Journal of
open research software

Integrating Odeint Time Stepping into OpenFPM for Distributed and GPU Accelerated Numerical Solvers

SOFTWARE METAPAPER

]u[ubiquity press

ABHINAV SINGH
LANDFRIED KRAATZ
SERHII YASKOVETS
PIETRO INCARDONA
IVO F. SBALZARINI

*Author affiliations can be found in the back matter of this article

ABSTRACT

We present a scientific numerical software for multi-stage, multi-step, and adaptive explicit time integration on distributed-memory parallel computers and on Graphics Processing Units (GPUs). Our implementation integrates the Odeint library from Boost with the OpenFPM framework for scientific computing, enabling compact and scalable numerical simulation codes. Specifically, we extend the Odeint data types to OpenFPM's metaprogramming system. This makes the time-integration methods from Odeint available in a concise template-expression language for numerical codes distributed and parallelized using OpenFPM. We benchmark the software for exponential and sigmoidal dynamics and present application examples to the 3D Gray-Scott reaction-diffusion problem and the "dam break" problem from fluid mechanics. We find a strong-scaling efficiency of 80% on up to 512 CPU cores and a five-fold speedup on a single GPU.

CORRESPONDING AUTHOR:
Ivo F. Sbalzarini
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany; Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany; Center for Systems Biology Dresden, Dresden, Germany; Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden/Leipzig, Germany
sbalzarini@mpi-cbg.de

KEYWORDS:
numerical integration; scientific computing; parallel computing; differential equations; GPU acceleration; distributed computing; time integration

TO CITE THIS ARTICLE:
Singh A, Kraatz L, Yaskovets S, Incardona P, Sbalzarini IF 2026 Integrating Odeint Time Stepping into OpenFPM for Distributed and GPU Accelerated Numerical Solvers. *Journal of Open Research Software,* 14: 15. DOI: https://doi.org/10.5334/jors.598

(1) OVERVIEW

INTRODUCTION

High-performance computers are commonly used for computer simulations of dynamical models and for numerically solving mathematical equations describing the dynamics of a system or process. This is done using *time-integration* or *time-stepping* methods, which are numerical algorithms that approximate the trajectory of the state of a dynamical system to a future point in time. Time integration can be done explicitly or implicitly. An implicit integrator solves a system of equations for the state at a future time point. An explicit integrator evolves the state in time by iteratively taking (sufficiently small) time steps. This successively advances the state $\mathbf{u}(t)$ of the simulation from a time t to a time $t + \delta t$, $\delta t > 0$. In distributed-memory high-performance computing, explicit solvers are usually preferred, since they avoid the global communication associated with solving an implicit equation system. They do, however, still require local communication between neighboring processes. For more advanced explicit time integrators—such as multi-stage and multi-step schemes, or schemes with dynamically adjusted step size—implementing this local communication in a scalable and portable way requires great care.

Here, we provide a generic and performance-portable implementation of explicit time integration on parallel computer architectures. Our implementation hides the internal communication of the time integrator from the application logic of the simulation code, achieving separation of concerns. This speeds up development times for massively parallel numerical simulations and reduces the potential for errors. We achieve this by providing an integration layer between two existing open-source libraries: Boost Odeint [1, 2] provides the sequential, single-process time integration schemes, whereas OpenFPM [8] provides the parallel and distributed data structures and operators.

OpenFPM reduces development times for parallel numerical solvers by providing a template expression system to encode PDEs in near-mathematical notation [16]. Here, we complement this with a template system that uses Odeint for abstract encapsulation of time derivatives. This gives OpenFPM users access to the high-order and adaptive time-integration methods implemented in Odeint. Combining Odeint with OpenFPM is feasible, since both rely on C++ template metaprogramming to provide hardware-abstracted data structures and performance-portable algorithms. The implementation presented here seamlessly links the two and enables parallel solvers for arbitrary first-order ordinary differential equations (ODE) of the form

$$\frac{d\mathbf{u}}{dt} = \mathcal{F}\left(t, \mathbf{u}(t)\right),$$

starting from an initial condition $\mathbf{u}(0) = \mathbf{u}_0 \in \mathbb{R}^d$. The right-hand side $\mathcal{F}$ of this initial-value problem is a potentially nonlinear function given by the model that is solved or simulated, or it results from spatial discretization of a partial differential equation (PDE). In order for the computed sequence $\mathbf{u}_0 = \mathbf{u}(0\delta t)$, $\hat{\mathbf{u}}_1 \approx \mathbf{u}(1\delta t)$, $\hat{\mathbf{u}}_2 \approx \mathbf{u}(2\delta t)$,... to converge to the true dynamics $\mathbf{u}(t)$ when $\delta t \to 0$, a time-integration method must be both *consistent* and *stable*.

In simulations of nonlinear dynamics, identifying a consistent and stable time-integration method often requires experimenting with different step sizes and algorithms, or autotuning these choices [10]. Since the time-step size in an explicit time integrator is limited by the fastest dynamics to be resolved, methods with uniform step sizes are often wasteful, and instead adaptive methods are used [14], which adjust the step size to the simulated dynamics. This requires additional global communication of error estimates and step sizes among all processes. Both communications—state boundaries and global step size—are usually implemented in method-specific code.

Our generic OpenFPM–Odeint integration layer reduces the need for method-specific code by following the software design principle of separation of concerns. This means that the implementation of a distributed time-stepping scheme is not specific or limited to a given right-hand side $\mathcal{F}$, nor do the spatial operators used to discretize a PDE within $\mathcal{F}$ occur in the time-integration code. This allows modular combinations of right-hand sides and time-stepping methods, independent of the spatial discretization and the computer architecture used.

IMPLEMENTATION AND ARCHITECTURE

We make the explicit time-stepping schemes from `Boost::odeint` [2] available in OpenFPM [8] by providing an interface between these two software libraries. This interface is based on extending Odeint to OpenFPM data types and making the Odeint concepts available in OpenFPM's template metaprogramming expression system. Both OpenFPM and Odeint are templated C++ libraries.

ARCHITECTURE AND NOMENCLATURE OF BOOST ODEINT

Boost Odeint [2] offers a wealth of explicit time-stepping methods, and it also allows for implementing custom time steppers. Odeint uses template metaprogramming to render the implementation independent of the underlying data structures. The key data structure is the *state* of the ODE system, storing the vector $\mathbf{u}$ at a given time. The state is of abstract data type *state-type*, defined by a template parameter at compile time. An initial *state* has to be provided as input, along with a *System* function or functor that computes the right-hand side $\mathcal{F}$ for a given *state* $\mathbf{u}$. One can then use any available time stepper from Odeint, as summarized in Table 1, to advance the *state* to the next time step and update it in-place. When using dynamic step-size methods, the size of each time-step is determined from an error estimator and a predefined error tolerance.

All time integrators in Odeint are instantiated by a constructor specifying what Odeint calls an *Algebra*. An Odeint *Algebra* defines arithmetic operations on the *state-type*, allowing Odeint to compute linear combinations of *states* to, e.g., combine the stages in a multi-stage time integrator.

CATEGORY	METHOD	ACCURACY ORDER
fixed step size	explicit Euler	1
	modified midpoint	2
	Runge–Kutta 4	4
	Runge–Kutta–Cash–Karp 54	5
	Runge–Kutta–Dopri 5	5
	Runge–Kutta–Fehlberg 78	8
dynamic step size	Runge–Kutta–Cash–Karp 54	5
	Runge–Kutta–Dopri 5	5
	Runge–Kutta–Fehlberg 78	8
multi-step	Adams–Bashforth	1…8
	Adams–Bashforth–Moulton	1…8
symplectic	symplectic Euler	1
	velocity Verlet	2
	Runge–Kutta–McLachlan	4

Table 1 Explicit time-stepping schemes provided by Odeint.

To use steppers with dynamic step sizes, the *state* object must provide a method to compute a norm over the *state-type*. This norm is usually the l_∞ norm to bound the global error, but any other error norm can alternatively be implemented. In addition, a separate function or functor can be passed to Odeint as an *Observer*, which is called before each time step. This can be used, e.g., to write output to a file or to provide additional code to be run before each time step. The interplay of these objects in Odeint is illustrated in Figure 1.

By default, Odeint supports *state-types* such as `std::vector` and `boost::array`. None of the default *state-types*, however, allow for distributed computing on heterogeneous architectures. We relax this limitation by implementing custom *state-types* using the parallel and distributed data structures of OpenFPM [7]. This makes Odeint time steppers available to OpenFPM users and allows them to run on parallel computers and on (clusters of) GPUs.

THE DISTRIBUTED OpenFPM STATE TYPE AND ALGEBRA

We provide a custom Odeint *state-type*, along with its *Algebra*, for the distributed data types of OpenFPM. This OpenFPM *state-type* encapsulates OpenFPM's abstract data structures and performance-portable algorithms for shared- and distributed-memory computing, as well as multi-GPU computing. OpenFPM is implemented in C++ using template metaprogramming, making it compatible with Odeint in style and architecture. The custom Odeint

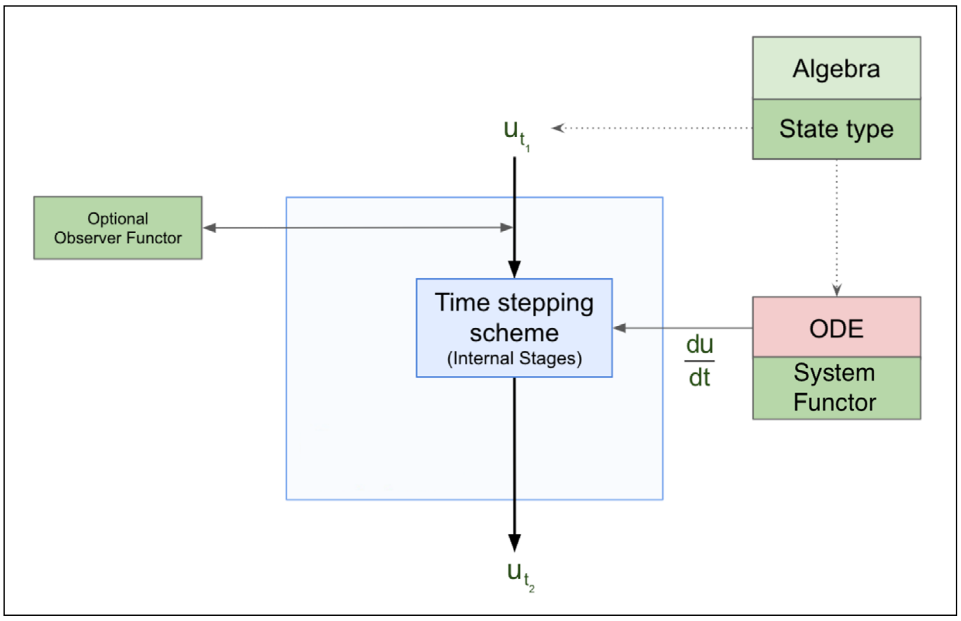
Algebra
State type
u_{t_1}
Optional Observer Functor
Time stepping scheme (Internal Stages)
du/dt
ODE
System Functor
u_{t_2}

Figure 1 Architecture of the Odeint library. In each iteration, the *state* $\mathbf{u}$ is advanced from a time t_1 to a later time $t_2 = t_1 + \delta t$, $\delta t > 0$. The right-hand side $\mathcal{F}(t,\mathbf{u}(t))$ of the ODE system is encapsulated in the *System* functor. An *Algebra* defines mathematical operations over the *state-type*, i.e., the data type of the *state*. An optional *Observer* functor can execute user code at the beginning of every time step.

state-types for OpenFPM are internally distributed and specialized for different hardware backends. This includes Nvidia and AMD GPU backends [7].

Specifically, we provide Odeint *state-types* and *Algebra*s for OpenFPM distributed vectors on CPUs and GPUs. These *state-types* are named `state_type_#d_ofp[_gpu]`, where `#` is the dimensionality *d* of the vectors in the array. The current implementation supports array dimensions between 1 and 6. For example, the *state-type* for a scalar field on the CPU is `state_type_1d_ofp` and for a 3D vector field on the GPU it is `state_type_3d_ofp_gpu`.

These custom *state-types* are complemented by a custom Odeint *Algebra*. This is required because the default *Algebra* of Odeint does not consider the domain decomposition of the internally distributed OpenFPM data structures. The custom *Algebra* implements the required arithmetic operations over the distributed OpenFPM data containers. For this, every discretization element (grid point or particle) in an OpenFPM container becomes an element of the Odeint *state-type*. Our custom *Algebra* accounts for domain decomposition by partitioning the *state-type* across processes and transparently managing communication between them as required for data synchronization.

The custom OpenFPM *Algebra* further defines the parallel iterators `for_each#()` and `for_each_prop#()`. They encapsulate concurrent operations over discretization elements and their individual properties in multi-dimensional fields, respectively. In these iterators, `#` is the order of the operation. For example, if the iterator is to apply a unary operator across all elements of a *state-type*, `#=1`. For a binary operator across pairs of state elements, `#=2`, and so on. The time-integration methods of Odeint use these iterators. The most complex Odeint integrators require iterators across up to `#=15` state elements.

For GPU backends, the custom *Algebra* encapsulates the operations to be dispatched to CUDA (for Nvidia GPUs) or HIP (for AMD GPUs) kernels for efficient execution of element-wise computations and reductions (e.g., norm evaluations). This ensures performance portability across heterogeneous architectures [7] and provides a practical blueprint for developers looking to implement custom Odeint *Algebra*s for distributed or multi-GPU environments.

Finally, our custom *Algebra* for Odeint provides the methods `for_each_norm()` and `for_each_prop_resize()` for computing norms over *states* in adaptive time steppers and for distributed *state* object resizing, respectively. All problem- or method-specific code is generated by the compiler from this *Algebra* in conjunction with the OpenFPM template expression system for differential equations [16]. This ensures modularity of the design and performance portability across different hardware architectures.

```
//Templated with OpenFPM distributed vector type
template<typename vector_type>
struct state_type_2d_ofp{
  state_type(){}
  typedef size_t size_type;
  typedef int is_state_vector;
  aggregate<texp_v<double>,texp_v<double>> data;

  //Method to get the size
  size_t size() const
  { return data.get<0>().size(); }

  //Method to resize
  void resize(size_t n)
  {
    data.get<0>().resize(n);
    data.get<1>().resize(n);
  }
};

//Additional structs as required by Odeint
namespace boost::numeric::odeint {
  template<>
  struct is_resizeable<state_type<vector_type>> {
    typedef boost::true_type type;
    static const bool value = type::value;
  };
  template<typename T>
  struct vector_space_norm_inf<state_type<T>>
  {
    typedef typename T::stype result_type;
  };
}
```

Listing 1 The OpenFPM distributed vector *state-type* for Odeint.

Listing 1 shows an example for an OpenFPM distributed vector on the CPU with two components of type `double` expressed as an `aggregate` (Line 7). The container `texp_v` is a special container in OpenFPM's embedded domain-specific language [16]. It allows array and tensor operations to be expressed using MATLAB [12] or Numpy [6] syntax [16]. It also supports implicit mathematical equations that require matrix assembly and numerical solution of a system of equations. This container is, therefore, a convenient choice to store (temporary) results in a way that is compatible with the operators of OpenFPM's embedded domain-specific language. Because `texp_v` is a resizable object, we notify Boost that the entire *state-type* is resizable (Lines 22–25). In addition, since adaptive steppers require calculating a norm, we specify the result type for the infinity norm (Lines 28–32). The custom *Algebra* for this *state-type* is `odeint::vector_space_algebra_ofp`. It defines the mathematical operations on OpenFPM distributed vectors and needs to be specified at compile time (Listing 3, Line 6). Together, these objects enable the efficient, inlined computations required for intermediate stage evaluations of explicit multi-stage schemes.

QUALITY CONTROL

We use a Continuous Integration / Continuous Delivery (CI/CD) development methodology in order to accelerate the software development life cycle and improve the overall quality of the code. In our CI/CD pipeline, we systematically assure compatibility with multiple compilers (GNU Compiler Collection, Clang Compiler, Intel Compiler), operating systems (Linux, UNIX-like, macOS, Windows/Cygwin), and hardware (x86_64, AMD64, ARM64, Nvidia GPU, AMD GPU). As an orchestration tool, we use `Rundeck`. It supports automatic overnight builds and deployment, has interfaces to GitLab and GitHub to enable repository mirroring, and it has an interface to the official OpenFPM website https://openfpm.mpi-cbg.de to upload up-to-date reports on run-time performance, static analysis, and test coverage.

The website also contains information about compiling the software from source, installing the software in a Docker container, and how to use the examples supplied with the package. In addition, Doxygen code documentation is available at https://ppmcore.mpi-cbg.de/doxygen/openfpm/. The documentation is automatically generated from the source code with every build.

Code quality and correctness are ensured by unit tests. Our CI pipeline runs the tests upon every commit in the master branch. If any test fails, the developers are notified and the commits causing the fail are reverted. Since test coverage is difficult to evaluate for templated C++ code, we report the coverage estimated by the `gcov` tool. We ensure that this coverage is above 90% for each OpenFPM submodule. The test suite consists of more than 600 unit tests covering more than 95% of the entire OpenFPM project. Finally, our CI/CD pipeline facilitates code dissemination by providing automatically built Docker container images. This improves portability and reduces the entry barrier for new users.

CORRECTNESS TESTS

We benchmark the implementation for correctness on problems with known analytical solution. In order to generate many such problems, we consider the parametric families of exponential and sigmoidal dynamics defined by:

$$\frac{\partial u(t)}{\partial t} = xye^{t},\ (x,y)\in[0,1]\times[0,1], \qquad (1a)$$

$$\frac{du(t)}{dt} = \frac{1}{1+e^{-t}}\left(1-\frac{1}{1+e^{-t}}\right) \qquad (1b)$$

The exponential family contains as many different ODEs as the number of discretization points in space, which is representative of systems of ODEs arising from the spatial discretization of PDEs. We use the implementation presented above to solve this problem with the analytical solution as initial condition at $t_0 = -5$. We compare the computed solutions with the analytical solution at final time $t_f = 5$ using the infinity norm $||\mathbf{e}||_\infty = \max(|e_1|, ..., |e_n|)$ and the Euclidean L_2 norm $||\mathbf{e}||_2 = \sqrt{e_1^2 + \ldots + e_n^2}$ of the absolute error $\mathbf{e} = \mathbf{u} - \mathbf{u}_{\text{exact}}$ across all n time steps. Methods with fixed time-step size are tested for different δt to verify their order of convergence.

We first test the fixed-step multi-stage methods Runge–Kutta 4, Runge–Kutta–Dopri 5, and Runge–Kutta–Fehlberg 78 (see Table 1) for the exponential dynamics in Eq. (1a). Figure 2a shows the convergence plots for increasing numbers of time steps. All methods converge with the expected order, as indicated by the solid lines, until machine precision is reached for the `double` data type used, and finite-precision round-off errors start accumulating. The same is observed in Figure 2b for the sigmoidal dynamics of Eq. (1b).

Next, we test the fixed-step Adams–Bashforth integrator as an example of a multi-step method. We solve the same problems using 1 to 8 steps, resulting in convergence orders between 1 and 8. This is verified for the exponential and sigmoidal dynamics in Figure 2c and 2d, respectively. In all cases, the theoretically optimal order of convergence is observed down to machine precision.

All tests are repeated for different numbers of OpenFPM processes (1, 2, 6, 24) to confirm that the results are the same, regardless of the degree of parallelism. The implementation of our custom Odeint *state-type* and *Algebra* is the same for all integrators from Table 1.

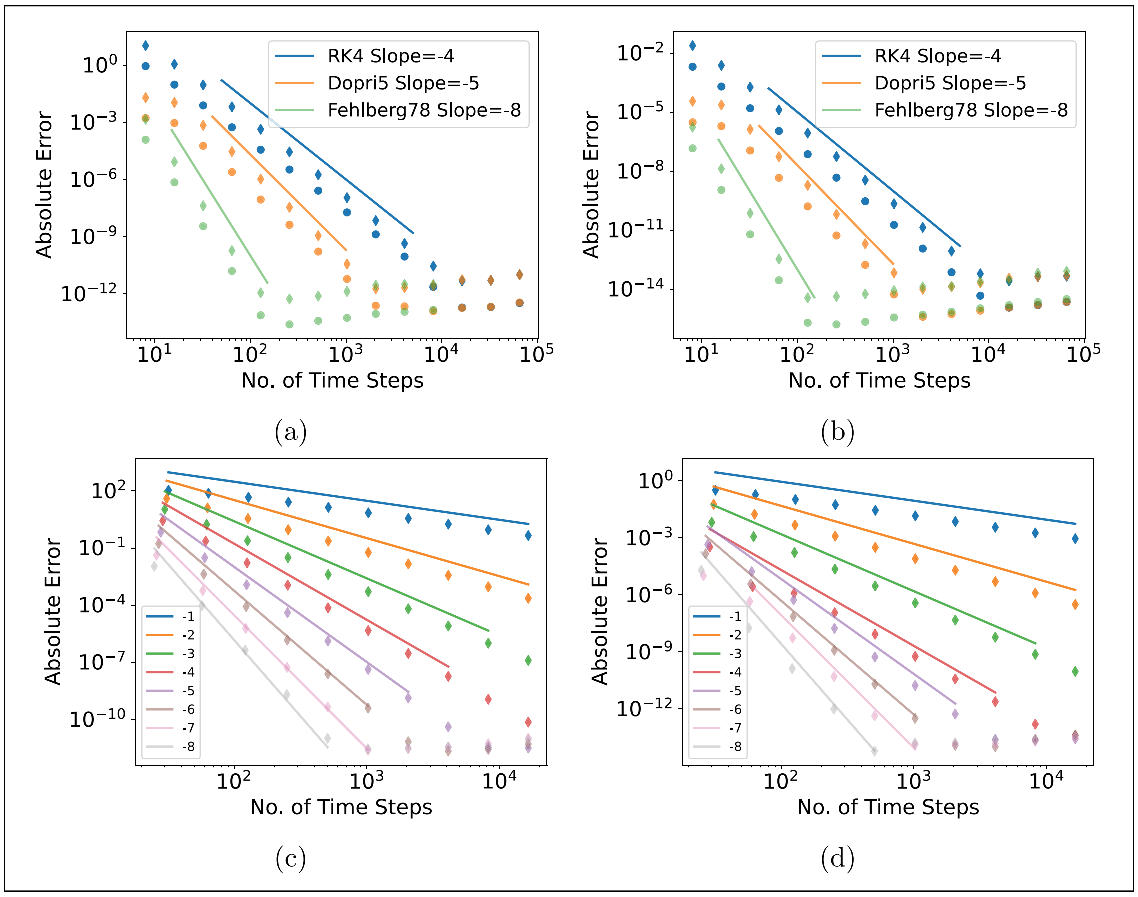

Figure 2 Numerical convergence of time-integration schemes with increasing numbers of time steps. (a,b) L_∞(◆) and L_2(●) error norms for the exponential dynamics of Eq. (1a) **(a)** and the sigmoidal dynamics of Eq. (1b) **(b)** with different one-step multi-stage methods (colors, inset legend). Solid lines indicate the theoretically expected slopes. **(c,d)** L_∞(◆) error norms for the exponential (c) and sigmoidal (d) dynamics solved using Adams–Bashforth with different numbers of steps (1...8, colors, inset legend). Solid lines indicate the theoretically expected slopes.

These results therefore confirm the correctness of the implementation.

PERFORMANCE TESTS

We test the parallel scalability of the present software for multi-stage methods, which incur additional communication overhead in a distributed-memory parallel program when the ODEs result from spatial discretization. We again test the fixed-step time integrators Runge–Kutta 4, –Dopri 5, and –Fehlberg 78, and additionally –Dopri 5 with adaptive time steps. As a baseline, we use a native implementation of Runge–Kutta 4 in OpenFPM without using the presented Odeint interface.

We solve the exponential dynamics given in Eq. (1a) on 512×512 points that discretize the domain $(x, y) \in [0, 1]^2$ with equal spacing. This amounts to 262,144 ODEs from the exponential family, which are distributed among the processes using OpenFPM's domain decomposition and solved in parallel. We perform a strong scaling measurement with a constant problem size distributed across increasing numbers of processes (1 process per CPU core). The measurements are taken on a cluster of Intel Xeon E5-2680v3 CPUs @2.5 GHz with each node containing 24 (2×12) cores with shared memory. The nodes in the cluster are connected using a 4-lane FDR InfiniBand network (14 Gb/s per lane) with a latency of 0.7 µs for message passing using the OpenMPI library.

The results are shown in Figure 3a. The present OpenFPM+Odeint implementation is almost an order of magnitude faster than the native OpenFPM-only implementation. This is likely due to the performance gains from optimizing the on-the-fly computation of stages in Odeint [2]. Since there is no spatial coupling in this problem, we find close to ideal (solid black line) scaling of the computational time for all tested steppers. This establishes the ideal baseline scalability in the absence of spatial coupling.

To test how the performance changes when spatial coupling and corresponding communication overhead is introduced, we consider the 3D Gray-Scott reaction-diffusion problem that couples the ODEs in space:

$$\frac{\partial}{\partial t}\begin{bmatrix} C_0(t,x,y,z) \\ C_1(t,x,y,z) \end{bmatrix} = \begin{bmatrix} d_1\Delta_{(x,y,z)}C_0 - C_0C_1^2 + F(1-C_0) \\ d_2\Delta_{(x,y,z)}C_1 + C_0C_1^1 - (F+K)C_0 \end{bmatrix} \quad (2)$$

for the parameters $d_1 = 2 \cdot 10^{-4}$, $d_2 = 10^{-4}$, $F = 0.053$, and $K = 0.014$. The Laplace operators in space $\Delta_{(x,y,z)}$ are discretized using the DC-PSE method [15] as implemented in the template expression system of OpenFPM [16]. We

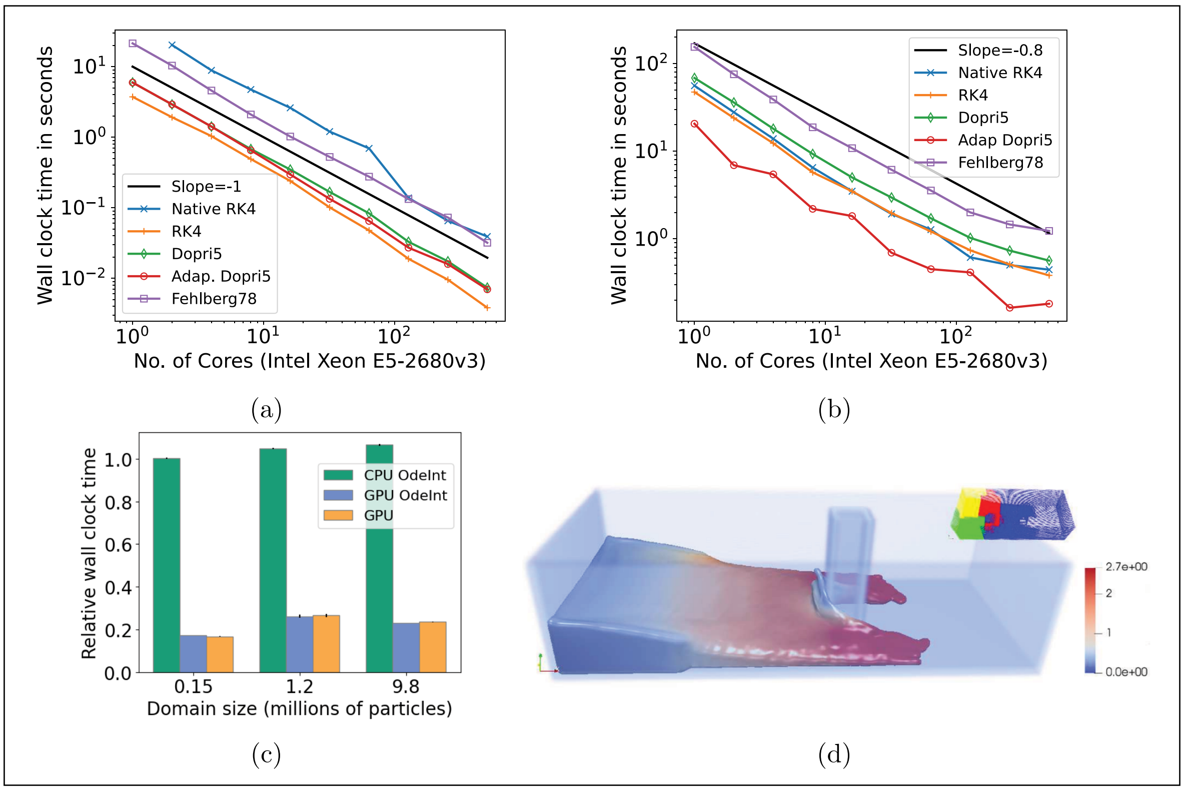

Figure 3 Strong scaling of the OpenFPM+Odeint time integration schemes with increasing numbers of CPU cores and on a GPU. **(a)** Average wall-clock times in seconds over three independent runs of solving the exponential dynamics from Eq. (1a) using different one-step multi-stage methods (colors, inset legend, error bars below symbol size). The solid black line indicates the optimal speed-up. **(b)** Average wall-clock times in seconds over 3 independent runs for the 3D Gray-Scott Eq. (2) using different one-step multi-stage methods (colors, inset legend, error bars below symbol size). The solid black line indicates the speed-up for 80% parallel efficiency. **(c)** Average wall-clock speedups for the SPH dam break case, normalized to the run times reported for the reference CPU implementation without Odeint from [8] (=1.0). All results are averaged over three independent runs (error bars show standard deviation) for different numbers of particles (bar groups). Speedups are given for a single Nvidia GeForce RTX 4090 GPU (blue bars), compared with running the code without the present Odeint interface on the same GPU (orange bars), and running and OpenFPM+Odeint code on 32 cores of an AMD Ryzen 3990X CPU (green bars). **(d)** Visualization of the fluid, colored by velocity magnitude (color bar), of the SPH dam break case with 1.2 million particles. An OpenFPM domain decomposition onto four processes is shown in the inset figure (one color per process).

solve the Gray-Scott model in the 3D cube $[0,2.5]^3$ with periodic boundary conditions on a regular Cartesian grid of 64 × 64 × 64 points, time stepping with $\delta t = 1$ until final time $t_f = 20$. The wall-clock times for different multi-stage schemes are plotted in Figure 3b. We find a parallel efficiency of about 80% up to 512 CPU cores (solid black line) for all tested multi-stage schemes. Inter-processor communication of stages and of spatial differential operators are performed transparently (inside the Odeint *System* functor) using OpenFPM's MPI backend. Here, the native OpenFPM implementation has similar performance as the OpenFPM+Odeint implementation, suggesting the bottleneck to be the spatial derivatives computation. Adaptive time stepping does not incur a significant scalability penalty, while raw wall-clock times roughly halve. Taken together, these results show that the present OpenFPM–Odeint interface does not introduce any new performance bottlenecks.

While involving non-trivial time stepping, the previous test case was still relatively simple in terms of the equations that are solved. We therefore next quantify the performance of the present software in a more complex, real-world test case. For this, we consider the "dam break" scenario from computational fluid dynamics, which we simulate using Smoothed Particle Hydrodynamics (SPH) as previously described [4]. This solves the weakly compressible fluid mechanics of water slushing around a U-profile beam in a rectangular tank. This problem involves non-trivial geometries and free-surface flows. In this simulation, the water body is discretized with irregularly spaced Lagrangian particles p whose velocities $\boldsymbol{v}_p$ and densities ρ_p evolve according to:

$$\frac{d\boldsymbol{v}_p}{dt} = -\sum_{q\in\mathcal{N}(p)} m_q \left(\frac{P_p + P_q}{\rho_p \rho_q} + \Pi_{pq} \right) \nabla W\left(\boldsymbol{x}_q - \boldsymbol{x}_p\right) + \boldsymbol{g}, \quad (3a)$$

$$\frac{d\rho_p}{dt} = \sum_{q\in\mathcal{N}(p)} m_q \boldsymbol{v}_{pq} \cdot \nabla W\left(\boldsymbol{x}_q - \boldsymbol{x}_p\right), \quad (3b)$$

$$P_p = \frac{1}{\gamma} c_{sound}^2 \, \|\boldsymbol{g}\|_2 \, h_{swl} \rho_0 \left[\left(\frac{\rho_p}{\rho_0} \right)^{\gamma} - 1 \right]. \quad (3c)$$

Here, h_{swl} is the initial height of the fluid, γ = 7, and c_{sound}=20 [13]. $\mathcal{N}(p)$ 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 classic cubic SPH kernel [13], 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. The equation of state (Eq. 3c) relates the hydrostatic pressure P_p to the density ρ_p, where ρ_0 is the density of water at P=0. The viscosity term Π_{pq} is defined as:

$$\Pi_{pq} = \begin{cases} -\dfrac{\alpha \bar{c}_{pq}\mu_{pq}}{\bar{\rho}_{pq}} & \boldsymbol{v}_{pq}\cdot\boldsymbol{r}_{pq} > 0, \\ 0 & \boldsymbol{v}_{pq}\cdot\boldsymbol{r}_{pq} < 0. \end{cases} \tag{4}$$

The vector $\boldsymbol{r}_{pq} = \boldsymbol{r}_p - \boldsymbol{r}_q$ points from particle q to particle p, and the constants are defined as: $\mu_{pq} = \frac{h\boldsymbol{v}_{pq}\cdot\boldsymbol{r}_{pq}}{\|\boldsymbol{r}_{pq}\|_2^2+\eta^2)}$, with a smoothing parameter η, and $\bar{c}_{pq} = c_{sound}\sqrt{\|\mathbf{g}\|_2 h_{swl}}$. All parameters and settings are as specified in the original case description [4] and taken from https://github.com/DualSPHysics/DualSPHysics/tree/master/examples/main/01_DamBreak. We solve the equations using the SPH method implemented in OpenFPM and the Euler method from Odeint to perform Verlet time stepping with the step size computed for a Courant-Friedrichs-Lewy (CFL) number of 0.2. As a baseline, we compare with the handwritten OpenFPM CPU implementation from [8].

We first quantify how much overhead the present Odeint interface adds over the manually implemented time integration used in the reference code (= speedup 1.0) when using all 32 cores of an AMD Ryzen 3990X CPU in parallel. The green bars in Figure 3c show that the overhead is below 7% (between 0.3% and 6.6%, depending on problem size) for particle numbers ranging from 150,000 to 9.8 million. An example visualization of the simulation result with 1.2 million particles is shown in Figure 3d, along with the automatically determined domain decomposition. The visualization is shown for four parallel processes (one per CPU core) for better visual clarity.

After having established the baseline overhead introduced by the present OpenFPM–Odeint interface on the CPU, we quantify the speedup achieved on a single Nvidia GeForce RTX 4090 consumer GPU. The present software natively supports parallel computing on GPUs through the GPU *state-types* as described above. GPU settings, like block size, threads/block, etc., are controlled by the user as usual. The settings used here are given in example/Numerics/OdeInt/SPH_dlb_gpu/main.cu. Again, we compare with a handwritten reference implementation (orange bars in Figure 3c) and the implementation using the present Odeint interface (blue bars) for time stepping. For all problem sizes tested, the code is about 5 times faster on the GPU than on 32 CPU cores. The higher GPU speedup for the smallest problem size is likely due to lower data-transfer overhead. The Odeint interface does not add any significant overhead on the GPU when compared with the handwritten GPU code (orange bars). Importantly, the GPU version of the present code did not require writing any CUDA by hand. All GPU targeting is done automatically by the OpenFPM backend as previously described [7].

In conclusion, this real-world case shows the applicability of the present OpenFPM–Odeint integration to a realistically complex application on both CPU and GPU.

(2) AVAILABILITY

We designed and implemented an interface between Boost Odeint [2] and OpenFPM [8]. This interface enables compact codes for scalable time integration on parallel computers with performance portability between CPU and GPU clusters. The presented software implementation is based on a custom state type and an internally distributed algebra for Odeint. These objects were implemented using the templates and distributed data types from the OpenFPM library. The interface is bi-directional, making distributed OpenFPM types and network communication available in Odeint objects, and making the Odeint time integrators available in OpenFPM's domain-specific language for differential equations [16]. This endows the template expression language of OpenFPM with native primitives for time derivatives, which can be evaluated using Odeint's wealth of methods.

We tested the present implementation for correctness and performance. The results verified correct convergence orders up to finite machine precision. We demonstrated that our implementation scales ideally in the absence of inter-process communication and has a parallel efficiency of about 80% (strong scaling, 512 CPU cores) with the typical communication overhead of a coupled PDE problem in 3D, solved using different—adaptive and non-adaptive—multi-stage schemes. The additional Odeint abstractions did not add detectable overhead over a native OpenFPM implementation, and in some cases were even faster, due to the highly optimized code of Odeint.

In summary, the present software implementation simplifies the use of time-stepping methods in scientific problem-solving environments, such as the OpenPME environment [11, 9], on distributed CPU and GPU computers. This has the potential of extending autotuning systems for spatial discretization methods [10] to the time dimension, as changing the time stepper, or the step size, amounts to a single-codeline change, which can be implemented in an autotuning framework. The presented system allows for rapid rewriting of distributed and parallel numerical solvers with minimal changes to the source code, while maintaining scalability and performance portability.

OPERATING SYSTEMS

x86_64/AMD64: Linux, macOS & Unix-like, Windows only with Cygwin (partial support); ARM64: macOS (min. macOS 13 Ventura).

PROGRAMMING LANGUAGES

C++14, CUDA (min. 9.2, for Nvidia GPU support), HIP (min. 5.0.0, for AMD GPU support)

MINIMAL HARDWARE REQUIREMENTS

x86_64 or AMD64 CPU for Linux/Unix/macOS/Windows systems; ARM64 CPU for macOS. Minimum resources in all cases: 1 core, 1 GB RAM, 15 GB disk space, text terminal access. Optionally: a supported GPU from Nvidia or AMD. Minimal multi-GPU setup tested: Intel(R) Xeon(R) CPU E5-2660v3 with 2 x Nvidia GeForce RTX 4090.

SOFTWARE DEPENDENCIES

See Table 2.

LIST OF CODE CONTRIBUTORS

Code contributors in alphabetical order are:

- Pietro Incardona
- Landfried Kraatz
- Abhinav Singh
- Serhii Yaskovets

SOFTWARE LOCATION

Code repository

Name: GitHub

Persistent identifier: https://github.com/mosaic-group/openfpm

Licence: 3-clause BSD

Date published: January 15, 2026

Web site: https://openfpm.mpi-cbg.de/

Availability of support and help

Issue tracker: https://github.com/mosaic-group/openfpm/issues

Documentation: https://ppmcore.mpi-cbg.de/doxygen/openfpm/Odeint.html

Examples: https://github.com/mosaic-group/openfpm/tree/develop/example

E-mail: Please contact the corresponding author of this manuscript under the given e-mail address or the development team under the address provided in the footer of the project web page https://openfpm.mpi-cbg.de/.

DOCUMENTATION AND SUPPORT LANGUAGE

English

DEPENDENCY	USED BY	OPTIONAL	VERSION
OpenMPI	openfpm_vcluster	NO	4.1.6
METIS	openfpm_pdata	YES (or ParMETIS)	5.1.0
ParMETIS	openfpm_pdata, openfpm_numerics	YES (or METIS)	4.0.3
BOOST	openfpm_data, openfpm_vcluster, openfpm_io, openfpm_pdata, openfpm_numerics	NO	1.84.0
zlib	openfpm_io	NO	1.3.1
HDF5	openfpm_io	NO	1.14.3
Vc	openfpm_data	NO	1.4.4
libhilbert	openfpm_data	NO	master
HIP	openfpm_devices	Yes	
alpaka	openfpm_devices	Yes	
OpenBLAS	openfpm_numerics	NO	0.3.26
suitesparse	openfpm_numerics	NO	5.7.2
Eigen	openfpm_numerics	Yes (or Petsc)	3.4.0
Blitz++	openfpm_numerics	NO	1.0.2
Algoim	openfpm_numerics	NO	master
PETSc	openfpm_numerics	Yes (or Eigen)	3.20.5

Table 2 Software dependencies of OpenFPM.

(3) REUSE POTENTIAL

With the OpenFPM *state-types* and *Algebra* described here, a user can implement Odeint solvers for dynamical systems using OpenFPM's embedded domain-specific language for differential equations [16]. Any time integrator provided by Odeint can be used to approximate time derivatives in OpenFPM code. The performance portability and scalability of the OpenFPM data structures are inherited.

The present implementation is currently limited to the explicit steppers available in the Odeint library and to fields of dimensionality ≤6. While Odeint's abstractions allow for defining arbitrary custom steppers, our algebra is currently limited to steppers requiring no more than 15 inline methods. Further, implicit and stiff Odeint time integrators are not currently supported, as they are tied to uBLAS, which does not support generic types or parallelism. Support for SUNDIALS [5] could be added in the future as another time-integration backend.

Despite these limitations, the presented implementation has already proven instrumental in solving active hydrodynamic equations in complex-shaped 3D geometries [17]. This has led to the discovery of novel wrinkling instabilities in biological active fluids [18, 3]. The governing equations of active hydrodynamics are challenging to solve, requiring extensive numerical experimentation and frequent changes to the code. This benefited from the rapid code rewriting afforded by the portable encapsulation of the OpenFPM–Odeint interface.

In order to implement an ODE solver, the right-hand side $\mathcal{F}$ of the initial-value problem needs to be provided as an Odeint *System* object with the signature of the *state* class. The *System* function or functor is called by the time stepper at each stage. It is parameterized by the templated *state-type* and the data type of its time derivative. Although it is unlikely that the solution state $\mathbf{u}$ and its time derivative $d\mathbf{u}/dt$ have different data types, Odeint allows for them to be specified separately in order to provide the most general implementation possible. Finally, the current time t is a parameter for the *System*. Using these parameters, the *System* evaluates the right-hand side of the ODE system, $\mathcal{F}(t,\mathbf{u}(t))$.

Listing 2 shows the *System* functor for the 3D Gray-Scott reaction-diffusion problem considered in the performance benchmarks above. This example also illustrates that it is straightforward to integrate templated OpenFPM operators for spatial derivatives into the ODE system (Laplacian `Lap`, Lines 5, 8, 25, 26). The *System* functor can also contain additional code that is to be run for every right-hand-side evaluation, e.g., for performance profiling. Since the data and the operations are transparently distributed by OpenFPM in a multi-node or multi-GPU computer, boundary layers

```
 1  //Templated type of the time derivative operator
 2  template<typename Laplacian_type>
 3  struct System_Functor
 4  {
 5    Laplacian_type &Lap
 6    double K = 0.053, F = 0.014, d1 = 2e-4, d2 = 1e-4; //Physical contants
 7
 8    System_Functor(Laplacian_type &Lap) : Lap(Lap)
 9    {}
10
11    void operator()(const state_type_2d_ofp &u, state_type_2d_ofp &dudt,
12    const double t ) const
13    {
14      //Get OpenFPM distributed domain
15      ofp_dist_vector_type &Domain= *(ofp_dist_vector_type*) DistVecPointer;
16
17      auto C=getV<Conc>(Domain) ; //Get the property
18
19      //Get the data from the current state.
20      C[0]=u.data.get<0>();
21      C[1]=u.data.get<1>();
22
23      //Compute the time derivate using PDE expressions.
24      Domain.ghost_get<Conc>();
25      dudt.data.get<0>() = d1*Lap(C[0]) - C[0] * C[1] * C[1] + F - F * C[0];
26      dudt.data.get<1>() = d2*Lap(C[1]) + C[0] * C[1] * C[1] - (F+K) * C[1];
27    }
28  };
```

Listing 2 Odeint *System* functor for the 3D Gray-Scott problem in OpenFPM.

```
 1  //Initialize OpenFPM data strucures and spatial derivatives
 2  auto C=getV<Conc>(Domain);
 3  Laplacian Lap(Domain);
 4
 5  //Define stepper type
 6  odeint::runge_kutta4<state_type_2d_ofp, double, state_type_2d_ofp, double, odeint::
        vector_space_algebra_ofp> Odeint_rk4;
 7
 8  //Declare and initialize state
 9  state_type_2d_ofp u;
10  u.data.get<0>()=C[0];
11  u.data.get<1>()=C[1];
12
13  //Initialize System functor with spatial Laplacian R.H.S.
14  System_Functor<Laplacian> System(Lap);
15
16  //Invoke RK4 stepper for t=[0,20] with time step 0.1
17  double t =0, tf = 20, dt = 0.1;
18  size_t steps = integrate_const(Odeint_rk4(), System, u, t, tf, dt);
```

Listing 3 An OpenFPM program using an Odeint time stepper.

("ghost layers") may need to be communicated between processes. This MPI communication is performed in the OpenFPM *Domain* class. A reference to the OpenFPM *Domain* object is therefore required in Line 15. Data from the *state* is then stored in the respective property of the OpenFPM *Domain* compatible with the spatial operators computed. The required inter-process communication of stage data is transparently performed during right-hand-side evaluation from within the *System* functor (Line 24), guaranteeing consistent *states* to Odeint at all times.

After defining the *System*, any Odeint time stepper can be used for time integration in the `main()` program, as shown in Listing 3 for the *System* from Listing 2.

The time integrator used in Line 6 is one of the built-in steppers of Odeint (see Table 1). Before starting time integration, we initialize the *state* in Lines 9–11 using the 2D OpenFPM CPU vector state type `state_type_2d_ofp`. The two components of the vector are then set to the initial condition $\mathbf{u}_0$. The correct distributed *Algebra* for this state type is instantiated in Line 6 from `odeint::vector_space_algebra_ofp` as described above. We then initialize the system functor from Listing 2 in Line 14. Finally, the time stepper with constant step size is invoked to run until a given final time tf in Line 18. Alternatively, the method `do_step()` of the stepper could be called to perform a single time step only. Adaptive time stepping with error-controlled steppers can be enabled by calling `integrate_adaptive()` with the desired tolerance. The stepper returns the number time steps performed, and the state is updated in-place. Thanks to the use of OpenFPM, the program in Listing 3 compiles and runs on shared- and distributed-memory CPU systems, as well as on Nvidia and AMD GPUs and multi-GPU setups.

FUNDING STATEMENT

This work was funded in parts by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) under grants SB-350008342 ("OpenPME") (author P.I.) and EXC-2068, Cluster of Excellence "Physics of Life" (author I.F.S.), and by the Federal Ministry of Education and Research of Germany (Bundesministerium für Bildung und Forschung, BMBF) under funding codes 031L0160 (project "SPlaT-DM – computer simulation platform for topology-driven morphogenesis") (author A.S.) and the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) Dresden/Leipzig (author A.S.).

COMPETING INTERESTS

The authors have no competing interests to declare.

AUTHOR AFFILIATIONS

Abhinav Singh orcid.org/0000-0003-2180-8423
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany; Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany; Center for Systems Biology Dresden, Dresden, Germany; Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden/Leipzig, Germany

Landfried Kraatz orcid.org/0009-0009-5215-825X
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany

Serhii Yaskovets orcid.org/0009-0000-1912-0254
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany; Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany; Center for Systems Biology Dresden, Dresden, Germany

Pietro Incardona
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany; Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany; Center for Systems Biology Dresden, Dresden, Germany

Ivo F. Sbalzarini orcid.org/0000-0003-4414-4340
Dresden University of Technology, Faculty of Computer Science, Dresden, Germany; Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany; Center for Systems Biology Dresden, Dresden, Germany; Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), Dresden/Leipzig, Germany

REFERENCES

1. **Ahnert K, Mulansky M, Simos TE, Psihoyios G, Tsitouras C, Anastassi Z.** Odeint – solving ordinary differential equations in C++. In *AIP Conference Proceedings*. AIP; 2011. DOI: https://doi.org/10.1063/1.3637934
2. **Ahnert M.** ODEINT state types, algebras and operations; 2015. URL https://www.boost.org/doc/libs/1_66_0/libs/numeric/odeint/doc/html/index.html. [Online; accessed 20-April-2021].
3. **Alam S, Najma B, Singh A, Laprade J, Gajeshwar G, Yevick HG, Baskaran A, Foster PJ, Duclos G.** Active Fréedericksz transition in active nematic droplets. *Phys. Rev. X*. 2024;14:041002. URL https://link.aps.org/doi/10.1103/PhysRevX.14.041002
4. **Crespo A, Domínguez J, Rogers B, Gómez-Gesteira M, Longshaw S, Canelas R, Vacondio R, Barreiro A, García-Feal O.** Dualsphysics: Open-source parallel cfd solver based on smoothed particle hydrodynamics (sph). *Computer Physics Communications*. 2015;187:204–216. URL https://www.sciencedirect.com/science/article/pii/S0010465514003397
5. **Gardner DJ, Reynolds DR, Woodward CS, Balos CJ.** Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*; 2022. DOI: https://doi.org/10.1145/3539801
6. **Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE.** Array programming with NumPy. *Nature*. 2020;585(7825):357–362. DOI: https://doi.org/10.1038/s41586-020-2649-2
7. **Incardona P, Gupta A, Yaskovets S, Sbalzarini IF.** A portable C++ library for memory and compute abstraction on multi-core CPUs and GPUs. *Concurrency Computat. Pract. Exper*. 2023;e7870. DOI: https://doi.org/10.1002/cpe.7870
8. **Incardona P, Leo A, Zaluzhnyi Y, Ramaswamy R, Sbalzarini IF.** OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications*. 2019;241:155–177. URL https://www.sciencedirect.com/science/article/pii/S0010465519300852
9. **Karol S, Nett T, Castrillon J, Sbalzarini IF.** A domain-specific language and editor for parallel particle methods. *ACM Trans. Math. Softw*. 2018;44(3):34:1–34:32. DOI: https://doi.org/10.1145/3175659
10. **Khouzami N, Michel F, Incardona P, Castrillon J, Sbalzarini IF.** Model-based autotuning of discretization methods in numerical simulations of partial differential equations. *Journal of Computational Science*. 2022;57:101489. URL https://www.sciencedirect.com/science/article/pii/S1877750321001563
11. **Khouzami N, Schütze L, Incardona P, Kraatz L, Subic T, Castrillon J, Sbalzarini IF.** The OpenPME problem solving environment for numerical simulations. In Paszynski M, Kranzlmüller D, Krzhizhanovskaya VV, Dongarra JJ, Sloot PMA, editors. Computational Science – ICCS 2021. *Lecture Notes in Computer Science, Springer International Publishing, Cham*. 2021:614–627. DOI: https://doi.org/10.1007/978-3-030-77961-0_49
12. **MATLAB.** *Version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts; 2010.
13. **Monaghan JJ.** Smoothed particle hydrodynamics. *Annu. Rev. Astron. Astrophys*. 1992;30:543–574. DOI: https://doi.org/10.1146/annurev.aa.30.090192.002551
14. **Press WH, Teukolsky SA.** Adaptive stepsize Runge-Kutta integration. *Computers in Physics*. 1992;6(2):188–191. DOI: https://doi.org/10.1063/1.4823060
15. **Schrader B, Reboux S, Sbalzarini IF.** Discretization correction of general integral pse operators for particle methods. *Journal of Computational Physics*. 2010;229(11):4159–4182. DOI: https://doi.org/10.1016/j.jcp.2010.02.004
16. **Singh A, Incardona P, Sbalzarini IF.** A C++ expression system for partial differential equations enables generic simulations of biological hydrodynamics. *Eur. Phys. J. E*. 2021;44(9):117. DOI: https://doi.org/10.1140/epje/s10189-021-00121-x
17. **Singh A, Suhrcke PH, Incardona P, Sbalzarini IF.** A numerical solver for active hydrodynamics in three dimensions and its application to active turbulence. *Physics of Fluids*. 2023;35(10). DOI: https://doi.org/10.1063/5.0169546
18. **Singh A, Vagne Q, Jülicher F, Sbalzarini IF.** Spontaneous flow instabilities of active polar fluids in three dimensions. *Phys. Rev. Res*. 2023;5:L022061. URL https://link.aps.org/doi/10.1103/PhysRevResearch.5.L022061

TO CITE THIS ARTICLE:
Singh A, Kraatz L, Yaskovets S, Incardona P, Sbalzarini IF 2026 Integrating Odeint Time Stepping into OpenFPM for Distributed and GPU Accelerated Numerical Solvers. *Journal of Open Research Software,* 14: 15. DOI: https://doi.org/10.5334/jors.598

Submitted: 08 July 2025 **Accepted:** 17 February 2026 **Published:** 06 March 2026