An efficient particle tracking algorithm for large-scale parallel pseudo-spectral simulations of turbulence

Cristian Lalescu, Bérenger Bramas, Markus Rampp, Michael Wilczek

An Efficient Particle Tracking Algorithm for Large-Scale Parallel Pseudo-Spectral Simulations of Turbulence

Cristian C. Lalescu[a,b,1], Bérenger Bramas[c,b,d], Markus Rampp[b], Michael Wilczek[a,e]

[a]*Max Planck Institute for Dynamics and Self-Organization, 37077 Göttingen, Germany*
[b]*Max Planck Computing and Data Facility, 85748 Garching, Germany*
[c]*Inria Nancy – Grand-Est, CAMUS Team, 54600 Villers-lès-NancyFrance*
[d]*ICube, ICPS Team, 67400 Illkirch-Graffenstaden, France*
[e]*Theoretical Physics I, University of Bayreuth, 95440 Bayreuth, Germany*

Abstract

Particle tracking in large-scale numerical simulations of turbulent flows presents one of the major bottlenecks in parallel performance and scaling efficiency. Here, we describe a particle tracking algorithm for large-scale parallel pseudo-spectral simulations of turbulence which scales well up to billions of tracer particles on modern high-performance computing architectures. We summarize the standard parallel methods used to solve the fluid equations in our hybrid MPI/OpenMP implementation. As the main focus, we describe the implementation of the particle tracking algorithm and document its computational performance. To address the extensive inter-process communication required by particle tracking, we introduce a task-based approach to overlap point-to-point communications with computations, thereby enabling improved resource utilization. We characterize the computational cost as a function of the number of particles tracked and compare it with the flow field computation, showing that the cost of particle tracking is very small for typical applications.

Keywords: Particle tracking; Turbulence simulations; MPI; OpenMP

1. Introduction

Understanding particle transport in turbulent flows is fundamental to the problem of turbulent mixing [1, 2, 3, 4, 5, 6] and relevant for a wide range of applications such as dispersion of particles in the environment [7, 8, 9, 10], the growth of cloud droplets through collisions [11, 12, 13, 14, 15], and phytoplankton swimming in the ocean [16, 17, 18]. Direct numerical simulations (DNS) of turbulence are nowadays an established tool for investigating such phenomena and have a long history in scientific computing [19, 20, 21, 22, 23]. DNS have become a major application and technology driver in high-performance computing since the scale separation between the largest and the smallest scales increases drastically with the Reynolds number R_λ, which characterizes the degree of small-scale turbulence [24]. Dimensional estimates of the required computational resources scale at least as R_λ^6 [24]. Recent literature [25], however, shows that, due the occurrence of extremely small-scale structures, resolution requirements increase even faster than simple dimensional arguments suggest. Until today DNS have reached values of up to $R_\lambda \approx 2300$ [26, 27, 22], still smaller than the latest experiments, which have reached $R_\lambda > 5000$ [28], or natural settings such as cumulus clouds, which show Reynolds numbers on the order of 10^4 [29]. Hence DNS of turbulence will continue to be computationally demanding for the foreseeable future.

Due to the large number of grid points, practical implementations of DNS typically employ one- or two-dimensional domain decompositions within a distributed memory parallel programming paradigm. While the numerical solution of the field equations is typically achieved with well-established methods, the efficient implementation of particle tracking within such parallel approaches still poses major algorithmic challenges. In particular, particle tracking requires an accurate interpolation of the flow fields on distributed domains, and particles traversing the domain need to be passed on from one subdomain/process to another. As the Reynolds number increases, the number of particles required to adequately sample the turbulent fields needs to grow with the increasing number of grid points since this is a measure of the degrees of freedom of the flow. In addition higher-order statistics might be needed to address specific research questions, and thus the number of particles required for converged statistics increases as well [4, 30, 31, 32, 33, 34, 35, 36]. Overall, this requires an approach which handles the parallel implementation in an efficient manner for arbitrarily accurate methods. One option explored in the literature is the use of the high-level programming concept of *coarrays*, in practice shifting responsibility for some of the communication operations to the compiler, see [23]. The general solution that we describe makes use of MPI and OpenMP for explicit management of hardware resources. The combination of MPI [37] and OpenMP [38] has become a de facto standard in the development of large-scale applications [39, 40, 41, 42, 43, 44]. MPI [45] is used for communication

*Corresponding author.
E-mail address: Cristian.Lalescu@mpcdf.mpg.de

between processes and OpenMP to manage multiple execution threads over multicore CPUs using shared memory. Separately, large datasets must be processed with specific data-access patterns to make optimal use of modern hardware, as explained for example in [46].

To address the challenges outlined above, we have developed the numerical framework "Turbulence Tools: Lagrangian and Eulerian" (TurTLE), a flexible pseudo-spectral solver for fluid and turbulence problems implemented in C++ with a hybrid MPI/OpenMP approach [47, 48]. TurTLE allows for an efficient tracking of various types of particles. In particular, TurTLE showcases a parallel programming pattern for particle tracking that is easy to adapt and implement, and which allows for efficient use of available resources. Our event-driven approach is especially suited for the case where individual processes require data exchanges with several other processes while also being responsible for local work. For this, asynchronous inter-process communication and tasks are used, based on a combined MPI/OpenMP implementation. As we will show in the following, TurTLE permits numerical particle tracking at relatively small costs, while retaining flexibility with respect to number of particles and numerical accuracy. We show that TurTLE scales well up to $O(10^4)$ computing cores, with the flow field solver approximately retaining the performance of the used Fourier transform libraries for DNS with 3×2048^3 and 3×4096^3 degrees of freedom. We also measure the relative cost of tracking up to 2.2×10^9 particles as approximately only 10% of the total wall-time for the 4096^3 case, demonstrating the efficiency of the new algorithm even for very demanding particle-based studies.

In the following, we introduce TurTLE and particularly focus on the efficient implementation of particle tracking. Section 2 introduces the evolution equations for the fluid and particle models, as well as the corresponding numerical methods. Section 3 provides an overview of our implementation, including a more detailed presentation of the parallel programming pattern used for particle tracking. Finally, Section 4 summarizes a performance evaluation using up to 512 computational nodes.

2. Evolution equations and numerical method

2.1. Fluid equations

While TurTLE is developed as a general framework for a range of fluid equations, we focus on the Navier-Stokes equations as a prototypical example in the following. The incompressible Navier-Stokes equations take the form

$$\begin{aligned} \partial_t \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} &= -\nabla p + \nu \Delta \boldsymbol{u} + \boldsymbol{f}, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned} \tag{1}$$

Here, $\boldsymbol{u}(\boldsymbol{x}, t)$ denotes the three-dimensional velocity field, $p(\boldsymbol{x}, t)$ is the kinematic pressure, ν is the kinematic viscosity, and $\boldsymbol{F}(\boldsymbol{x}, t)$ denotes an external forcing that drives the flow. We consider periodic boundary conditions, which allows for the use of a Fourier pseudo-spectral scheme. Within this scheme, a finite Fourier representation is used for the fields, and the nonlinear term of the Navier-Stokes equations is computed in real space — an approach pioneered by Orszag and Patterson [19]. For the concrete implementation in TurTLE, we use the vorticity formulation of the Navier-Stokes equation, which takes the form

$$\partial_t \boldsymbol{\omega} = \nabla \times (\boldsymbol{u} \times \boldsymbol{\omega}) + \nu \Delta \boldsymbol{\omega} + \boldsymbol{F}, \tag{2}$$

where $\boldsymbol{\omega}(\boldsymbol{x}, t) = \nabla \times \boldsymbol{u}(\boldsymbol{x}, t)$ is the vorticity field and $\boldsymbol{F}(\boldsymbol{x}, t) = \nabla \times \boldsymbol{f}(\boldsymbol{x}, t)$ denotes the curl of the Navier-Stokes forcing. The Fourier representation of this equation takes the form [49, 50]

$$\partial_t \hat{\boldsymbol{\omega}}(\boldsymbol{k}, t) = \mathrm{i}\boldsymbol{k} \times \mathcal{F}[\boldsymbol{u}(\boldsymbol{x}, t) \times \boldsymbol{\omega}(\boldsymbol{x}, t)] - \nu k^2 \hat{\boldsymbol{\omega}}(\boldsymbol{k}, t) + \hat{\boldsymbol{F}}(\boldsymbol{k}, t), \tag{3}$$

where $\mathcal{F}$ is the direct Fourier transform operator. In Fourier space, the velocity can be conveniently computed from the vorticity using Biot-Savart's law,

$$\hat{\boldsymbol{u}}(\boldsymbol{k}, t) = \frac{\mathrm{i}\boldsymbol{k} \times \hat{\boldsymbol{\omega}}(\boldsymbol{k}, t)}{k^2}. \tag{4}$$

Equation (3) is integrated with a third-order Runge-Kutta method [51], which is an explicit Runge-Kutta method with the Butcher tableau

$$\begin{array}{c|ccc} 0 & & & \\ 1 & 1 & & \\ 1/2 & 1/4 & 1/4 & \\ \hline & 1/6 & 1/6 & 2/3 \end{array}. \tag{5}$$

In addition to the stability properties described in [51], this method has the advantage that it is memory-efficient, requiring only two additional field allocations, as can be seen from [49]

$$\begin{aligned} \hat{\boldsymbol{w}}_1(\boldsymbol{k}) &= (\hat{\boldsymbol{\omega}}(\boldsymbol{k}, t) + h\mathcal{N}[\hat{\boldsymbol{\omega}}(\boldsymbol{k}, t)])e^{-\nu k^2 h}, \\ \hat{\boldsymbol{w}}_2(\boldsymbol{k}) &= \tfrac{3}{4}\hat{\boldsymbol{\omega}}(\boldsymbol{k}, t)e^{-\nu k^2 h/2} + \tfrac{1}{4}(\hat{\boldsymbol{w}}_1(\boldsymbol{k}) + h\mathcal{N}[\hat{\boldsymbol{w}}_1(\boldsymbol{k})])e^{\nu k^2 h/2}, \\ \hat{\boldsymbol{\omega}}(\boldsymbol{k}, t+h) &= \tfrac{1}{3}\hat{\boldsymbol{\omega}}(\boldsymbol{k}, t)e^{-\nu k^2 h} + \tfrac{2}{3}(\hat{\boldsymbol{w}}_2(\boldsymbol{k}) + h\mathcal{N}[\hat{\boldsymbol{w}}_2(\boldsymbol{k})])e^{-\nu k^2 h/2}, \end{aligned} \tag{6}$$

where h is the time step, limited in practice by the Courant–Friedrichs–Lewy (CFL) condition [52]. The nonlinear term

$$\mathcal{N}[\hat{\boldsymbol{w}}(\boldsymbol{k})] = \mathrm{i}\boldsymbol{k} \times \mathcal{F}\left[\mathcal{F}^{-1}\left[\frac{\mathrm{i}\boldsymbol{k} \times \hat{\boldsymbol{w}}(\boldsymbol{k})}{k^2}\right] \times \mathcal{F}^{-1}\left[\hat{\boldsymbol{w}}(\boldsymbol{k})\right]\right] \tag{7}$$

is computed by switching between Fourier space and real space. If the forcing term is nonlinear, it can be included in the right-hand side of (7). To treat the diffusion term, we use the standard integrating factor technique [53] in (6).

Equation (3) contains the Fourier transform of a quadratic nonlinearity. Since numerical simulations are based on finite Fourier representations, the real-space product of the two fields will in general contain unresolved high-wavenumber harmonics, leading to aliasing effects [53]. In TurTLE, de-aliasing is achieved through the use of a smooth Fourier filter, an approach that has been shown in [54] to lead to good convergence to the true solution of a PDE, even though it does not completely re-

move aliasing effects.

The Fourier transforms in TurTLE are evaluated using the FFTW library [55]. Within the implementation of the pseudo-spectral scheme, the fields have two equivalent representations: an array of vectorial Fourier mode amplitudes, or an array of vectorial field values on the real-space grid. For the simple case of 3D periodic cubic domains of size $[0, 2\pi]^3$, the real-space grid is a rectangular grid of $N \times N \times N$ points, equally spaced at distances of $\delta \equiv 2\pi/N$. Exploiting the Hermitian symmetry of real fields, the Fourier-space grid consists of $N \times N \times (N/2 + 1)$ modes. Therefore, the field data consists of arrays of floating point numbers, logically shaped as the real-space grid or arrays of floating point number pairs (e.g. `fftw_complex`) logically shaped as the Fourier-space grid. Extensions to non-cubic domains or non-isotropic grids are straightforward.

The direct numerical simulation algorithm then has two fundamental constructions: loops traversing the fields, with an associated cost of $O(N^3)$ operations, and direct/inverse Fourier transforms, with a cost of $O(N^3 \log N)$ operations.

2.2. Particle equations

A major feature of TurTLE is the capability to track different particle types, including Lagrangian tracer particles, ellipsoids, self-propelled particles and inertial particles. To illustrate the implementation, we focus on tracer particles in the following.

Lagrangian tracer particles are virtual markers of the flow field starting from the initial position $\boldsymbol{x}$. Their position $\boldsymbol{X}$ evolves according to

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{X}(\boldsymbol{x}, t) = \boldsymbol{u}(\boldsymbol{X}(\boldsymbol{x}, t), t),\ \boldsymbol{X}(\boldsymbol{x}, 0) = \boldsymbol{x}. \tag{8}$$

The essential characteristic of such particle equations is that they require as input the values of various flow fields at arbitrary positions in space.

TurTLE combines multi-step Adams-Bashforth integration schemes (see, e.g., §6.7 in [56]) with spline interpolations [57] in order to integrate the ODEs. Simple Lagrange interpolation schemes (see, e.g., §3.1 in [56]) are also implemented in TurTLE for testing purposes. There is ample literature on interpolation method accuracy, efficiency, and adequacy for particle tracking, e.g. [20, 58, 59, 60, 61]. The common feature of all interpolation schemes is that they can be represented as a weighted real-space-grid average of a field, with weights given by the particle's position. For all practical interpolation schemes, the weights are zero outside of a relatively small kernel of grid points surrounding the particle, i.e. the formulas are "local". For some spline interpolations, a non-local expression is used, but it can be rewritten as a local expression where the values on the grid are precomputed through a distinct global operation [20] — this approach, for example, is used in [23].

Thus an interpolation algorithm can be summed up as follows:

1. compute $\widetilde{\boldsymbol{X}} = \boldsymbol{X} \mod 2\pi$ (because the domain is periodic).
2. find the closest grid cell to the particle position $\widetilde{\boldsymbol{X}}$, indexed by $\boldsymbol{c} \equiv (c_1, c_2, c_3)$.
3. compute $\bar{\boldsymbol{x}} = \widetilde{\boldsymbol{X}} - \boldsymbol{c}\delta$.
4. compute a sum of the field over a kernel of I grid points in each of the 3 directions, weighted by some polynomials:

$$\boldsymbol{u}(\boldsymbol{X}) \approx \sum_{i_1,i_2,i_3=1-I/2}^{I/2} \beta_{i_1}\left(\tfrac{\bar{x}_1}{\delta}\right)\beta_{i_2}\left(\tfrac{\bar{x}_2}{\delta}\right)\beta_{i_3}\left(\tfrac{\bar{x}_3}{\delta}\right)\boldsymbol{u}(\boldsymbol{c} + \boldsymbol{i}), \tag{9}$$

where $\boldsymbol{i} = (i_1, i_2, i_3)$. The cost of the sum itself grows as I^3, the cube of the *size of the interpolation kernel*. The polynomials β_{i_j} are determined by the interpolation scheme (see [57]).

Accuracy improves with increasing I. In TurTLE, interpolation is efficiently implemented even at large I. As discussed below in §3.3, this is achieved by sorting particle data such that a minimal number of MPI messages is used to complete all the triple sums, independently of the total number of particles.

3. Implementation

3.1. Overview

TurTLE follows the object-oriented paradigm to facilitate maintenance and extensions through class inheritance and virtualization. The solver relies on two types of objects. Firstly, all solvers inherit three basic elements from an abstract parent class: generic *initialization*, *do work* and *finalization* functionality. Solvers are then grouped in this first type of object, i.e. members of the resulting class hierarchy. The second type of object encapsulates essential data structures (i.e. fields, sets of particles) and associated functionality (e.g. HDF5-based I/O): these are "building block"-classes. Each solver then consists of a specific "arrangement" of the building blocks.

The parallelization of TurTLE is based on a standard, MPI-based, one-dimensional domain-decomposition approach: The three-dimensional fields are decomposed along one of the dimensions into a number of slabs, with each MPI process holding one such slab. In order to efficiently perform the costly FFT operations with the help of a high-performance numerical library such as FFTW, process-local, two-dimensional FFTs are interleaved with a global transposition of the data in order to perform the FFTs along the remaining dimension. A well-known drawback of the slab decomposition strategy offered by FFTW is its limited parallel scalability, because at most N MPI processes can be used for N^3 data. We compensate for this by utilizing the hybrid MPI/OpenMP capability of FFTW (or functionally equivalent libraries such as Intel MKL), which allows to push the limits of scalability by at least an order of magnitude, corresponding to the number of cores of a modern multi-core CPU or NUMA domain, respectively. All other relevant operations in the field solver can be straightforwardly parallelized with the help of OpenMP. Our newly developed parallel particle tracking algorithm has been implemented on top of this slab-type data decomposition using MPI and OpenMP, as shall be detailed below. Slab decompositions are beneficial for particle tracking since MPI communication overhead is minimized compared to, e.g., two-dimensional decompositions.

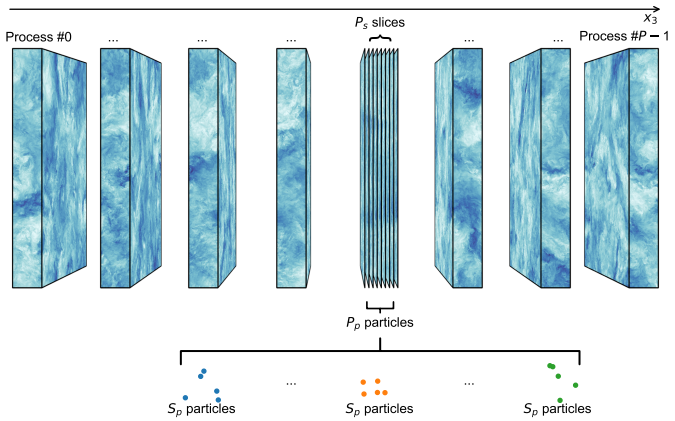

Figure 1: Distribution of real-space data between MPI processes in TurTLE. Fields are split into *slabs* and distributed between P MPI processes along the x_3 direction. The N_p particles are also distributed, with each MPI process storing P_p particles on average. Within each MPI process the particle data is sorted according to its x_3 location. This leads to a direct association between each of the P_s field slices to contiguous regions of the particle data arrays — in turn simplifying the interpolation procedure (see text for details). On average, S_p particles are held within each such contiguous region.

3.2. Fluid solver

The fluid solver consists of operations with field data, which TurTLE distributes among a total of P MPI processes with a standard slab decomposition, see Fig. 1. Thus the logical field layouts consist of $(N/P)\times N\times N$ points for the real-space representation, and $(N/P)\times N\times(N/2+1)$ points for the Fourier-space representation. This allows the use of FFTW [55] to perform costly FFT operations, as outlined above. We use the convention that fields are distributed along the real-space x_3 direction, and along the k_2 direction in the Fourier space representation (directions 2 and 3 are transposed between the two representations). Consequently, a problem on an N^3 grid can be parallelized on a maximum of N computational nodes using one MPI process per node and, possibly, OpenMP threads inside the nodes, see Fig. 1.

In the interest of simplifying code development, TurTLE uses functional programming for the costly traversal operation. Functional programming techniques allow to encapsulate field data in objects, while providing methods for traversing the data and computing specified arithmetic expressions — i.e. the class becomes a building block. While C++ allows for overloading arithmetic operators as a mechanism for generalizing them to arrays, our approach permits to combine several operations in a single data traversal, and it applies directly to operations between arrays of different shapes. In particular operations such as the combination of taking the curl and the Laplacian of a field (see (3)) are in practice implemented as a single field traversal operation.

3.3. Particle tracking

We now turn to a major feature of TurTLE: the efficient tracking of particles. The novelty of our approach warrants a more in-depth presentation of the data structure and the parallel algorithms, for which we introduce the following notations (see also Fig. 1):

- P : the number of MPI processes (should be a divisor of the field grid size N);
- $P_s = N/P$: the number of field slices in each slab;
- N_p : the number of particles in the system;
- P_p: the number of particles contained in a given slab (i.e. hosted by the corresponding process) — on average equal to N_p/P;
- S_p: the number of particles per slice, i.e. number of particles found between two slices — on average equal to N_p/N;
- I : the width of the interpolation kernel, i.e. the number of slices needed to perform the interpolation.

The triple sum (9) is effectively split into I double sums over the x_1 and x_2 directions, the results of which then need to be distributed/gathered among the MPI processes such that the sum along the x_3 direction can be finalized. Independently of P and N, there will be N_p sums of I^3 terms that have to be performed. However, the amount of information to exchange depends on the DNS parameters N, N_p, and I, and on the job parameter P.

Whenever more than one MPI process is used, i.e. $P > 1$, we distinguish between two cases:

1. $I \leq P_s$, i.e. each MPI domain extends over at least as many slices as required for the interpolation kernel. In this case particles are shared between at most two MPI processes, therefore each process needs to exchange information with two other processes. In this case, the average number of shared particles is $S_p(I-1)$.
2. $I > P_s$, i.e. the interpolation kernel always extends outside of the local MPI domain. The average number of shared particles is $S_p P_s$. Each given particle is shared among a maximum of $\lceil I/P_s \rceil$ processes, therefore each process must in principle communicate with $2\lceil I/(2P_s)\rceil$ other processes.

The second scenario is the more relevant one for scaling studies. Our expectation is that the communication costs will outweigh the computation costs, therefore the interpolation step should scale like $N_p I/P_s = N_p IP/N$. In the worst case scenario, when the individual double sums have a significant cost as well, we expect scaling like $N_p I^3 P/N$.

3.3.1. Particle data structure

The field grid is partitioned in one dimension over the processes, as described in Section 3.2, such that each process owns a field slab. For each process, we use two arrays to store the data for particles included inside the corresponding slab. The first array contains state information, including the particle locations — required to perform the interpolation of the field. We

denote this first array *state*. The second array, called *rhs*, contains the value of the right-hand side of (8), as computed at the most recent few iterations (as required for the Adams-Bashforth integration); updating this second array requires interpolation. The two arrays use an array of structures pattern, in the sense that data associated to one particle is contiguous in memory. While this may lead to performance penalties, as pointed out in [46], there are significant benefits for our MPI parallel approach, as explained below. We summarize in the following the main operations that are applied to the arrays.

Ordering the particles locally. When $N > P$, processes are in charge of more than one field slice, and the particles in the slab are distributed across different slices. In this case, we store the particles that belong to the same slice contiguously in the arrays, one slice after the other in increasing x_3-axis order. This can be achieved by partitioning the arrays into P_s different groups (indexed by s in the following) and can be implemented as an incomplete Quicksort with a complexity of $O(P_p \log P_s)$ on average. The different groups will in general contain different numbers of particles, which implies that the associated data is stored at arbitrary locations within the contiguous *state* and *rhs* arrays. We therefore build an array *offset* of size $P_s + 1$, where *offset[s]* returns the starting index of the first particle for the group s and the number of particles in this same group s can be computed as *offset[s+1]-offset[s]*. For simplicity, we explicitly store *offset*$[P_s]$= P_p as the $P_s + 1$-th element of *offset*. This allows direct access to the contiguous data regions corresponding to each field slice, access which is in turn relevant for MPI exchanges (see below).

Exchanging the particles for computation. With our data structures, we are able to send the state information of all the particles located in a single group with only one communication, which reduces communication overhead. Moreover, sending the particles from several contiguous groups can also be done in a single operation because the groups are stored sequentially inside the arrays.

Particles displacement/update. The positions of the particles are updated at the end of each iteration, and so the arrays must be rearranged accordingly. The changes in the x_3 direction might move some particles to a different slice and even to a slice owned by a different process. Therefore, we first partition the first and last groups (the groups of the first and last slices of the process's slab) to move the particles that are now outside of the process's grid interval at the extremities of the arrays. We only act on the particles located at the lower and higher groups because we assume that the particles cannot move a distance greater than $2\pi/N$. For regular tracers (8) this is in fact required by the CFL stability condition of the fluid solver. This partitioning is done with a complexity $O(P_p/P_s)$. Then, every process exchanges those particles with its direct neighbors, ensuring that the particles are correctly distributed. Finally, each process sorts its particles to take into account the changes in the positions and the newly received particles as described previously.

3.3.2. Parallelization

The interpolation of the field at the particle locations concentrates most of the workload of the numerical particle tracking. For each particle, the interpolation uses the I^3 surrounding field nodes. However, because we do not mirror the particle or the field information on multiple processes, we must actively exchange either field or particle information to perform a complete interpolation. Assuming that the number of particles in the simulation is much smaller than the number of field nodes, i.e. the relation $P_p < IN^2$ holds, less data needs to be transferred on average when particle locations are exchanged rather than field values at required grid nodes. Consequently, in our implementation we exchange the particle information only.

A straightforward implementation in which the communication and computation are dissociated consists in the following operations:

(a) each process computes the interpolation of its particles on its field;

(b) all the processes exchange particle positions with their neighbors (each process sends and receives arrays of positions);

(c) each process computes the interpolation using its field on the particle positions it received from other processes in (b);

(d) all the processes exchange the results of the interpolations from (c) with the corresponding neighbors;

(e) each process merges the results it received in (d) and the results from its own computation from (a).

In our implementation, we interleave these five operations to overlap communication with computation. As we detail in the following, the master thread of each MPI process creates computation work packages, then performs communications while the other threads are already busy with the work packages. This is achieved with the use of non-blocking MPI communications and OpenMP tasks, as illustrated in Fig. 2. Broadly speaking, the execution is guided by a pairing of communications with relevant actions that need to be performed on the associated data (either sent or received), with all such pairs put together in a list $\mathcal{L}$ (`cap_list` in Fig. 2). Actions of "compute" type are turned into tasks distributed to all available threads, whereas actions of "send/receive" type are handled separately by the master thread. A more detailed overview follows. The figure shows a simplified version of the parallel execution pattern (the function `exchange_compute`), along with a possible execution timeline. For simplicity we also define the function `proceed`, which encompasses much of the communication and coordination activity of the master thread. Here we focus on describing the algorithm, only referring to the execution timeline implicitly through the color correspondence. In a first stage, the master thread splits the local interpolation from (a) into tasks and submits them immediately but with a low priority. Then, it posts all the sends/receives related to (b) and all the receives

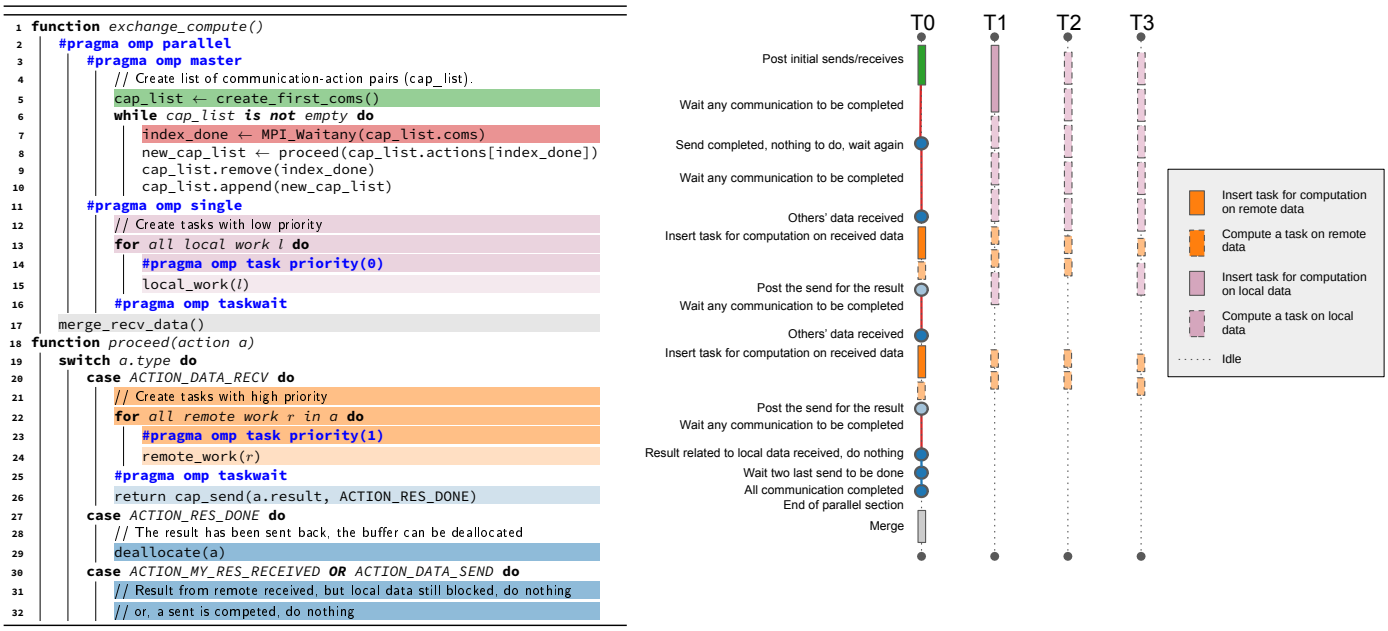

Figure 2: Overview of parallel interpolation algorithm (left) and reference execution timeline for one process with four threads (right). Colored lines of code correspond to individual timeline elements. The threads compute local operations by default but switch to remote operations when the master thread creates the corresponding new higher-priority tasks. With the use of priorities, the execution pattern allows for quicker communication of results to different processes. See text for details.

related to (d), and creates the list $\mathcal{L}$. In the core part of the algorithm, the master thread performs a wait-any on the list of MPI requests (line 7 in the pseudocode). This MPI function is blocking and returns as soon as one of the communications in the list is completed. Hence, when a communication is completed, the master thread acts accordingly to the type of action $\mathcal{A}$ it represents (`a.type` in Fig. 2). If $\mathcal{A}$ is the completion of a send of local particle positions, from (b), there is nothing to do and the master thread directly goes back to the wait-any. In this case, it means that a send is completed and that there is nothing new to do locally. If $\mathcal{A}$ is the completion of a receive of remote particle positions, from (b), then the master thread creates tasks to perform the interpolation of these positions, from (c), and submits them with high priority (lines 23 and 24 in the pseudocode). Setting a high priority ensures that all the threads will work on these tasks even if the tasks inserted earlier to interpolate the local positions, from (a), are not completed. When these tasks are completed, the master thread posts a non-blocking send to communicate the results to the process that owns the particles and adds the corresponding MPI request to $\mathcal{L}$. Then, the master thread goes back to the wait-any on $\mathcal{L}$. If $\mathcal{A}$ is the completion of a send of interpolation on received positions, as just described, the master thread has nothing to do and goes back to the wait-any on $\mathcal{L}$. In fact, this event simply means that the results were correctly sent. If $\mathcal{A}$ is the completion of a receive, from (d), of interpolation performed by another process, done in (c), the master thread keeps the buffer for merging at the end, and it goes back to the wait-any. When $\mathcal{L}$ is empty, it means that all communications (b,d) but also computations on other positions (c) are done. If some local work still remains from (a), the master thread can join it and compute some tasks. Finally, when all computation and communication are over, the threads can merge the interpolation results, operation (e) (line 17 in the pseudocode).

The described strategy is a parallel programming pattern that could be applied in many other contexts when there are local and remote works to perform and where remote work means first to exchange information and second to apply computation on it.

3.4. In-Order Parallel Particle I/O

Saving the states of the particles on disk is a crucial operation to support checkpoint/restart and for post-processing. We focus on output because TurTLE typically performs many more output than input operations (the latter only happen during initialization). The order in which the particles are saved is important because it influences the writing pattern and the data accesses during later post-processing of the files. As the particles move across the processes during the simulation, a naive output of the particles as they are distributed will lead to inconsistency from one output to the other. Such a structure would require reordering the particles during the post-processing or would result in complex file accesses. That is why we save the particles in order, i.e. in the original order given as input to the application.

The algorithm that we use to perform the write operation is shown in Fig. 3. There are four main steps to the procedure: *pre-sort* ("Sort" and "Find pivot" in the figure), followed by *exchange* ("Send/Recv" in Fig. 3), with a final *post-sort* ("Sort") before the actual HDF5 *write* ("Parallel write to file" in the figure).

Each process first sorts its local particles using the global indices with $O(P_p \log P_p)$ complexity. This sort can be done in

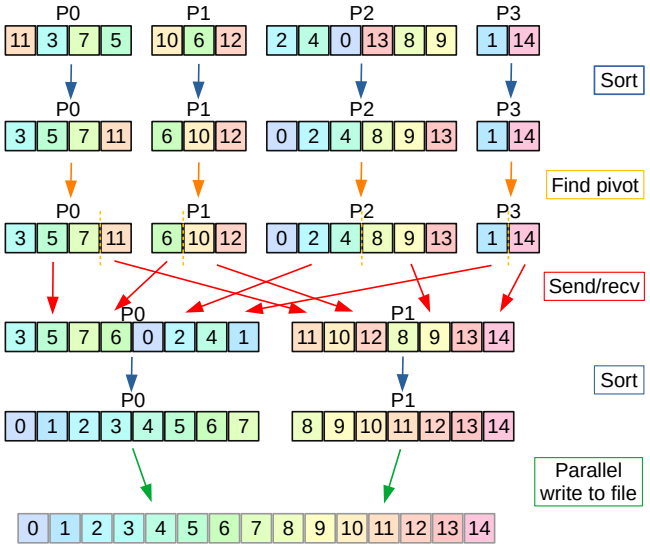

Figure 3: The different stages to perform a parallel saving of the particles in order. Here, we consider that the particle data (illustrated by the global particle index) is distributed among 4 processes, but that only 2 of them are used in the write operation.

parallel using multiple threads. Then, each process counts the number of particles it has to send to each of the processes that are involved in the file writing. These numbers are exchanged between the processes allowing each process to allocate the reception buffer. If we consider that P_O processes are involved in the output operation, each of them should receive N_p/P_O particles in total, and a process of rank r should receive the particles from index $r \times N_p/P_O$ to $(r+1) \times N_p/P_O - 1$.

In the *exchange* step, the particles can be sent either with multiple non-blocking send/receive operations or with a single all-to-all operation, with the total number of communications bounded by $P \times P_O$. Finally, the received particles are sorted with a complexity of $O(N_p/P_O \log N_p/P_O)$, and written in order into the output file.

The number P_O of processes involved in the writing should be carefully chosen because as P_O increases, the amount of data output per process decreases and might become so small that the write operation becomes inefficient. At the same time, the preceding exchange stage becomes more and more similar to a complete all-to-all communication with N_p^2 relatively small messages. On the other hand, as P_O decreases, the size of the messages exchanged will increase, and the write operation can eventually become too expensive for only a few processes, which could also run out of memory. This is why we heuristically fix P_O using three parameters: the minimum amount of data a process should write, the maximum number of processes involved in the write operation, and a chunk size. As N_p increases, P_O increases up to the given maximum. If N_p is large enough, the code simply ensures that $P_O - 1$ processes output the same amount of data (being a multiple of the chunk size), and the last process writes the remaining data. In our implementation, the parameters are chosen empirically (based on our experience with several HPC clusters running the IBM GPFS/SpectrumScale parallel file system), and they can be tuned for specific hardware configurations if necessary.

We use a similar procedure for reading the particle state: P_O processes read the data, they sort it according to spatial location, then they redistribute it to all MPI processes accordingly.

4. Computational performance

4.1. Hardware and software environment

To evaluate the computational performance of our approach, we perform benchmark simulations on the HPC cluster SuperMUC-NG from the Leibniz Supercomputing Centre (LRZ): we use up to 512 nodes containing two Intel Xeon Platinum 8174 (Skylake) processors with 24 cores each and a base clock frequency of 3.1 GHz, providing 96 GB of main memory. The network interconnect is an Intel OmniPath (100 Gbit/s) with a pruned fat-tree topology that enables non-blocking communications within islands of up to 788 nodes. We use the Intel compiler 19.0, Intel MPI 2019.4, HDF5 1.8.21 and FFTW 3.3.8. For our benchmarks, we always fully populate the nodes, i.e. the combination of MPI processes per node (ppn) and OpenMP threads per MPI process is chosen such that their product equals 48, and that the threads spawned by an MPI rank are confined within the NUMA domain defined by a single processor.

4.2. Overall performance

	N = 2048		N = 4096	
Number of nodes \ ppn	8	16	8	16
64	10.3	9.43	—	—
128	6.92	4.76	41.6	—
256	2.52	—	31.1	17.5
512	—	—	11.9	—

Table 1: Execution times for TurTLE on SuperMUC-NG (in seconds), various configurations (values correspond to Fig. 4a).

Figure 4 provides an overview of the overall parallel scaling behavior of the code for a few typical large-scale setups (panel a as well as Table 1) together with a breakdown into the main algorithmic steps (panel b). We use the execution time for a single time step (averaged over a few steps) as the primary performance metric and all data and computations are handled in double precision. The left panel shows, for two different setups ($N = 2048, 4096$), that the code exhibits excellent strong-scaling efficiency (the dashed line represents ideal scaling) from the minimum number of nodes required to fit the code into memory up to the upper limit which is given by the maximum number of MPI processes that can be utilized with our one-dimensional domain decomposition. The total runtime of TurTLE, and hence its major parallel scaling properties are largely determined by the three-dimensional fast Fourier transforms

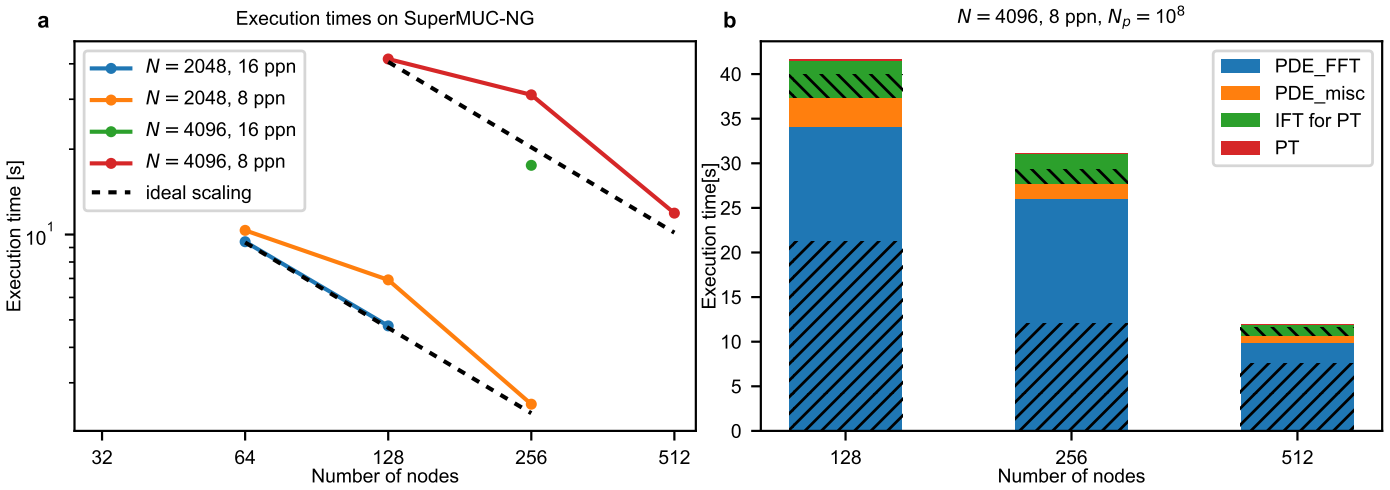

Figure 4: Computational performance of TurTLE. Strong scaling behavior of the total runtime (a) for grids with $N = 2048$ and $N = 4096$, respectively, and using 8 or 16 MPI processes per node (ppn) on up to 512 fully populated nodes (24576 cores) of SuperMUC-NG ($I = 8$, $N_p = 10^8$). For $N = 4096$ and 8 ppn panel (b) shows a breakdown of the total runtime into the main functional elements, namely solving the system of Navier-Stokes partial differential equations ("PDE_misc" and "PDE_FFT") and particle tracking ("IFT for PT" and "PT"). The fluid solver cost is largely dominated by the fast Fourier transforms ("PDE_FFT"). The cost of particle tracking for 10^8 particles (with $I = 8$) is determined by the additional inverse Fourier transform ("IFT for PT"), whereas the runtime for our novel particle tracking algorithm ("PT") is still negligible for 10^8 particles. Hatched regions represent the fraction of MPI communication times.

(using FFTW) for solving the system of Navier Stokes partial differential equations, as shown in Fig. 4b (labeled "PDE_FFT", entire blue area) for the example of the large setup ($N = 4096$) with 8 processes per node (see also Table 2). With increasing node count, the runtime of the FFTs in turn gets increasingly dominated by an all-to-all type of MPI communication pattern which is arising from the FFTW-internal global transpositions (blue-hatched area) of the slab-decomposed data.

Algorithm unit \ Number of nodes	128	256	512
PDE_FFT	34 (62%)	26 (46%)	9.9 (76%)
PDE_misc	3.3 (-%)	1.7 (-%)	0.74 (-%)
IFT for PT	4.1 (62%)	3.3 (46%)	1.2 (76%)
PT	0.13 (61%)	0.07 (55%)	0.043 (67%)

Table 2: Breakdown of the total TurTLE runtime (in seconds) into main functional elements (values correspond to Fig. 4b).

While a good OpenMP efficiency can be noted for the case of 64 nodes by comparing the blue and the orange curve in Fig. 4a, i.e. the same problem computed with a different combination of MPI processes per node (8, 16) and a corresponding number of OpenMP threads per process (6, 3), we generally observe FFTW exhibiting limited OpenMP scaling beyond a number of 6 to 8 threads per MPI process for the grid resolutions (N) considered here. Moreover, even at moderate thread counts, FFTW, due to internal load-imbalances, fails to efficiently handle the process-local transforms for certain dimensions of the local data slabs: In the case of 256 nodes, FFTW apparently cannot efficiently use more than 3 OpenMP threads for parallelizing over the local slabs of dimension $2 \times 4096 \times 2049$ (Fourier representation), whereas good scaling up to the desired maximum of 6 threads is observed for a dimension of $8 \times 4096 \times 2049$ (128 nodes) and also $1 \times 4096 \times 2049$ (512 nodes). The same argument applies for the smaller setup ($N = 2048$) on 128 nodes and is the root cause for the kink in the strong scaling curves of panel a. We plan for TurTLE to support FFTs also from the Intel Math Kernel Library (MKL) which are expected to deliver improved threading efficiency. For practical applications, this is less of a concern, since a user can perform a few exploratory benchmarks for a given setup of the DNS on the particular computational platform, and available node counts, in order to find an optimal combination of MPI processes and OpenMP threads. Since the runtime per time step is constant for our implementation of the Navier-Stokes solver, a few time steps are sufficient for tuning a long-running DNS.

Thanks to our efficient and highly scalable implementation of the particle tracking, its contribution to the total runtime is barely noticeable in the figure ("PT", red colour in Fig. 4b). This holds even for significantly larger numbers of particles than the value of $N_p = 10^8$ which was used here (see below for an in-depth analysis). The only noticeable additional cost for particle tracking, amounting to roughly 10% of the total runtime, comes from an additional inverse FFT ("IFT for PT", green colour) which is required to compute the advecting velocity field, which is independent of N_p and scales very well.

Finally, Fig. 4a also suggests good weak scaling behavior of TurTLE: When increasing the resolution from $N = 2048$ to $N = 4096$ and at the same time increasing the number of nodes from 64 to 512, the runtime increases from 10.35s to 11.45s, which is consistent with a $O(N^3 \log N)$ scaling of the runtime, given the dominance of the FFTs.

4.3. *Particle tracking performance*

Fig. 5 provides an overview and some details of the performance of our particle tracking algorithm, extending the assess-

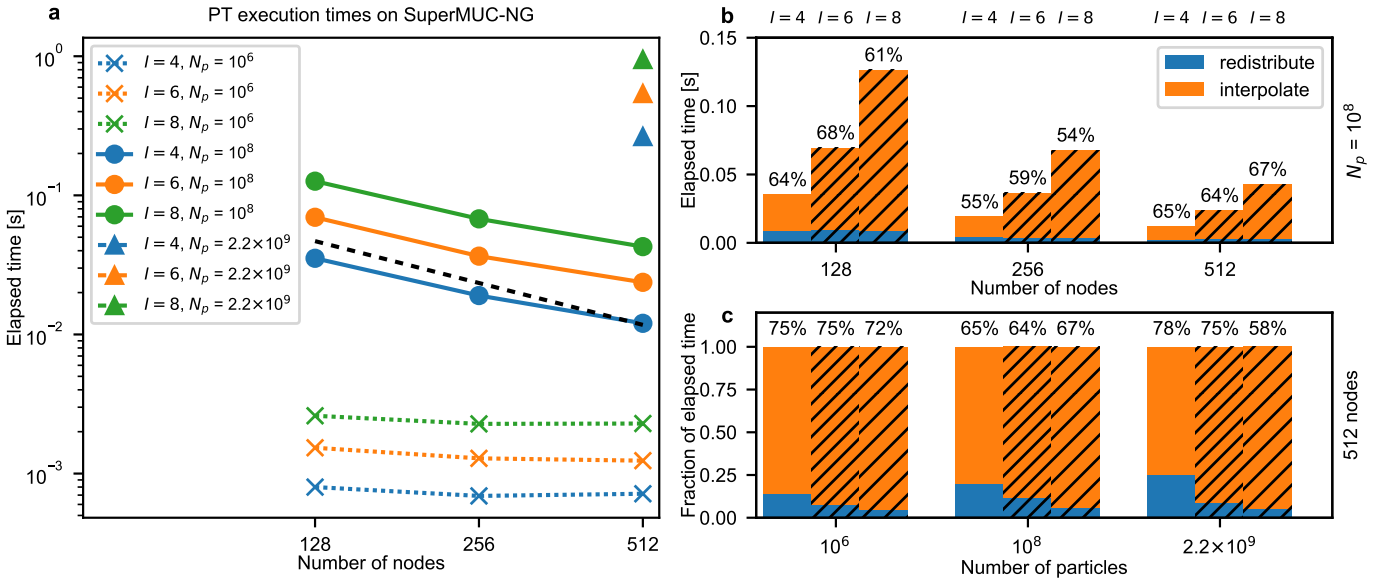

Figure 5: Computational performance of the particle tracking code using 8 MPI processes per node and a DNS of size $N = 4096$, for different sizes of the interpolation kernel I. Panel (a): strong scaling for different numbers of particles N_p and sizes of the interpolation kernel (memory requirements limit the $N_p = 2.2 \times 10^9$ case to 512 nodes). The dashed line corresponds to ideal strong scaling. Panel (b): contributions of interpolation and redistribution operations to the total execution time, for a fixed number of particles, $N_p = 10^8$, and for different sizes of the interpolation kernel (the corresponding vertical bars are distinguished by hatch style, see labels on top) as a function of the number of compute nodes. Panel (c): relative contributions of interpolation and redistribution as a function of N_p. Percentages represent the fraction of time spent in MPI calls.

ment of the previous subsection to particle numbers beyond the current state of the art [23]. We use the same setup of a DNS with $N = 4096$ and 8 MPI processes per node on SuperMUC-NG, as presented in the previous subsection.

Fig. 5a summarizes the strong-scaling behavior on 128, 256 or 512 nodes of SuperMUC-NG for different numbers of particles (10^6, 10^8 and 2.2×10^9) and for different sizes of the interpolation kernel I (4, 6, 8). Most importantly, the absolute run times are small compared to the fluid solver: Even when using the most accurate interpolation kernel, a number of 2.2×10^9 particles can be handled within less than a second (per time step), i.e. less than 10% to the total computational cost of TurTLE on 512 nodes per time step (cf. Fig. 4).

The case of $N_p = 10^6$ is shown only for reference here. This number of particles is too small to expect good scalability in the regime of 128 compute nodes and more. Still, the absolute runtimes are minuscule compared to a DNS of typical size. For $N_p = 10^8$ we observe good but not perfect strong scaling, in particular for the largest interpolation kernel ($I = 8$), suggesting that we observe the $N_p I P/N$ regime, as discussed previously. It is worth mentioning that we observe a sub-linear scaling of the total runtime with the total number of particles (Fig. 5a).

Fig. 5b shows a breakdown of the total runtime of the particle tracking algorithm into its main parts, interpolation (operations detailed in Fig. 2, shown in orange) and redistribution (local sorting of particles together with particle exchange, blue), together with the percentage of time spent in MPI calls. The latter takes between half and two thirds of the total runtime for $N_p = 10^8$ particles (cf. upper panel b) and reaches almost 80% for $N_p = 2.2 \times 10^9$ particles on 512 nodes (lower panel c). Overall, the interpolation cost always dominates over redistribution, and increases with the size of the interpolation kernel roughly as I^2, i.e. the interpolation cost is proportional to the number of MPI messages required by the algorithm (as detailed above).

4.4. Particle output

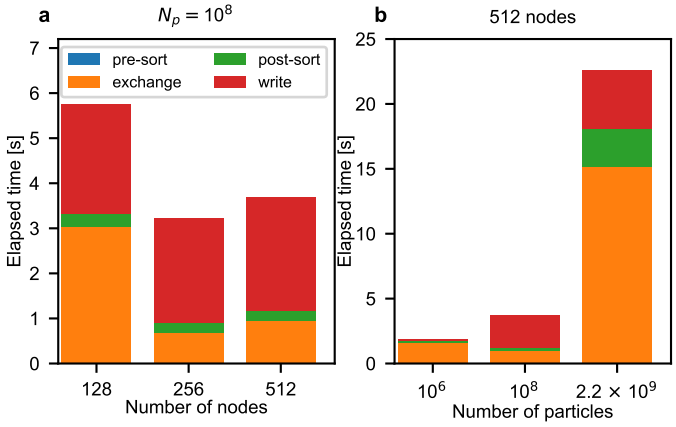

Figure 6: Performance of particle output, as distributed between the four different operations (see also Fig. 3): *pre-sort* (particles are sorted by each process), MPI *exchange* (particle data is transferred to processes actually participating in I/O), *post-sort* (particles are sorted on each I/O process), and *write* (HDF5 write call). Panel (a): elapsed times as a function of the total number of nodes, for a fixed $N_p = 10^8$ (see also Fig. 5b). Panel (b): elapsed time as a function of the number of particles, in the case of 512 nodes (see also Fig. 5c).

Figure 6 provides an overview of the computational costs of the main parts of the output algorithm, namely sorting particles according to their initial order (*pre-sort* and *post-sort* stages,

cf. Sect. 3.4), communicating the data between processes (*exchange* stage), and writing data to disk using parallel HDF5 (*write* stage). Here, the same setup is used as in Fig. 5 panels b and c, respectively, noting that the output algorithm does not depend on the size of the interpolation kernel. The figure shows that the total time is largely dominated by the *write* and *exchange* stages, with the sorting stages not being significant. Of the latter, the *post-sort* operation is relatively more expensive than the *pre-sort* stage, because only a comparably small subset of $P_O < P$ processes is used in the *post-sort* stage (in the present setup $P_O = 1$ for 10^6 particles, $P_O = 72$ for 10^8 particles, and $P_O = 126$ for 2.2×10^9 particles were used). This indicates that our strategy of dumping the particle data in order adds only a small overhead, which is mostly spent in the communication stage (unsorted output could be done with a more simple communication pattern) but not for the actual (process-local) reordering of the particles. For a given number of particles N_p, the number of processes P_O involved in the *write* operation is fixed, independent of the total number P of processes used for the simulation. Consequently, the time spent in the *write* stage does not depend on the number of nodes (and hence P), as shown in Fig. 6a. However, P_O may increase with increasing N_p (and fixed P).

Fig. 6b shows that the cost of writing 10^6 particles with a single process is negligible, whereas writing 10^8 particles with 72 processes becomes significant, even though a similar number of particles per output process (1.4×10^6 particles) is used. This reflects the influence of the (synchronization) overhead of the parallel HDF5 layer and the underlying parallel IO system. On the other hand, it takes about the same amount of time for 126 processes to write 1.7×10^7 particles each, compared with 72 processes writing 1.4×10^6 particles each, which motivates our strategy of controlling the number of processes P_O that are involved in the interaction with the IO system. However, the choice of P_O also influences the communication time spent in the *exchange* stage. When looking at the *exchange* stage in Figure 6a, we recall that 72 processes write the data for all three node counts. As P increases, the 72 processes receive less data per message but communicate with more processes. From these results it appears that this is beneficial: reducing the size of the messages but increasing the number of processes that communicate reduces the overall duration of this operation (that we do not control explicitly since we rely on the `MPI_Alltoallv` collective routine). For a fixed number of processes and an increasing number of particles (see Figure 6b), the total amount of data exchanged increases and the size of the messages varies. The number P_O (i.e., 1, 72 and 126) is not increased proportionally with the number of particles N_p (i.e., 10^6, 10^8 and 2.2×10^9), which means that the messages get larger and, more importantly, each process needs to send data to more output processes. Therefore, increasing P_O also increases the cost of the *exchange* stage but allows to control the cost of *write* stage. Specifically, it takes about 4s to output 10^8 particles (1s for *exchange* and 3s for *write*). It takes only 6 times longer, about 23s (15s for *exchange*, 4s for *write*, and 3s *post-sort*) to output 22 times more, 2.2×10^9, particles.

Overall, our strategy of choosing the number of processes P_O participating in the IO operations independent of the total number P of processes allows us to avoid performance-critical situations where too many processes would access the IO system, or too many processes would write small pieces of data. The coefficients used to set P_O can be adapted to the specific properties (hardware and software stack) of an HPC system.

5. Summary and conclusions

In the context of numerical studies of turbulence, we have presented a novel particle tracking algorithm using an MPI/OpenMP hybrid programming paradigm. The implementation is part of TurTLE, which uses a standard pseudo-spectral approach for the direct numerical simulation of turbulence in a 3D periodic domain. TurTLE succeeds at tracking billions of particles with a negligible cost relative to solving the fluid equations. MPI communications are overlapped with computation thanks to a parallel programming pattern that mixes OpenMP tasks and MPI non-blocking communications. At the same time, the use of a contiguous and slice-ordered particle data storage allows to minimize the number of required MPI messages for any size of the interpolation kernel. This way, our approach combines both numerical accuracy and computational performance to address open questions regarding particle-laden flows by performing highly resolved numerical simulations on large supercomputers[36, 62, 63, 64]. Indeed, as the benchmarks presented in Section 4 have shown, TurTLE scales up to many thousands of CPU cores on modern high-performance computers.

We expect that due to our task-based parallelization and the asynchronous communication scheme the particle-tracking algorithm is also well suited for offloading to massively parallel accelerators (e.g. GPUs) of a heterogeneous HPC node architecture. Concerning the fluid solver, the pseudo-spectral approach taken by TurTLE appears in principle amenable to GPU acceleration, as demonstrated by similar fluid codes (e.g. [65, 66, 67, 68]). Whether it can be implemented in a (performance-) portable and scalable way on large GPU-accelerated HPC systems remains to be investigated.

6. Acknowledgments

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). Some computations were also performed at the Max Planck Computing and Data Facility. This work was supported by the Max Planck Society. This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001081).

References

[1] G. I. Taylor, Diffusion by continuous movements, P. Lond. Math. Soc. s2-20 (1) (1922) 196–212 (1922). doi:10.1112/plms/s2-20.1.196.

[2] A. N. Kolmogorov, The local structure of turbulence in incompressible viscous fluid for very large Reynolds numbers, Proc. R. Soc. London, Ser. A 434 (1890) (1991) 9–13 (1991). doi:10.1098/rspa.1991.0075.

[3] P. K. Yeung, S. B. Pope, Lagrangian statistics from direct numerical simulations of isotropic turbulence, J. Fluid Mech. 207 (1989) 531 (1989). doi:10.1017/S0022112089002697.

[4] P. K. Yeung, Lagrangian investigations of turbulence, Annu. Rev. Fluid Mech. 34 (1) (2002) 115–142 (2002). doi:10.1146/annurev.fluid.34.082101.170725.

[5] F. Toschi, E. Bodenschatz, Lagrangian Properties of Particles in Turbulence, Annu. Rev. Fluid Mech. 41 (1) (2009) 375–404 (2009). doi:10.1146/annurev.fluid.010908.165210.

[6] H. Homann, J. Bec, H. Fichtner, R. Grauer, Clustering of passive impurities in magnetohydrodynamic turbulence, Phys. Plasmas 16 (8) (2009) 082308 (2009). doi:10.1063/1.3204100.

[7] A. Stohl, S. Eckhardt, C. Forster, P. James, N. Spichtinger, P. Seibert, A replacement for simple back trajectory calculations in the interpretation of atmospheric trace substance measurements, Atmos. Environ. 36 (29) (2002) 4635 – 4648 (2002). doi:10.1016/S1352-2310(02)00416-8.

[8] A. Stohl, Characteristics of atmospheric transport into the arctic troposphere, J. Geophys. Res.-Atmos. 111 (D11) (2006). doi:10.1029/2005JD006888.

[9] E. Behrens, F. U. Schwarzkopf, J. F. Lübbecke, C. W. Böning, Model simulations on the long-term dispersal of 137 Cs released into the Pacific Ocean off Fukushima, Environ. Res. Lett. 7 (3) (2012) 034004 (2012).

[10] T. Haszpra, I. Lagzi, T. Tél, Dispersion of aerosol particles in the free atmosphere using ensemble forecasts, Nonlinear Proc. Geoph. 20 (5) (2013) 759–770 (2013). doi:10.5194/npg-20-759-2013.

[11] R. A. Shaw, Particle-turbulence interactions in atmospheric clouds, Annu. Rev. Fluid Mech. 35 (2003) 183–227 (2003). doi:10.1146/annurev.fluid.35.101101.161125.

[12] E. Bodenschatz, S. P. Malinowski, R. A. Shaw, F. Stratmann, Can we understand clouds without turbulence?, Science 327 (5968) (2010) 970–971 (2010). doi:10.1126/science.1185138.

[13] B. J. Devenish, P. Bartello, J.-L. Brenguier, L. R. Collins, W. W. Grabowski, R. H. A. IJzermans, S. P. Malinowski, M. W. Reeks, J. C. Vassilicos, L.-P. Wang, Z. Warhaft, Droplet growth in warm turbulent clouds, Q. J. Roy. Meteor. Soc. 138 (667) (2012) 1401–1429 (2012). doi:10.1002/qj.1897.

[14] W. W. Grabowski, L.-P. Wang, Growth of cloud droplets in a turbulent environment, Annu. Rev. Fluid Mech. 45 (1) (2013) 293–324 (2013). doi:10.1146/annurev-fluid-011212-140750.

[15] A. Pumir, M. Wilkinson, Collisional aggregation due to turbulence, Annu. Rev. Condens. Matter Phys. 7 (1) (2016) 141–170 (2016). doi:10.1146/annurev-conmatphys-031115-011538.

[16] W. M. Durham, E. Climent, M. Barry, F. de Lillo, G. Boffetta, M. Cencini, R. Stocker, Turbulence drives microscale patches of motile phytoplankton, Nat. Commun. 4 (2013) 2148 (2013). doi:10.1038/ncomms3148.

[17] R. E. Breier, C. C. Lalescu, D. Waas, M. Wilczek, M. G. Mazza, Emergence of phytoplankton patchiness at small scales in mild turbulence, Proc. Natl. Acad. Sci. U.S.A. 115 (48) (2018) 12112–12117 (2018). doi:10.1073/pnas.1808711115.

[18] N. Pujara, M. A. R. Koehl, E. A. Variano, Rotations and accumulation of ellipsoidal microswimmers in isotropic turbulence, J. Fluid Mech. 838 (2018) 356–368 (2018). doi:10.1017/jfm.2017.912.

[19] S. A. Orszag, G. S. Patterson, Numerical simulation of three-dimensional homogeneous isotropic turbulence, Phys. Rev. Lett. 28 (1972) 76–79 (1972). doi:10.1103/PhysRevLett.28.76.

[20] P. K. Yeung, S. B. Pope, An algorithm for tracking fluid particles in numerical simulations of homogeneous turbulence, J. Comput. Phys. 79 (2) (1988) 373 – 416 (1988). doi:10.1016/0021-9991(88)90022-8.

[21] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, Y. Kaneda, 16.4-tflops direct numerical simulation of turbulence by a Fourier spectral method on the earth simulator, in: Supercomputing, ACM/IEEE 2002 Conference, 2002, p. 50–50 (2002).

[22] T. Ishihara, K. Morishita, M. Yokokawa, A. Uno, Y. Kaneda, Energy spectrum in high-resolution direct numerical simulations of turbulence, Phys. Rev. Fluids 1 (2016) 082403 (2016). doi:10.1103/PhysRevFluids.1.082403.

[23] D. Buaria, P. K. Yeung, A highly scalable particle tracking algorithm using partitioned global address space (pgas) programming for extreme-scale turbulence simulations, Comput. Phys. Commun. 221 (2017) 246–258 (2017). doi:10.1016/j.cpc.2017.08.022.

[24] S. B. Pope, Turbulent Flows, Cambridge University Press, 2000 (2000).

[25] P. K. Yeung, K. R. Sreenivasan, S. B. Pope, Effects of finite spatial and temporal resolution in direct numerical simulations of incompressible isotropic turbulence, Phys. Rev. Fluids 3 (2018) 064603 (2018). doi:10.1103/PhysRevFluids.3.064603.

[26] T. Ishihara, Y. Kaneda, M. Yokokawa, K. Itakura, A. Uno, Small-scale statistics in high-resolution direct numerical simulation of turbulence: Reynolds number dependence of one-point velocity gradient statistics, J. Fluid Mech. 592 (2007) 335–366 (2007). doi:10.1017/S0022112007008531.

[27] P. K. Yeung, X. M. Zhai, K. R. Sreenivasan, Extreme events in computational turbulence, Proc. Natl. Acad. Sci. U.S.A. 112 (41) (2015) 12633–12638 (2015). doi:10.1073/pnas.1517368112.

[28] C. Küchler, G. Bewley, E. Bodenschatz, Experimental study of the bottleneck in fully developed turbulence, J. Stat. Phys. (2019). doi:10.1007/s10955-019-02251-1.

[29] Z. Warhaft, Turbulence in nature and in the laboratory, Proc. Natl. Acad. Sci. U.S.A. 99 (suppl 1) (2002) 2481–2486 (2002). doi:10.1073/pnas.012580299.

[30] L. Biferale, G. Boffetta, A. Celani, B. J. Devenish, A. Lanotte, F. Toschi, Multifractal statistics of Lagrangian velocity and acceleration in turbulence, Phys. Rev. Lett. 93 (2004) 064502 (2004). doi:10.1103/PhysRevLett.93.064502.

[31] G. L. Eyink, Stochastic flux freezing and magnetic dynamo, Phys. Rev. E 83 (2011) 056405 (2011). doi:10.1103/PhysRevE.83.056405.

[32] G. Eyink, E. Vishniac, C. Lalescu, H. Aluie, K. Kanov, K. Bürger, R. Burns, C. Meneveau, A. Szalay, Flux-freezing breakdown in high-conductivity magnetohydrodynamic turbulence, Nature 497 (7450) (2013) 466–469 (2013). doi:10.1038/nature12128.

[33] L. Biferale, A. Lanotte, R. Scatamacchia, F. Toschi, Intermittency in the relative separations of tracers and of heavy particles in turbulent flows, J. Fluid Mech. 757 (2014) 550–572 (2014). doi:10.1017/jfm.2014.515.

[34] P. L. Johnson, C. Meneveau, Large-deviation joint statistics of the finite-time Lyapunov spectrum in isotropic turbulence, Phys. Fluids 27 (8) (2015) 085110 (2015). doi:10.1063/1.4928699.

[35] C. C. Lalescu, M. Wilczek, Acceleration statistics of tracer particles in filtered turbulent fields, J. Fluid Mech. 847 (2018) R2 (2018). doi:10.1017/jfm.2018.381.

[36] C. C. Lalescu, M. Wilczek, How tracer particles sample the complexity of turbulence, New J. Phys. 20 (1) (2018) 013001 (2018). doi:10.1088/1367-2630/aa8ecd.

[37] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd Edition, Scientific And Engineering Computation Series, MIT Press, 1999 (1999).

[38] O. A. R. Board, OpenMP application program interface version 4.5 (2015).
URL http://www.openmp.org/mp-documents/openmp-4.5.pdf

[39] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, D. Told, The global version of the gyrokinetic turbulence code gene, J. Comput. Phys. 230 (18) (2011) 7053–7071 (2011). doi:10.1016/j.jcp.2011.05.034.

[40] P. D. Mininni, D. Rosenberg, R. Reddy, A. Pouquet, A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence, Parallel Comput. 37 (6) (2011) 316 – 326 (2011). doi:10.1016/j.parco.2011.05.004.

[41] D. Pekurovsky, P3DFFT: a framework for parallel computations of Fourier transforms in three dimensions, SIAM J. Sci. Comput. 34 (4) (2012) C192–C209 (2012).

[42] D. Pekurovsky, P3dfft v. 2.7.9 (2019). doi:10.5281/zenodo.2634590.
URL https://doi.org/10.5281/zenodo.2634590

[43] M. P. Clay, D. Buaria, T. Gotoh, P. K. Yeung, A dual communicator and dual grid-resolution algorithm for petascale simulations of turbulent mixing at high schmidt number, Comput. Phys. Commun. 219 (2017) 313–328 (2017). doi:10.1016/j.cpc.2017.06.009.

[44] A. G. Chatterjee, M. K. Verma, A. Kumar, R. Samtaney, B. Hadri, R. Khurram, Scaling of a Fast Fourier Transform and a Pseudo-spectral Fluid Solver up to 196608 cores, J. Parallel Distr. Com. 113 (2018) 77–91 (2018). doi:10.1016/j.jpdc.2017.10.014.

[45] D. W. Walker, J. J. Dongarra, MPI: A standard message passing interface, Supercomputer 12 (1) (1996) 56–68 (1996).
[46] H. Homann, F. Laenen, SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes, Comput. Phys. Commun. 224 (2018) 325 – 332 (2018). doi:10.1016/j.cpc.2017.11.015.
[47] TurTLE source code, https://gitlab.mpcdf.mpg.de/TurTLE/turtle.
[48] TurTLE documentation, http://turtle.pages.mpcdf.de/turtle/.
[49] H. Homann, Lagrangian statistics of turbulent flows in fluids and plasmas, Ph.D. thesis, Ruhr-Universität Bochum (2007).
[50] M. Wilczek, Statistical and numerical investigations of fluid turbulence, Ph.D. thesis, Westfälische Wilhelms-Universität Münster (2011). doi:10.17617/2.3075289.
[51] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock-capturing schemes, J. Comput. Phys. 77 (2) (1988) 439–471 (1988). doi:10.1016/0021-9991(88)90177-5.
[52] R. Courant, K. Friedrichs, H. Lewy, On the Partial Difference Equations of Mathematical Physics, IBM J. Res. Dev. 11 (1967) 215–234 (1967). doi:10.1147/rd.112.0215.
[53] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang, Spectral Methods in Fluid Dynamics, corrected third printing Edition, Springer-Verlag Berlin Heidelberg, 1988 (1988). doi:10.1007/978-3-642-84108-8.
[54] T. Y. Hou, R. Li, Computing nearly singular solutions using pseudo-spectral methods, J. Comput. Phys. 226 (2007) 379–397 (2007). doi:10.1016/j.jcp.2007.04.014.
[55] M. Frigo, S. G. Johnson, The design and implementation of FFTW3, P. IEEE 93 (2) (2005) 216–231, special issue on "Program Generation, Optimization, and Platform Adaptation" (2005).
[56] K. E. Atkinson, An Introduction to Numerical Analysis, 2nd Edition, John Wiley & Sons, Inc., 1989 (1989).
[57] C. C. Lalescu, B. Teaca, D. Carati, Implementation of high order spline interpolations for tracking test particles in discretized fields, J. Comput. Phys. 229 (17) (2010) 5862 – 5869 (2010). doi:10.1016/j.jcp.2009.10.046.
[58] F. Lekien, J. Marsden, Tricubic interpolation in three dimensions, Int. J. Numer. Meth. Eng. 63 (3) (2005) 455–471 (2005). doi:10.1002/nme.1296.
[59] H. Homann, J. Dreher, R. Grauer, Impact of the floating-point precision and interpolation scheme on the results of dns of turbulence by pseudo-spectral codes, Comput. Phys. Commun. 177 (2007) 560–565 (2007). doi:10.1016/j.cpc.2007.05.019.
[60] M. A. T. van Hinsberg, J. H. M. Thije Boonkkamp, F. Toschi, H. J. H. Clercx, On the efficiency and accuracy of interpolation methods for spectral codes, SIAM J. Sci. Comput. 34 (4) (2012) B479–B498 (2012). doi:10.1137/110849018.
[61] M. A. T. van Hinsberg, J. H. M. t. T. Boonkkamp, F. Toschi, H. J. H. Clercx, Optimal interpolation schemes for particle tracking in turbulence, Phys. Rev. E 87 (2013) 043307 (2013). doi:10.1103/PhysRevE.87.043307.
[62] L. Bentkamp, C. Lalescu, M. Wilczek, Persistent accelerations disentangle lagrangian turbulence, Nat. Commun. 10 (1) (2019) 3550 (2019). doi:10.1038/s41467-019-11060-9.
[63] N. Pujara, J.-A. Arguedas-Leiva, C. C. Lalescu, B. Bramas, M. Wilczek, Shape- and scale-dependent coupling between spheroids and velocity gradients in turbulence, J. Fluid Mech. 922 (2021) R6 (2021). doi:10.1017/jfm.2021.543.
[64] L. Bentkamp, T. D. Drivas, C. C. Lalescu, M. Wilczek, The statistical geometry of material loops in turbulence, Nat. Commun. 13 (2022) 2088 (2022). doi:10.1038/s41467-022-29422-1.
[65] R. Mukherjee, R. Ganesh, V. Saini, U. Maurya, N. Vydyanathan, B. Sharma, Three dimensional pseudo-spectral compressible magneto-hydrodynamic GPU code for astrophysical plasma simulation, in: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), 2018, pp. 46–55 (2018). doi:10.1109/HiPCW.2018.8634104.
[66] K. Ravikumar, D. Appelhans, P. K. Yeung, GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2019).
[67] J. M. López, D. Feldmann, M. Rampp, A. Vela-Martín, L. Shi, M. Avila, nsCouette – A high-performance code for direct numerical simulations of turbulent Taylor–Couette flow, SoftwareX 11 (2020) 100395 (2020). doi:10.1016/j.softx.2019.100395.
[68] D. Rosenberg, P. D. Mininni, R. Reddy, A. Pouquet, GPU parallelization of a hybrid pseudospectral geophysical turbulence framework using CUDA, Atmosphere 11 (2) (2020). doi:10.3390/atmos11020178.