation overhead. We demonstrate this for OpenPME and its compiler on three different test cases—particle-based, mesh-based, and hybrid particle-mesh—showing up to 7-fold reduction in the number of lines of code compared to a direct OpenFPM implementation in C++.

**Keywords:** Domain Specific Compiler · Particle-Mesh Methods.

## 1 Introduction

Computer simulations are the third pillar of science, alongside theory and experiment. Scientists increasingly rely on simulated models to investigate scales that are not experimentally accessible and nonlinearities that are not theoretically treatable. However, the complexity of modern high-performance computing (HPC) systems, and the complexity of HPC programming models increasingly limit implementation efficiency and performance portability. Alleviating this has therefore become a key research focus for HPC [25]. Examples are problem solving environments (PSE) [6], which reduce the programming barrier by providing higher-level domain-specific abstractions. Since higher-level abstractions enable more powerful compiler transformations, this often leads to performance improvements over hand-written code [21]. A PSE typically consists of a Domain Specific Language (DSL) and an Integrated Development Environments

(IDE) [6]. In scientific computing, such approaches have successfully been proposed for array programming [28,29], finite-element simulations [9,16,18,22,23], and for tensor expressions in numerical simulations [3,24].

Most numerical simulations can be expressed in terms of particles, meshes, or a combination thereof. Meshes are often used to discretize continuous fields, e.g., for finite-differences or finite volumes. Particles can be used to represent either discrete objects, like atoms in a molecule or cars in road traffic, or as arbitrarily discretization points for continuous fields. The classic Particle-In-Cell (PIC) codes in plasma physics [10] are an example of a combination of particles and meshes. Existing DSLs for particle-mesh simulations include PPML [2], PPME [14], and FDPS [12]. Of these, FDPS and PPME are limited to particles, with FDPS focusing on N-body simulations. PPML supports both particles and meshes, but relies on the discontinued Fortran library PPM [27]. To the best of our knowledge, there exists no DSL that supports particles, meshes, and hybrid particle-mesh codes on a modern and actively maintained platform library.

This paper presents a DSL and IDE for particle, mesh, and hybrid particle-mesh simulations on parallel computers, the Open Particle-Mesh Environment (OpenPME). OpenPME is based on the Open Framework for Particles and Meshes (OpenFPM) [11], a recent and actively maintained open-source C++ library. Compared to writing C++ code for OpenFPM, OpenPME reduces the implementation effort and lowers the entry barrier for users. It hides the distributed-memory constructs of OpenFPM and provides error messages that are easier to understand than the template-engine errors of the C++ compiler.

Our DSL rests on a metamodel that captures all five types of simulations: particles (continuous and discrete), mesh (continuous), and two types of hybrid simulations. The OpenPME compiler implements a staged compilation process, where high-level DSL constructs are reduced to an intermediate metamodel. The metamodel provides a language-independent representation and enables model-to-model transformations. This allows the OpenPME compiler to automatically inject OpenFPM communication operations for distributed-memory programs, freeing the user of having to write them. In a second step, the metamodel is translated to OpenFPM C++ code. We showcase the expected productivity increase and DSL design of OpenPME using three real-world examples that highlight the seamless support for hybrid particle-mesh simulations and the automatic insertion of data-movement and parallel communication primitives by the compiler.

## 2 Background and Motivation

We introduce the general structure of particle-mesh simulations, describe the OpenFPM library, and detail our motivation for developing OpenPME.

### 2.1 Hybrid particle-mesh simulations

Particle-mesh methods provide a universal framework for numerical simulations in scientific computing. They can be used to simulate both discrete and continuous models, either deterministically or stochastically [20,26]. When simulating

discrete models, particles directly represent entities in the model. In continuous models, particles correspond to mathematical discretization points. Particles are point objects that have a position and a list of properties. They can *interact* with other particles through pairwise interactions, and then *evolve* as a consequence of the interactions, i.e., particles change their position and/or their properties. These interactions can be deterministic or probabilistic. Hybrid particle-mesh methods are used to obtain more efficient simulations or to simulate multi-physics models. Particle-to-mesh and mesh-to-particle interpolation allows transferring data between the two representations [10].

### 2.2 The OpenFPM library

OpenFPM [11] is an open-source C++ template library for implementing scalable parallel particle-mesh simulations on multi-CPU and multi-GPU computer hardware. It provides multiple layers of abstraction: based on memory allocators and memory-layouting abstractions, OpenFPM implements single-core data structures. Using data-decomposition and network communication abstractions, these are then transformed to multi-core and distributed-memory data structures. Finally, a library of frequently used numerical solvers is implemented using these data structures. A profiling interface and transparent in-situ visualization of simulation results [8] complete the library.

In OpenFPM, particles can carry any composite container of C++ objects as properties, and simulations can be performed in arbitrary-dimensional domains. It guarantees transparent memory layout conversion (e.g., between CPU and GPU) and run-time dynamic load-balancing to distribute data evenly and adapt to changes in local mesh resolution or particle density. OpenFPM includes checkpointing, parallel file I/O, and communications abstractions, including the `ghost_get` operator to transparently communicate boundary data between processors in a domain decomposition.

### 2.3 Motivation

OpenFPM heavily relies on C++ templates and template meta-programming to achieve its flexibility and performance. This renders the source code more complex, leading to programming mistakes and cryptic error messages from the C++ compiler. A typical example of OpenFPM code is shown in Fig. 1 for a simulation of the three-dimensional Navier-Stokes equation of fluid mechanics (only one dimension shown). The more concise OpenPME three-dimensional representation in Fig. 9, point ④ will be discussed in Sec. 4.3. Not only is the C++ more error-prone, but also makes it impossible for the compiler to detect semantic errors that arise from accessing objects in the wrong place or misplaced operators in the formula. User also must manually place `ghost_get` communication operations to account for data exchange on a parallel computer. These calls must be in the right place and include the correct properties to be communicated, which is not always obvious for users. Extraneous `ghost_get` calls reduce the scalability and performance of the simulation.

```
g_dwp.template get<rhs>(key)[x]=
  fac1*(g_vort.template get<vorticity>(key.move(x,1))[x]+
  g_vort.template get<vorticity>(key.move(x,-1))[x])+
  fac2*(g_vort.template get<vorticity>(key.move(y,1))[x]+
  g_vort.template get<vorticity>(key.move(y,-1))[x])+
  fac3*(g_vort.template get<vorticity>(key.move(z,1))[x]+
  g_vort.template get<vorticity>(key.move(z,-1))[x])-
  2.0f*(fac1+fac2+fac3)*
  g_vort.template get<vorticity>(key)[x]+
  fac4*g_vort.template get<vorticity>(key)[x]*
  (g_vel.template get<velocity>(key.move(x,1))[x]-
  g_vel.template get<velocity>(key.move(x,-1))[x])+
  fac5*g_vort.template get<vorticity>(key)[y]*
  (g_vel.template get<velocity>(key.move(y,1))[x]-
  g_vel.template get<velocity>(key.move(y,-1))[x])+
  fac6*g_vort.template get<vorticity>(key)[z]*
  (g_vel.template get<velocity>(key.move(z,1))[x]-
  g_vel.template get<velocity>(key.move(z,-1))[x]);
```

**Fig. 1.** OpenFPM C++ code snippet to calculate the $x$-component of the three-dimensional Navier-Stokes equation.

OpenPME's goal is to allow computational scientists to write efficient simulations with domain-specific abstractions and error messages. OpenPME's abstractions enable high-level optimizations that would otherwise require complex and brittle analysis in a custom C++ compiler pass. We demonstrate this by automatically placing `ghost_get` operations.

## 3 OpenPME Design and Implementation

We detail the design and implementation of OpenPME based on a metamodel that captures all (hybrid) particle-mesh methods.

### 3.1 Design overview

OpenPME introduces an intermediate layer between the user's simulation application and the OpenFPM C++ library, as illustrated in Fig. 2. It is based on two metamodels representing particle-mesh simulations and the C++ code using OpenFPM. Model-to-model transformations identify syntactic elements and semantic relations that enable, e.g., expression rewriting or automatic insertion of communication statements. The generated code executes on any platform supported by OpenFPM. This includes multi-core CPUs, GPUs, distributed-memory CPU clusters, and multi-GPU clusters. Parallel efficiency and scalability are inherited from OpenFPM [11]. OpenPME is implemented in Jetbrains MPS.[5]

The simulation metamodel is implemented as a stack of sub-languages (see Fig. 2). The use of sub-languages enables separation of concerns and keeps the implementation modular, extensible, and maintainable. Lower levels provide foundational elements of the language that are used to build more complex features higher in the stack.

[5] Jetbrains Meta Programming System (MPS) https://www.jetbrains.com/mps/

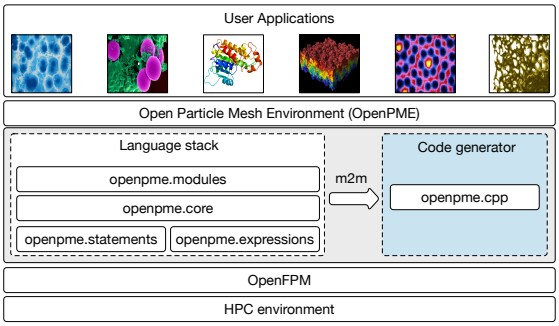


**Fig. 2.** Modular architecture of OpenPME. An intermediate layer between HPC application domains interfacing OpenFPM for HPC architecture systems.

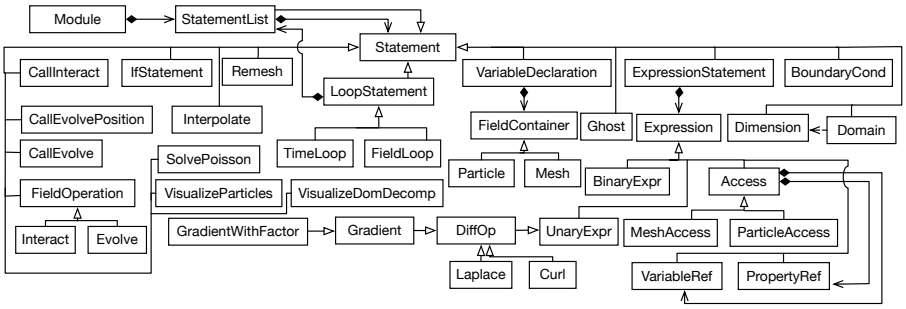


**Fig. 3.** Excerpt from the metamodel to describe particle-mesh simulations.

The OpenFPM C++ metamodel captures the domain of imperative, object-oriented programs. It is implemented in the `openpme.cpp` language (see Fig. 2). This model is expressive enough to generate the complete C++ code of a simulation using OpenFPM. We also use this model as an intermediate representation when lowering simulation code using model-to-model transformation.

### 3.2 Metamodel for particle-mesh simulations

The metamodel for particle-mesh simulations builds on our prior work in [14]. It supports three types of simulations: particle-only, mesh-only, and hybrid particle-mesh of both continuous and discrete models. Fig. 3 shows an excerpt of the model as UML diagram. An OpenPME program consists of modules, each with a sequence of statements. A statement can declare variables or a boundary condition, or define the simulation domain and dimensionality (see Fig. 3). A `TimeLoop` statement supports different time-discretization methods as first-class concepts in the model. The model supports basic data types found in most programming languages, as well as domain-specific types such as `Particle` or `Mesh`.

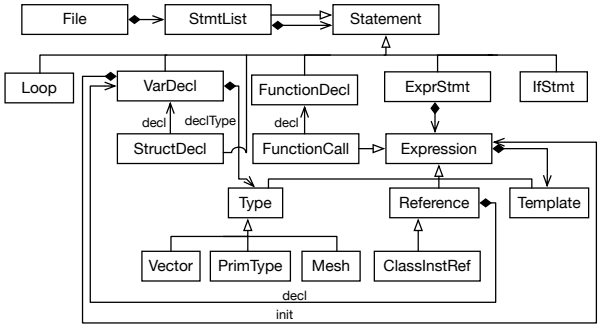


**Fig. 4.** Excerpt from the metamodel to describe OpenFPM C++ programs.

`Expression` statements are binary or unary arithmetic and logical expressions that access constants or particle and mesh properties. There are first-class concepts to express computations on a complete field, such as differential operators (`DiffOp`), e.g., a gradient. Domain-specific types allow specifying `Interpolate` statement to transform data from particles to meshes and vice-versa.

### 3.3 OpenPME DSL syntax

The syntax of the OpenPME DSL allows for both imperative and declarative programs. Its declarative nature allows hiding loops and conditional statements, e.g., when specifying particle interaction and evolution (see Fig. 7, lines 25-26). Another design principle is to remain close to the mathematical notation of the equations to be simulated. This can be seen in Fig. 8, where the Partial Differential Equation (PDE) to be simulated is explicitly expressed. Properties of both particles and meshes can be accessed in bulk which frees the user from having to iterate through a loop (see Fig. 9). This hides OpenFPM's iterator classes, where using templates is mandatory to reach high performance.

### 3.4 Metamodel for OpenFPM C++

The second metamodel captures features of imperative, object-oriented C++ code that compiles against the OpenFPM library. Model-to-model transformations from the metamodel for simulations allows detecting and avoiding programming mistakes. An excerpt of the model is shown in Fig. 4 as a UML diagram.

This metamodel has a single root concept `File` containing a list of statements. Statements can be, e.g., the declaration of a variable (`VarDecl`) of primitive type or of any OpenFPM class, such as distributed vectors and meshes. Computations are expressed through expression statements (`ExprStmt`), with most common ones being arithmetic operations, references to variables, or function calls. The model captures function calls to the OpenFPM API in a generic way, which is only specified during code generation. This allows to adapt to

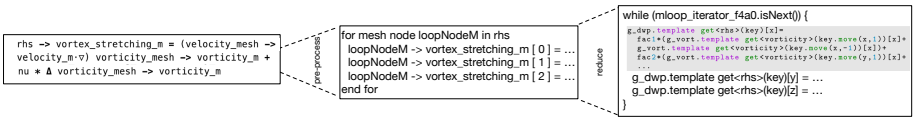


**Fig. 5.** Code generation for the 3D Navier-Stokes equation

future changes in the OpenFPM API. Template parameters can be inferred by semantic analysis during code generation.

### 3.5 Model-to-model transformation and code generation

Code generation in OpenPME is implemented by multiple model-to-model transformations refining and optimizing the program, a model-to-model transformation lowering the program to our intermediate representation, and a final text-generation stage that produces C++ code. Models in MPS are directed graphs that have distinct spanning trees, which correspond to abstract syntax trees. A model-to-model transformation maps an input graph to an output graph, where the output graph may consist of elements from different metamodels.

To illustrate how these transformations can be chained, consider ④ in Fig. 9, which computes a complete field stored in a mesh. Because the computation happens on the whole field, the expression is first refined to be contained in a `FieldLoop` (see Fig. 3) to yield a point-wise calculation over all mesh nodes. In a subsequent transformation, the assignment to the property is replaced by an assignment to each dimension of the property, rewriting the expression on the right-hand side to consider only the respective dimension. While lowering the expression into its intermediate representation, the differential operator (here $\nabla\times$) is replaced by finite differences. In the last transformation, the intermediate program is traversed, generating C++ code for each element. For instance, the access to properties of the mesh is replaced by templated C++ code where each property is represented by its template argument (see Fig. 5).

Communication between processors is hidden in the OpenPME DSL and automatically inserted during code generation. For example, particle positions can only change due to particle interactions. Interaction happens, by design, in particle loops. Therefore, whenever the position of a particle is assigned, an OpenFPM communication statement can be inserted after the respective loop.

## 4 Evaluation

We evaluate OpenPME in three use-cases representative of different types of simulations: First, we consider a molecular dynamics simulation of a Lennard-Jones gas, which is representative of a particles-only simulation. Second, we use OpenPME to simulate the Gray-Scott reaction-diffusion model in a representative mesh-only simulation. Third, we simulate incompressible fluid dynamics

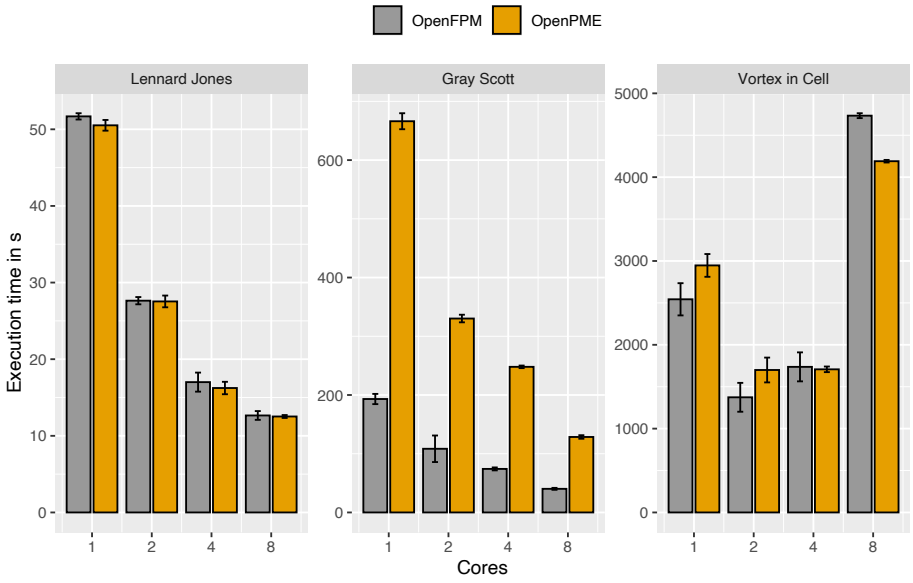


**Fig. 6.** Execution time of simulations written directly in OpenFPM versus those generated by OpenPME for the three uses-cases with a varying amount of cores.

by solving the Navier-Stokes equations using a Vortex Method representative of hybrid particle-mesh simulations. We compare how many lines of code (LOC) were necessary to implement each use-case. The experiments were repeated 10 times and scaled from 1 to 8 cores on a HPC cluster. The overall result is shown in Fig. 6 where each is discussed in the following sections.

### 4.1 Lennard-Jones: Particles-only simulation of a discrete model

We consider a molecular dynamics simulation in three dimensions (3D) where particles represent atoms in a gas that interact according to the pairwise Lennard-Jones [13], exerting a force on both atoms as a function of their distance. Fig. 7 shows an excerpt of the corresponding OpenPME program. The simulation process is encapsulated in the methods `interact` and `evolve` defined on the particles sets (lines 14-18). The simulation uses Verlet time stepping, where the particle velocities and positions are advanced alternatively with the force recomputed in-between. Calls to `interact` and `evolve` must specifying the property to use (lines 25-26). The communications required when particles move across processor boundaries, as well as to update the ghost layers with the new velocities from the neighboring processors, are automatically added by the compiler and do not need to be specified in the OpenPME program. Finally, the program computes the potential energy of the system using an imperative iteration over particles (line 29). At the end, particle data is stored in a file and visualized according to

```
1  Module Lennard Jones
2   initialization
3    dimension: 3
4    domain_size: box((0.0,0.0,0.0),(1.0,1.0,1.0))
5    boundary_conditions: periodic
6    ...
7   simulation
8    type of simulation: particle-based
9    Particle sets:
10    name particles
11    properties
12     velocity d:3
13     force d:3
14    Define interact in particles with self as p_force, neighbor as q_force
15     p_force->force += 24.0 * 2.0 * sigma/r^7 - sigma6/r^4 *
16      diff(p_force,q_force)
17    Define evolve in particles with self as p_velocity
18     p_velocity->velocity += 0.5 * dt * p_velocity->force
19    ...
20   Body:
21    ...
22    interact force, particles
23    time loop start:0 stop: 10000
24     ...
25     evolve velocity, particles
26     interact force, particles
27     ...
28     for particle p_energy in particles
29       ...
30       E += 2.0 * sigma12/rn_e^6 - sigma6/rn_e^3 - shift
31     end for
32    end timeloop
33  visualization
34    output file: "particles"
35 end module
```

**Fig. 7.** OpenPME program for Lennard-Jones molecular dynamics simulation.

the specifications in the visualization phase (line 34). As can be seen in Fig. 6 both approaches perform similar. The implementation with OpenPME requires 57 LOC instead of 151 LOC using OpenFPM.

### 4.2 Gray-Scott: Mesh-only simulation of a continuous model

To illustrate the use of OpenPME for a mesh-only simulation of a continuous model, we simulate the Gray-Scott reaction-diffusion model given by the PDEs [7]

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

that govern the space-time evolution of the concentration fields $u(\boldsymbol{x}, t)$ and $v(\boldsymbol{x}, t)$ of two chemicals $U$ and $V$. $D_u$ and $D_v$ are the diffusion constants of the two chemicals, and $F$ and $k$ are the chemical reaction rates. This model is nonlinear and shows self-organized emergence of sustained spatiotemporal patterns. The OpenPME program in Fig. 8 shows a 3D model on a regular Cartesian mesh of $128 \times 128 \times 128$ nodes, discretizing the two scalar ($d = 1$) fields $U$ and $V$

```
1  Module Gray Scott
2   initialization
3   ...
4   simulation
5    type of simulation: mesh-based
6    Meshes:
7     name Old
8     properties
9      u d:1
10     v d:1
11    size{128,128,128}
12    name New
13    properties
14     u d:1
15     v d:1
16    size{128,128,128}
17 Body:
18  ...
19  Load Old from "init_mesh.hdf5"
20  time loop start: 0 stop: 5000
21   New->u = Old->u + dt * du *
22        △Old->u - Old->u *
23        Old->v^2 + F *
24        1.0 - Old->u
25   New->v = Old->v + dt * dv *
26        △Old->v + Old->u *
27        Old->v^2 - F + k * Old->v
28   copy from New to Old
29   Resync Ghost Old<u,v>
30  end timeloop
31  ...
32 end module
```

**Fig. 8.** OpenPME program for Gray-Scott simulation on a mesh.

(lines 6-16). Time stepping is done using the explicit Euler method. The Laplace operator $\Delta$ over the continuous fields $u$ and $v$ is discretized using central finite differences. The corresponding OpenPME expressions closely mirror the terms and notation in the model PDEs. The DSL compiler can automatically insert OpenFPM communication operators and distributed OpenFPM data structures. In this case, two identically distributed meshes are required in order to read and write simultaneously. We are on average 3.25× slower than OpenFPM original code. The reason is that consecutive bulk access of mesh properties results in independent mesh loops that are merged in the hand-written OpenFPM C++ program which requires more communication and computation. This is one of many optimizations for the code generation we are targeting in future work. For this simulation, OpenPME requires 40 LOC instead of 100 LOC in OpenFPM.

### 4.3 Vortex-in-Cell: Hybrid particle-mesh simulation of a continuous model

We simulate incompressible fluid dynamics as governed by the 3D Navier-Stokes equations in vorticity form using a Vortex Method simulation [4]. This is a hybrid particle-mesh method where particles represent fluid elements that carry vorticity, and a mesh is used to solve the Poisson equation for the flow velocity field, which in turn moves the vorticity particles.

The OpenPME program in Fig. 9 starts by initializing the flow field on a mesh and creating particles at the mesh nodes using the `remesh` statement ①. After initializing the particles, the simulation enters the time loop with a fixed time step `dt = 0.0125`. In each time step, the vorticity is interpolated from particles to mesh in order to solve the Poisson equation for the flow velocity on the mesh ②. Computing and storing the right-hand side of the Poisson equation requires a second mesh `phi` ③. Next, the vortex stretching term is computed on the mesh, where the Laplace operator $\Delta$ is automatically discretized by central finite differences ④. As all properties were calculated on meshes, we have to interpolate back to the particles ⑤. This is used to update particle positions ⑥.

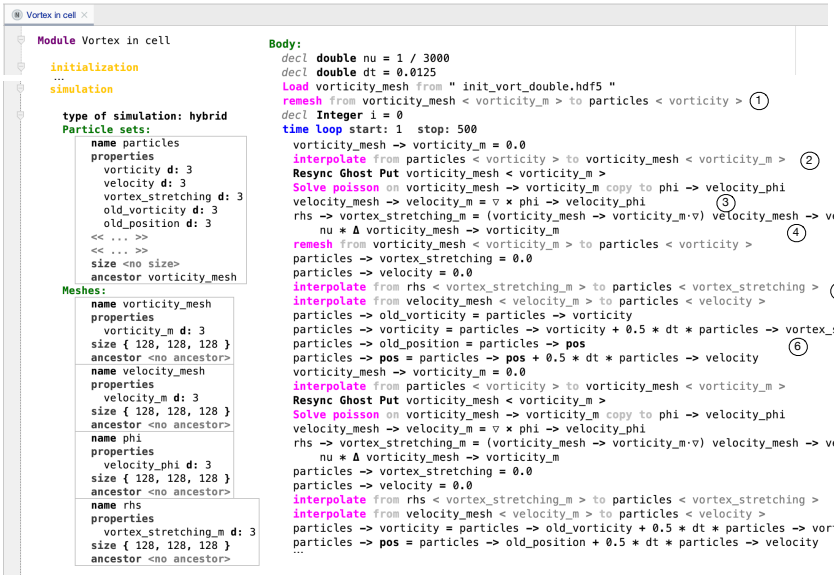


**Fig. 9.** OpenPME program for a Vortex Method simulation of incompressible flows.

The generated code performs on average as OpenFPM. For this simulation the parallel efficiency drops the more cores are used [11]. For the measured problem size 2 cores give the best performance. This implementation of a Vortex Method in OpenPME requires 73 LOC instead of 508 LOC in OpenFPM which is a 7-fold reduction.

## 5 Related Work

Many frameworks have been proposed to ease the use of HPC systems for scientific computing and simulations.

Among internal DSLs, Blitz++ [29] is a C++ template library and DSL for defining mesh stencils from a high-level mathematical specification library. Liszt [5] is a portable DSL, embedded into Scala, to implement PDE solvers on meshes, supporting different parallel programming models. Freefem++ [9] is a DSL for finite-element methods allowing users to define analytic and finite-element functions using abstractions like meshes and differential operators.

Other approaches use external DSLs. Examples include the Unified Form Language (UFL) [1] to define PDEs and finite-element simulations. There are several optimizing compilers that generate low-level code from UFL, e.g., the FEniCS Form Compiler (FFC) [15]. The Firedrake project [22] is an extension to FEniCS that adds composing abstractions like parallel loop operations. Users express simulations in a high-level specification translated to an abstract syntax tree by the COFFEE [18] compiler, able to apply suitable low-level optimizations.

Many DSLs use domain-specific optimizations. Examples include PyOP2 [23], a DSL for unstructured-mesh simulations executing numerical kernels in parallel, which enhances FFC to generate optimized kernels from finite-element expressions written in UFL. Devito [17] is a DSL that uses symbolic Python expressions to provide optimized stencil computations. Saiph [19] is an optimizing DSL for computational fluid dynamics, implemented in Scala, with a custom compiler to translate code to an intermediate representation that is subsequently translated to C++. In the CFDlang [24] DSL, the locality of operators is exploited to optimize tensor operations for performance in computational fluid dynamics.

## 6 Conclusions and Future Work

We presented OpenPME, a PSE for particle, mesh, and hybrid particle-mesh simulations on parallel and high-performance computers. It is based on two novel metamodels, covering the domain of particle, mesh, and hybrid particle-mesh simulations, while also covering the imperative object-oriented C++ API of OpenFPM. It supports simulations of all types for both continuous and discrete models. The OpenPME compiler uses a sequence of model-to-model transformations between the two metamodels in order to automatically inject the required communication and synchronization operations for distributed-memory codes, and to translate OpenPME programs to OpenFPM C++ code. This leverages the scalability, GPU support, and performance of OpenFPM to generate efficient HPC applications. The OpenPME language stack is modularly composed of multiple sub-languages, improving maintainability and extensibility. The OpenFPM metamodel is only specified during code generation, hence flexibly adapting to future API changes in OpenFPM.

OpenPME programs can be developed independently at a high level of abstraction, close to the mathematical model. It frees users from having to explicitly deal with the C++ template constructs of OpenFPM, from having to use explicit iterators, and from having to decide on and place communication abstractions in a distributed-memory parallel program. This enables high-level optimizations not otherwise possible, renders simulation programs more compact, easier to read, and more accessible to new users. In the benchmarks presented here the code size of simulations is reduced up to a factor of 7 when implementing them in OpenPME versus directly writing C++ code for OpenFPM. The generated code performs in general as the hand-written C++ code using OpenFPM.

In the future, we plan to further improve OpenPME by adding features to support optimized memory layouts in OpenFPM, like space-filling curves as sub-domain or mesh-node indices in order to improve cache efficiency. The high-level OpenPME syntax cleanly separates the model specification from the choice and parameters of the numerical methods used to simulate the model. This will enable auto-tuning in the compiler to automatically select simulation parameters (e.g., mesh resolution or time step size) and discretization methods to reduce the simulation runtime required to reach a certain target accuracy. A more declarative syntax could allow domain experts to apply manual optimizations

and gives us more control over the generated code. The OpenPME PSE – the DSL, IDE, and compiler – are available as open-source software.[6]

**Acknowledgements.** This work was partly supported by the German Research Foundation (DFG) within the OpenPME project (number 350008342) and by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under funding code 031L0160 (project "SPlaT-DM – computer simulation platform for topology-driven morphogenesis").

## References

1. Alnæs, M.S., Logg, A., Ølgaard, K.B., Rognes, M.E., Wells, G.N.: Unified form language: A domain-specific language for weak formulations of partial differential equations. ACM Trans. Math. Softw. **40**(2), 1–37 (2014)
2. Awile, O., Mitrovic, M., Reboux, S., Sbalzarini, I.F.: A domain-specific programming language for particle simulations on distributed-memory parallel computers. In: Proc. III Intl. Conf. Particle-Based Methods (PARTICLES). Stuttgart, Germany (2013)
3. Baumgartner, G., Auer, A., Bernholdt, D.E., Bibireata, A., Choppella, V., Cociorva, D., Gao, X., Harrison, R.J., Hirata, S., Krishnamoorthy, S., et al.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proc. IEEE **93**(2), 276–292 (2005)
4. Cottet, G.H., Koumoutsakos, P.: Vortex Methods – Theory and Practice. Cambridge University Press, New York (2000)
5. DeVito, Z., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A.: Liszt: A domain specific language for building portable mesh-based PDE solvers. In: Proc. SC'11. p. 1. ACM Press (2011)
6. Gallopoulos, E., Houstis, E., Rice, J.R.: Computer as thinker/doer: problem-solving environments for computational science. IEEE Comput. Sci. Engrg. **1**(2), 11–23 (1994)
7. Gray, P., Scott, S.K.: Sustained oscillations and other exotic patterns of behavior in isothermal reactions. J. Phys. Chem. **89**(1), 22–32 (1985)
8. Gupta, A., Incardona, P., Aydin, A.D., Gumhold, S., Günther, U., Sbalzarini, I.F.: An architecture for interactive in situ visualization and its transparent implementation in OpenFPM. In: Proc. Workshop In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV), SC'20. pp. 20–26. ACM (2020)
9. Hecht, F.: New development in Freefem++. J. Numer. Math. **20**(3-4) (2012)
10. Hockney, R.W., Eastwood, J.W.: Computer Simulation using Particles. Institute of Physics Publishing (1988)
11. Incardona, P., Leo, A., Zaluzhnyi, Y., Ramaswamy, R., Sbalzarini, I.F.: OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. Comput. Phys. Commun. **241**, 155–177 (2019)
12. Iwasawa, M., Tanikawa, A., Hosono, N., Nitadori, K., Muranushi, T., Makino, J.: Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. Publ. Astron. Soc. Japan **68**(4) (2016)

[6] OpenPME is available under https://github.com/Nesrinekh/OpenPME

13. Jones, J.E.: On the determination of molecular fields. II. from the equation of state of a gas. Proc. Roy. Soc. London A **106**(738), pp. 463–477 (1924)
14. Karol, S., Nett, T., Castrillon, J., Sbalzarini, I.F.: A domain-specific language and editor for parallel particle methods. ACM Trans. Math. Softw. **44**(3), 34:1–34:32 (2018)
15. Logg, A., Ølgaard, K.B., Rognes, M.E., Wells, G.N.: FFC: the FEniCS form compiler. In: Automated Solution of Differential Equations by the Finite Element Method, vol. 84, pp. 227–238. Springer Berlin Heidelberg (2012)
16. Luporini, F., Ham, D.A., Kelly, P.H.J.: An algorithm for the optimization of finite element integration loops. ACM Trans. Math. Softw. **44**(1), 1–26 (2017)
17. Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hückelheim, J., Yount, C., Kelly, P.H., Herrmann, F.J., Gorman, G.J.: Architecture and performance of Devito, a system for automated stencil computation. ACM Trans. Math. Softw. **46**(1), 1–28 (2020)
18. Luporini, F., Varbanescu, A.L., Rathgeber, F., Bercea, G.T., Ramanujam, J., Ham, D.A., Kelly, P.H.J.: Cross-loop optimization of arithmetic intensity for finite element local assembly. ACM Trans. Arch. & Code Opt. **11**(4), 1–25 (2015)
19. Macià, S., Mateo, S., Martínez-Ferrer, P.J., Beltran, V., Mira, D., Ayguadé, E.: Saiph: Towards a DSL for high-performance computational fluid dynamics. In: Proc. Real World DSL Workshop 2018. pp. 6:1–6:10. ACM, New York, NY, USA (2018)
20. Oñate, E., Owen, D.R.J.: Particle-based methods: fundamentals and applications. Springer (2013)
21. Püschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proc. IEEE **93**(2), 232 –275 (2005)
22. Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., McRae, A.T.T., Bercea, G.T., Markall, G.R., Kelly, P.H.J.: Firedrake: Automating the finite element method by composing abstractions. ACM Trans. Math. Softw. **43**(3), 1–27 (2016)
23. Rathgeber, F., Markall, G.R., Mitchell, L., Loriant, N., Ham, D.A., Bertolli, C., Kelly, P.H.J.: Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. SC Companion pp. 1116–1123 (2012)
24. Rink, N.A., Huismann, I., Susungi, A., Castrillon, J., Stiller, J., Fröhlich, J., Tadonki, C.: CFDlang: High-level code generation for high-order methods in fluid dynamics. In: Proc. 3rd Intl. Workshop on Real World DSLs. pp. 5:1–5:10. ACM, New York, NY, USA (2018)
25. Sbalzarini, I.F.: Abstractions and middleware for petascale computing and beyond. Intl. J. Distr. Systems & Techn. **1**(2), 40–56 (2010)
26. Sbalzarini, I.F.: Modeling and simulation of biological systems from image data. Bioessays **35**(5), 482–490 (2013)
27. Sbalzarini, I.F., Walther, J.H., Bergdorf, M., Hieber, S.E., Kotsalis, E.M., Koumoutsakos, P.: PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems. J. Comput. Phys. **215**(2), 566–588 (2006)
28. Šinkarovs, A., Bernecky, R., Vießmann, H.N., Scholz, S.B.: A rosetta stone for array languages. In: Proc. 5th ACM SIGPLAN Intl. Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 1–10 (2018)
29. Veldhuizen, T.L.: Blitz++: The library that thinks it is a compiler. In: Advances in Software Tools for Scientific Computing, vol. 10, pp. 57–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)