LCS.jl v1.0: A High-Performance, Multi-Platform Computational Model in Julia for Turbulent Particle-Laden Flows

Taketo Tominaga[1] and Ryo Onishi[2]

[1]Department of Mechanical Engineering, School of Engineering, Institute of Science Tokyo, Tokyo, Japan
[2]Supercomputing Research Center, Institute of Integrated Research, Institute of Science Tokyo, Tokyo, Japan

Correspondence: Taketo Tominaga (tominaga.t.2d38@m.isct.ac.jp)

Abstract. Multiphase turbulent flow phenomena are observed not only in industrial devices but also in environmental flows, and direct numerical simulation plays a key role in their investigation. Many numerical models have been developed; nevertheless, few models are highly optimized for GPU platforms, which represent the current mainstream in high-performance computing. In this study, we developed LCS.jl (Lagrangian Cloud Simulator in Julia), a single-source and multi-platform multiphase turbulence simulation model implemented in the Julia language and KernelAbstractions.jl. Validation results confirmed that the present fluid and particle statistics agree well with those obtained in prior studies. For GPU execution, the GPU-side prefix-scan implementation reduced the particle communication cost by approximately 88% relative to the CPU-delegated sequential implementation. LCS.jl achieved CPU performance comparable to the original Fortran implementation, with similar intra-node scaling behavior. For GPUs, strong-scaling efficiency was approximately 70% up to 256 GPUs, and weak-scaling efficiency was approximately 80% up to 216 GPUs on TSUBAME4.0 (a GPU supercomputer at the Institute of Science Tokyo). LCS.jl achieved a maximum estimated speedup of approximately 16.3× on GPUs over CPUs. Execution on both NVIDIA and AMD GPUs further supported portability across GPU architectures. A trial heterogeneous execution achieved an approximately 68% reduction in execution time compared to the CPU-only configuration even in configurations where the GPU was not the primary compute device. These results demonstrate that LCS.jl is a multiphase turbulence simulation platform that achieves both portability and scalability across a variety of computational resource configurations.

1 Introduction

Multiphase turbulent flow phenomena are observed not only in industrial devices but also in environmental flows. In environmental flows, one example is the growth of cloud droplets in turbulent clouds such as cumulus and cumulonimbus.

Understanding this process is relevant for reliable weather and climate modeling. In turbulent flows, small inertial particles such as cloud droplets cluster, i.e., become non-uniformly distributed in space. It has been recognized that this clustering significantly affects the collision-coalescence growth of cloud droplets. Extensive direct numerical simulation (DNS) studies have been conducted to model the effect of clustering (Sundaram and Collins, 1997; Wang et al., 2000; Ireland et al., 2016).

Environmental flows are characterized by high Reynolds numbers. DNS of such flows requires spatial resolutions down to the Kolmogorov scale (Kolmogorov, 1941; Pope, 2000). The number of computational grid points increases rapidly with

increasing Reynolds number, increasing computational cost. In flow–particle simulations, such as droplet growth in turbulent environments, the particle solver introduces additional computational cost. Consequently, conducting multiphase DNS for environmental applications remains computationally demanding even on modern HPC systems. Onishi et al. (2015) and Onishi and Seifert (2016) addressed this challenge and clarified the Reynolds number dependence of the collision kernel through large-scale computations. However, evaluation under even higher Reynolds number conditions remains computationally prohibitive. Many numerical models have been developed to enable high-Reynolds-number computations. Nevertheless, few models are highly optimized for GPU platforms, which represent the current mainstream in high-performance computing (HPC). In anticipation of future architectural diversification, ensuring multi-platform compatibility, i.e., performance portability across different architectures, is becoming increasingly important.

The Euler–Lagrangian framework is a widely used approach for multiphase DNS that tracks inertial particles in turbulent flows. One implementation of this framework is the Lagrangian Cloud Simulator (LCS), a Fortran model developed by Onishi et al. (2015). LCS adopts distributed-memory parallelization via the Message Passing Interface (MPI) and has achieved large-scale parallel computation in HPC environments. However, conventional multiphase DNS models in Fortran or C have been highly optimized for multi-CPU environments. Supporting multi-GPU architectures while ensuring high performance in both fluid and particle solvers is a critical task for such models.

In the LCS framework, the computational domain is decomposed into three-dimensional Cartesian subdomains, each owned by one MPI process. Neighboring processes exchange halo data for the Eulerian flow solver, while particles crossing a subdomain boundary are transferred to the corresponding neighboring process.

Two challenges arise when porting such models to GPUs. The first concerns the particle communication algorithm. In fluid computation, the communication range can be determined statically by the domain decomposition. In particle computation, by contrast, the number and destinations of particles migrating across subdomain boundaries vary in time. Compared with tracer particles, inertial particles carry additional dynamical state, including particle velocity, that must also be dynamically packed and communicated during migration. The original LCS implementation used a sequential counter-based packing procedure to determine storage positions for migrating particles. Because the storage position of each particle depended on the counter value resulting from previously processed particles, this procedure introduced a sequential dependency that limited direct parallelization. The second challenge concerns performance portability. The development of a multi-platform model that achieves high performance on both CPUs and GPUs is a non-trivial task. In recent years, Julia has attracted attention in the HPC community as a language that achieves C/Fortran-level execution performance and high expressiveness through just-in-time (JIT) compilation and type specialization (Bezanson et al., 2017). With the development of GPU extensions (Besard et al., 2019) and vendor-agnostic kernel abstractions such as KernelAbstractions.jl (Churavy, 2020), single-source and multi-platform design has become practical. In other words, Julia has the potential to achieve high performance, high expressiveness, and performance portability within a single language. For fluid solvers alone, Julia implementations such as Oceananigans.jl (Ramadhan et al., 2020) and MPAS-Ocean (Bishnu et al., 2023) have been reported to achieve performance comparable to or exceeding that of Fortran. Oceananigans.jl (Ramadhan et al., 2020) is a Julia-based flow solver that also supports Lagrangian tracer-particle

tracking. To the authors' knowledge, no Julia-based solver targeting the present combination of inertial-particle dynamics and distributed particle migration has been reported.

In this study, we aim to develop LCS.jl (Lagrangian Cloud Simulator in Julia), a single-source and multi-platform multiphase DNS model implemented in the Julia language and KernelAbstractions.jl. We validate the developed model through comparison of fluid and particle statistics with prior studies. We propose a parallel particle communication algorithm based on prefix scan and evaluate its GPU-side implementation against the CPU-delegated sequential implementation. We further evaluate the computational performance of the single-source design, including strong and weak scaling in large-scale parallel computations, GPU–CPU performance comparison using the same codebase, and a heterogeneous execution configuration in which GPUs are utilized in an auxiliary capacity. Through these contributions, we assess the portability and scalability of LCS.jl across a variety of computational resource configurations.

2 Methods

All computations reported in this study use double-precision floating-point arithmetic.

2.1 Flow Phase

The carrier fluid obeys the incompressible Navier–Stokes equations:

$$\nabla \cdot U = 0, \tag{1}$$

$$\frac{\partial U}{\partial t} + (U \cdot \nabla)U = -\frac{1}{\rho}\nabla p + \nu \nabla^2 U + F(\boldsymbol{x}, t), \tag{2}$$

where U is the fluid velocity, ρ is the constant fluid density, p is the pressure, and F is the external forcing. The kinematic viscosity ν was set to $1.5 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$, following the original LCS formulation (Onishi et al., 2015) for atmospheric conditions (1013 hPa, 298 K). Turbulence is sustained by the external forcing F. LCS.jl supports several turbulent-forcing schemes. In the validation series considered below, Reduced Communication Forcing (RCF) (Onishi et al., 2011) was used at all resolutions. RCF coarse-grains the velocity field before the Fourier transform and acts on the two lowest wavenumber shells, $k \in \{1, 2\}$. Writing $E_k(t)$ and $E_{k,\mathrm{target}}$ for the instantaneous and prescribed target energies of shell k, respectively, the Fourier-space velocity increment is

$$\widehat{U}_L^{\mathrm{inc}}(\boldsymbol{k}, t) = \left[\left(\frac{E_{k,\mathrm{target}}}{E_k(t)} \right)^{1/2} - 1 \right] \widehat{U}_L(\boldsymbol{k}, t), \qquad k - \frac{1}{2} \le |\boldsymbol{k}| < k + \frac{1}{2}. \tag{3}$$

The input $E_{1,\mathrm{target}}$ specifies the target energy of the $k = 1$ shell, while $E_{2,\mathrm{target}}$ is determined from it using the prescribed RCF target spectrum. The quantity above is the discrete velocity increment applied by RCF during the time-stepping procedure and thus represents the discrete-time action of the forcing term F in Eq. (2), rather than the continuous-time forcing acceleration itself; the detailed RCF formulation is given by Onishi et al. (2011). RCF depends on the instantaneous velocity field through $E_k(t)$ but is distinct from the forcing scheme proposed by Lamorgese et al. (2005). The governing equations were discretized

on a uniform Cartesian grid, with variables stored in a marker and cell (MAC) arrangement (Harlow and Welch, 1965). The computational domain is a cube of side $2\pi L_0$ with periodic boundary conditions applied in all directions. The grid spacing is $\Delta = 2\pi L_0/N$, where N is the number of grid points in each direction. For spatial discretization, the conservative fourth-order central difference scheme (Morinishi et al., 1998) was applied to the convection term, fourth-order central differences were applied to the viscous term, and second-order central differences were applied to the pressure term. Time integration was performed using the explicit midpoint method, a two-stage, second-order Runge–Kutta (RK2) method, with a half-step first stage followed by a full-step update. Pressure–velocity coupling was handled by the highly simplified marker and cell (HSMAC) scheme (Hirt and Cook, 1972) with red–black coloring. The HSMAC iteration uses nondimensional velocity and spatial coordinates, $U^* = U/U_0$ and $\boldsymbol{x}^* = \boldsymbol{x}/L_0$, where U_0 and L_0 are the reference velocity and length, respectively. For the uniform grid used here, $\Delta^* = \Delta/L_0$. Iteration was continued until the root-mean-square (RMS) velocity divergence satisfied $\Delta^* \mathrm{RMS}(\nabla^* \cdot U^*) < \delta$, with the dimensionless tolerance $\delta = 10^{-3}$ following Onishi et al. (2015). The time step Δt was determined to satisfy both the convective and viscous Courant–Friedrichs–Lewy (CFL) constraints for stability:

$$\mathrm{CFL}_{\mathrm{conv}} = \max \frac{\|U\| \Delta t}{\Delta} \leq C_1, \tag{4}$$

$$\mathrm{CFL}_{\mathrm{vis}} = \frac{\nu \Delta t}{\Delta^2} \leq C_2, \tag{5}$$

where $C_1 = 0.3$ and $C_2 = 0.125$. The bulk Reynolds number is defined as $\mathrm{Re} = U_0 L_0/\nu$. Three-dimensional domain decomposition was used for inter-process parallelization. The decomposition was chosen so that each subdomain is as close to cubic as possible, minimizing the surface-area-to-volume ratio and reducing communication cost.

The numerical formulation was inherited from the original LCS framework and retained for its suitability for large-scale parallel computation. The local finite-difference stencils require only neighboring-subdomain communication, while the conservative advection scheme preserves the appropriate discrete conservation properties. Pseudo-spectral methods provide high spatial accuracy but require global Fourier transforms, whose communication cost becomes increasingly important in large distributed-memory calculations. The present finite-difference formulation avoids these global transforms for the spatial discretization, while RCF retains large-scale spectral forcing with reduced global communication. Onishi et al. (2011, 2013) demonstrated that the present finite-difference model employing these numerical methods maintains sufficient accuracy compared to spectral methods.

2.2 Particle Phase

Particle motion obeys the following system of ordinary differential equations:

$$\frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}t} = \boldsymbol{V}, \tag{6}$$

$$\frac{\mathrm{d}\boldsymbol{V}}{\mathrm{d}t} = -\frac{f}{\tau_{\mathrm{p}}}\big(\boldsymbol{V} - U(\boldsymbol{X}, t)\big), \tag{7}$$

where $\boldsymbol{X}$ and $\boldsymbol{V}$ are the particle position and velocity, and $U(\boldsymbol{X}, t)$ is the fluid velocity at the particle position.

$\tau_{\mathrm{p}} = \frac{2}{9}(\rho_{\mathrm{p}}/\rho)(a^2/\nu)$ is the particle relaxation time, following Onishi et al. (2015), where a is the particle radius and ρ_{p} is the particle density. The particle-to-fluid density ratio ρ_{p}/ρ was set to 8.43×10^2, following Onishi et al. (2015), corresponding

to water droplets in air under atmospheric conditions (1013 hPa, 298 K). The nonlinear drag correction factor f and the particle Reynolds number $\mathrm{Re_p}$ are defined as follows (Rowe and Henwood, 1961):

$$f = 1 + 0.15\,\mathrm{Re_p}^{0.687}, \tag{8}$$

$$\mathrm{Re_p} = \frac{2a|\boldsymbol{V} - U(\boldsymbol{X},t)|}{\nu}. \tag{9}$$

The fluid velocity $U(\boldsymbol{X},t)$ at the particle position $\boldsymbol{X}$ was computed by trilinear interpolation from the eight surrounding values on the corresponding staggered velocity grid, following the original LCS formulation (Onishi et al., 2015). The interpolation choice in the original framework was made with reference to previous comparisons with higher-order schemes, including cubic Hermite, cubic Lagrangian, and fifth-order Lagrangian interpolation (Sundaram and Collins, 1996). Particle motion was integrated using the same explicit midpoint (RK2) method as the fluid equations.

The present particle model uses a one-way-coupled point-particle approximation for dilute suspensions of small, heavy inertial particles. It retains drag with a finite-particle-Reynolds-number correction while neglecting added-mass and history forces, gravity, particle feedback on the carrier flow, collisions, and hydrodynamic interactions between particles. It is therefore a reduced form of the classical Maxey–Riley–Gatignol framework (Maxey and Riley, 1983; Gatignol, 1983). The validity of one-way coupling is governed by physical quantities such as particle volume fraction and mass loading rather than by the total number of tracked computational particles alone. In the original LCS applications, the particle mass and volume fractions were sufficiently dilute that flow modulation could be neglected. Large particle ensembles are used to obtain statistically converged particle statistics, particularly pair statistics such as the radial distribution function (RDF); in the performance benchmarks they also provide a demanding workload for the particle solver.

2.3 Implementation and Optimization in Julia

2.3.1 Performance Portability: Single Source, Multi-Platform Strategy

LCS.jl is designed to achieve performance portability across arbitrary platforms including CPUs and GPUs with a single source code. The core of this design is the KernelAbstractions.jl (Churavy, 2020) library. In the present model, higher-level abstractions built on this library were adopted (Listing 1). `Parallel.foraxes` is a for-loop abstraction provided by Parallel, a custom parallel execution module in LCS.jl, and generates platform-specific code for a variety of computational platforms from a single source. In addition to ordinary backend-independent parallel iteration, `Parallel.foraxes` provides LCS-specific iteration patterns, including the red–black coloring used by the HSMAC solver, and permits backend-specific specialization. An explicit data movement design was also adopted. In LCS.jl, host–device transfers are specified explicitly where required, whereas directive-based GPU porting approaches such as OpenACC can manage offloading and data movement through compiler directives. This suppresses the proliferation of pragma annotations required for performance optimization and maintains code readability.

Listing 1. Implementation of a finite-difference kernel in LCS.jl (a) and Fortran (b). In both listings, `N` is the local subdomain size and `2:N-1` denotes local subdomain indices, `U` is the fluid velocity field, `dUdt` is the time derivative of `U`, and `dx` is the grid spacing. In (a), `backend` specifies the target backend (e.g., GPUs from NVIDIA, AMD, Apple, and Intel, as well as CPUs).

(a)

```
Parallel.foraxes(
 backend, (2:N-1, 2:N-1, 2:N-1)
) do i, j, k
 dUdt[i, j, k] = - (U[i+1, j, k] - U[i-1, j, k]) / (2 * dx)
end
```

(b)

```
do concurrent (k = 2:N-1, j = 2:N-1, i = 2:N-1)
 dUdt(i, j, k) = - (U(i+1, j, k) - U(i-1, j, k)) / (2.0 * dx)
end do
```

2.3.2 Halo Communication

In LCS.jl, communication–computation overlap and time-blocking were introduced as communication optimizations in the HSMAC iteration loop, which occupies the largest fraction of execution time. Communication–computation overlap hides communication by concurrently performing halo region communication and computation that does not depend on the halo region. Halo region communication refers to exchanging boundary data between neighboring processes in distributed computing. Time-blocking reduces the number of communication calls by aggregating multiple iteration steps and exchanging halo regions in a single batch. The pressure correction in the HSMAC method uses second-order central differences, requiring a halo width of one cell. This is smaller than the halo width required for the fourth-order convection discretization (three cells). Before the grouped HSMAC iterations, a wider halo is exchanged and the active computational domain is successively contracted so that every stencil evaluation uses valid data. This changes only the communication schedule and introduces no additional numerical approximation at subdomain boundaries. By grouping three HSMAC iteration steps together and exchanging a three-cell halo in a single communication call, the number of communications is reduced by a factor of three without increasing memory usage.

2.3.3 Parallel Particle Communication Based on Prefix Scan

In particle tracking, the communication destination of particles that cross subdomain boundaries is determined only at runtime. Unlike the static communication in halo exchange, both the number and destinations of migrating particles vary during the simulation, and the migrated state includes particle position and velocity.

In the original LCS (Fortran implementation), particle packing for communication was performed sequentially on the CPU. All particles are scanned in order; each time a particle outside the subdomain boundary is detected, a per-direction counter is

incremented to determine its storage position in the send buffer. In this scheme, the storage position of particle i depends on the processing result of particles 1 through $i-1$ (i.e., the counter value). This sequential dependency limits direct parallelization because concurrent updates to the same counter require synchronization.

In LCS.jl, the sequential counter-based packing was replaced by a parallel algorithm based on prefix scan. The procedure consists of the following three stages.

1. **Mask computation**: A boundary-crossing check is performed for all particles to generate a `mask` array. Particles to be sent are assigned 1; particles remaining within the subdomain are assigned 0. Each particle's check is independent, so this stage is executed in parallel by the selected backend.
2. **Prefix scan**: The cumulative sum of the mask array is computed to determine the storage position of each particle in the send buffer. The storage position of particle i is given by $\mathrm{scan}[i] = \sum_{j=1}^{i} \mathrm{mask}[j]$.
3. **Parallel packing**: For particles with mask $= 1$, each particle is written in parallel to the position indicated by $\mathrm{scan}[i]$. Particles with mask $= 0$ are excluded from writing, so no threads write to the same destination and no data race occurs.

As a concrete example, consider five particles where particles 2 and 4 are to be sent:

$$\mathrm{mask}: \quad 0,\ 1,\ 0,\ 1,\ 0$$
$$\mathrm{scan}: \quad 0,\ 1,\ 1,\ 2,\ 2$$

Particle 2 is written to $\mathrm{buffer}[1]$ via $\mathrm{scan}[2] = 1$, and particle 4 is written to $\mathrm{buffer}[2]$ via $\mathrm{scan}[4] = 2$.

In the sequential implementation, storage positions are determined one particle at a time by incrementing a counter. In the present implementation, the prefix scan and subsequent packing were executed in parallel on the selected backend. For GPU execution, this enabled particle packing to be performed directly on the GPU without delegation to the CPU.

3 Results

3.1 Validation

3.1.1 Flow Phase

Table 1 shows the flow statistics in the statistically steady state obtained from GPU computations using LCS.jl. Computations were performed up to $N^3 = 2048^3$. The kinematic viscosity and reference length L_0 were fixed, and the bulk Reynolds number $\mathrm{Re} = U_0 L_0/\nu$ was varied through the reference velocity U_0. RCF was used with the configured $k = 1$ target energies listed in Table 1; these values were adjusted slightly so that $u_{\mathrm{rms}}/U_0 \approx 1$. The Taylor microscale and Taylor-microscale Reynolds number are defined as $\lambda = \sqrt{15\nu u_{\mathrm{rms}}^2/\varepsilon}$ and $\mathrm{Re}_\lambda = u_{\mathrm{rms}}\lambda/\nu$, respectively, where ε is the mean energy dissipation rate. The Kolmogorov length and time scales are $\eta = (\nu^3/\varepsilon)^{1/4}$ and $\tau_\eta = (\nu/\varepsilon)^{1/2}$, respectively. All cases satisfy $k_{\max}\eta \approx 2$, where $k_{\max} = N/2$ is the maximum resolved wavenumber, ensuring sufficient spatial resolution down to the Kolmogorov scale. The obtained statistics agreed well with the prior study (Onishi et al., 2011).

Figure 1 shows kinetic energy spectra $E(k)$, defined as $E(k) = \frac{1}{2}\sum_k' |\widehat{U}(\boldsymbol{k})|^2$, where $\widehat{U}$ denotes the Fourier-transformed velocity field, $\sum_k' \equiv \frac{4\pi k^2}{N_k}\sum_{k-1/2\le|\boldsymbol{k}|<k+1/2}$, and N_k is the number of wavevectors $\boldsymbol{k}$ satisfying $k-1/2 \le |\boldsymbol{k}| < k+1/2$. The factor $4\pi k^2/N_k$ is a discrete mode-count correction that converts the shell average to the corresponding continuum spherical-shell measure, rather than a smoothing operation. For the largest case with $\mathrm{Re}_\lambda = 535$, an inertial subrange spanning approximately one decade is observed, demonstrating that the energy cascade is properly captured. This confirmed that LCS.jl correctly reproduced turbulence statistics, validating the fluid solver implementation.

3.1.2 Particle Phase

Figure 2 shows the RDF at contact $g(r = R)$ as a function of the Stokes number St and Re_λ. The RDF is defined as the ratio of the observed particle-pair density at separation r to that for a spatially uniform particle distribution. Here, r is the pair separation and R is the contact separation; for the monodisperse cases, $R = d = 2a$, where d is the particle diameter. The Stokes number is $\mathrm{St} = \tau_\mathrm{p}/\tau_\eta$. For all validation cases, $a/\eta \le 0.21$ and $d/\eta \le 0.42$, so the particle diameter remains below the Kolmogorov length scale. The quantity $g(r = R)$ quantifies clustering intensity, with unity corresponding to a uniform distribution, and is used in cloud-microphysics collision kernels. The results agreed well with the prior study (Onishi and Seifert, 2016) across all St and Re_λ conditions. The solid curves are the empirical RDF parameterization from that study. The differences between the parameterization and the DNS results at larger St reflect the approximation inherent in the parameterization. The non-monotonic dependence on St is consistent with the change in the dominant clustering response with inertia: clustering strengthens as the particle response time becomes comparable to the dissipative turbulent timescale and weakens again as larger-inertia particles respond less strongly to the smallest turbulent scales. This confirmed that LCS.jl quantitatively reproduced the St and Re_λ dependence of particle clustering in turbulence, validating the particle tracking implementation.

3.2 Computational Performance

The primary performance measurements were conducted on TSUBAME4.0 at the Institute of Science Tokyo. Table 2 shows the node configuration.

Table 1. Flow statistics in the statistically steady state at different resolutions. u_rms/U_0 is the normalized root-mean-square velocity; L/L_0, λ/L_0, and η/L_0 denote the normalized integral length scale, Taylor microscale, and Kolmogorov length scale, respectively. $E_{1,\mathrm{target}}/U_0^2$ is the configured RCF target energy of the $k = 1$ shell in the nondimensional solver variables. $-S$ and F denote the negative skewness and flatness of the longitudinal velocity gradient.

N^3	Re	$E_{1,\mathrm{target}}/U_0^2$	u_rms/U_0	L/L_0	λ/L_0	Re_λ	η/L_0	$k_\mathrm{max}\eta$	$-S$	F
128^3	143	0.664	0.969	1.42	0.577	79.9	0.0328	2.10	0.482	4.82
256^3	360	0.649	1.00	1.37	0.358	129	0.0160	2.05	0.488	5.45
512^3	907	0.685	1.03	1.25	0.223	208	0.00783	2.01	0.564	6.58
1024^3	2220	0.660	1.00	1.12	0.149	333	0.00416	2.13	0.570	7.47
2048^3	5590	0.650	0.995	1.27	0.0961	535	0.00211	2.16	0.604	9.02

3.2.1 Performance Measurement Procedure

Unless otherwise noted, computational performance is reported as the wall-clock time per complete time-integration step, including the fluid update, particle update, inter-process communication, and other operations executed within the time-stepping loop; initialization, JIT compilation, and output are excluded. For distributed-memory calculations, the time for each step is taken as the maximum wall-clock time over all MPI processes.

Warm-up steps were excluded before averaging the execution times. For the scaling and optimization measurements, CPU and single-GPU benchmarks used 100 warm-up steps, whereas multi-GPU benchmarks on TSUBAME4.0 used 1000 warm-up steps. The reported execution time was obtained by averaging the subsequent measured steps.

Measurement stability was assessed using two criteria: the relative difference between the mean execution times in the first and second halves of the measurement window, and the temporal trend in execution time, quantified as the relative change per 100 time steps. Both quantities were required to be no greater than 3%. Only measurement windows satisfying both criteria were used for the reported timings.

3.2.2 Halo Communication Optimization

Figure 3 shows the wall-time reduction produced by the halo communication optimizations as a function of the number of GPUs. Measurements were performed for grid size 1500^3 and particle count 750^3, ranging from 8 to 64 GPUs. Time-blocking alone improved the wall-clock time by approximately 1.6–6.6%. The reduction in halo exchanges is partly offset by the additional computation performed in the enlarged halo region, which limits the gain from time-blocking alone. Combining time-blocking with communication–computation overlap was the fastest configuration at every tested GPU count, improving the wall-clock time by approximately 5.6–10% relative to the unoptimized case. These results confirmed the performance improvement obtained by communication optimization.

3.2.3 Particle Communication Optimization Based on Prefix Scan

Table 3 shows the particle communication cost for two implementations: the GPU-side prefix-scan implementation and the CPU-delegated sequential implementation, whose cost was estimated as described below. In the latter, packing and unpacking of particle data were performed sequentially on the CPU and required the associated host–device transfers. The measurement conditions were grid size $N^3 = 1500^3$, particle count $N_\mathrm{p} = 750^3$, and 8 GPUs. The particle arrays consisted of velocity fields and positions for the RK2 stages (2 each), velocity time derivatives (1), particle IDs, and particle diameters (1 each). Since vector quantities had 3 components, the total number of Float64 arrays was $(2+2+1)\times 3+2 = 17$. The number of particles per rank was $N_\mathrm{p}^\mathrm{rank} = 750^3/8 \approx 5.3\times 10^7$, and the data transfer volume was

$$17 \times 8\,\mathrm{B} \times 5.3 \times 10^7 \approx 7.2\,\mathrm{GB}. \tag{10}$$

Using the PCIe Gen5 $\times 16$ one-way bandwidth of 64 GB/s on TSUBAME4.0, the estimated transfer time for each of D2H (device-to-host) and H2D (host-to-device) is

$$t_{\mathrm{D2H}} = t_{\mathrm{H2D}} = 7.2\,\mathrm{GB}/64\,\mathrm{GB/s} \approx 113\,\mathrm{ms}. \tag{11}$$

Sequential packing on the CPU was limited by the single-core memory bandwidth. The single-core memory bandwidth was measured using the STREAM Triad benchmark (McCalpin, 1991) and confirmed to be approximately 41 GB/s. The estimated cost for each of packing and unpacking was

$$t_{\mathrm{pack}} = t_{\mathrm{unpack}} = 7.2\,\mathrm{GB}/41\,\mathrm{GB/s} \approx 176\,\mathrm{ms}. \tag{12}$$

Unlike the CPU-delegated sequential implementation, the GPU-side prefix-scan implementation does not require D2H/H2D transfers of the full particle arrays. The measured packing and unpacking times were 12 ms and 60 ms, respectively. The communication cost per RK2 stage in the GPU-side prefix-scan implementation was $12\,\mathrm{ms} + 60\,\mathrm{ms} = 72\,\mathrm{ms}$, compared with the estimated $578\,\mathrm{ms}$ for the CPU-delegated sequential implementation, $(113\,\mathrm{ms} + 176\,\mathrm{ms}) \times 2$. This corresponds to an approximately 88% reduction, $(578-72)/578 \approx 0.88$. Under the same 8-GPU benchmark condition, the measured wall-clock time per complete time-integration step was 1.141 s. The estimated CPU-delegated particle communication cost for the two RK2 stages alone was 1.156 s, comparable to the entire measured step time of the GPU-side implementation. This comparison is limited to the sequential CPU-delegated implementation considered here. Parallel CPU-side packing and unpacking, for example using the backend-independent prefix-scan formulation, could reduce the CPU processing cost, whereas full-array D2H and H2D transfers would remain necessary. Other parallel packing strategies, such as atomic-based packing, were not evaluated in the present study.

This reduction therefore removes a major communication bottleneck and enables effective utilization of GPU performance in large-scale multiphase flow simulations.

3.2.4 Julia vs Fortran Comparison

Figure 4 shows the mean wall time per time-integration step as a function of the number of CPU cores for LCS.jl (Julia) and the original LCS (Fortran) implementation. Computations were performed with grid size $N^3 = 384^3$ and particle count $N_{\mathrm{p}} = 192^3$, with identical computational settings in both implementations.

At 1, 2, and 4 CPU cores, LCS.jl showed execution times approximately 8–11% longer than the Fortran implementation. At 8 CPU cores, the two implementations exhibited comparable execution times, with a difference of approximately 1%.

LCS.jl uses identical kernel code for both CPU and GPU execution. The Fortran implementation applies CPU-specific loop transformations, such as variable precomputation and loop fusion. In a single kernel targeting both CPU and GPU execution, constraints on inter-thread data sharing limit the extent to which equivalent optimizations can be applied. At small core counts, communication accounts for only a small fraction of the total execution time, making these differences in computational performance more visible, as reflected in the 8–11% gap observed at 1–4 CPU cores. As the core count increases, communication accounts for a larger fraction of the total execution time, reducing the influence of single-core computational performance on

the overall wall time and narrowing the gap to approximately 1% at 8 CPU cores. Further CPU-specific tuning may improve CPU performance, at the cost of additional architecture-specific implementation complexity.

These results confirmed that LCS.jl achieves CPU performance comparable to the Fortran implementation over the tested range of CPU cores while retaining a single-source and multi-platform design.

3.2.5 Scaling

Scaling measurements were performed on TSUBAME4.0 using up to 64 nodes. The number of particles was fixed at one eighth of the number of grid points; for example, 750^3 grid points were paired with 375^3 particles. This particle load lies within the range of approximately 0.1–10 particles per grid point used in typical cloud microphysics computations. Performance evaluation was conducted independently of physical statistical analysis. To isolate scaling performance from physical transients, the number of HSMAC iterations was fixed at 6, which is the typical value observed at statistically steady state. The common timing and stability-assessment procedure is described in Sect. 3.2.1.

For GPU scaling, one MPI process was assigned to each GPU. For inter-node CPU scaling, one MPI process was assigned to each CPU socket and 96 Julia threads were used per process, with one Julia thread per physical CPU core and simultaneous multithreading disabled. However, assigning one process with 96 threads to each CPU socket was not optimal. To determine the process–thread balance, performance measurements were conducted with the total number of physical CPU cores fixed at 384 on two nodes, using $N^3 = 1500^3$ grids and $N_\mathrm{p} = 750^3$ particles, while the number of threads per process was varied. A maximum speedup of $2.18\times$ was achieved with four Julia threads per process relative to the one-process-per-socket configuration with 96 threads per process. This measured factor is used below to estimate the effective CPU performance in the GPU–CPU comparison.

The JIT-related first-step overhead was also evaluated separately from the steady-state timings. Let T_first denote the wall-clock time of the first complete time-integration step and T_prod the production execution time per step; the first-step overhead is $T_\mathrm{first} - T_\mathrm{prod}$. This timing excludes process creation and MPI-launcher startup. The measured overhead was 4.6–5.8 s for the multi-process CPU cases and 12.0–13.4 s for the GPU strong-scaling cases, with approximately constant overhead from 8 to 256 GPUs.

To characterize CPU scaling from a single core to a full node, Fig. 5 shows fixed-size intra-node CPU strong scaling using 1–192 physical CPU cores on one TSUBAME4.0 node. One MPI process was assigned to each physical CPU core, with simultaneous multithreading disabled. From one to 192 physical CPU cores, the wall time decreased from 29.14 s to 0.4222 s, corresponding to a speedup of $29.14/0.4222 = 69.0$. The scaling behavior showed step-like variations, with nearly linear local scaling over several ranges, including 8–32 and 96–192 cores. Despite these variations, the wall time continued to decrease up to the full-node configuration of 192 cores.

Figure 6 shows strong scaling results for GPU and CPU execution. Total problem sizes of $N^3 = 1500^3$ grids with $N_\mathrm{p} = 750^3$ particles and $N^3 = 3000^3$ grids with $N_\mathrm{p} = 1500^3$ particles were used. Since each TSUBAME4.0 node has four GPUs and two CPU sockets, scaling was measured up to 256 GPUs and 128 CPU sockets under the 64-node limit. Strong-scaling efficiency

is defined as

$$E_{\mathrm{s}} = \frac{T_0}{T_P}\frac{P_0}{P}, \tag{13}$$

where T_0 and P_0 denote the execution time and resource count of the baseline point, and T_P is the execution time at resource count P. The baseline resource counts were 8 GPUs and 4 CPU sockets for $N^3 = 1500^3$, and 64 GPUs and 32 CPU sockets for $N^3 = 3000^3$. Here, the resource count denotes the number of GPUs for the GPU series and the number of CPU sockets for the CPU series. For the GPU series, the strong-scaling efficiency at 256 GPUs relative to 64 GPUs was $E_{\mathrm{s}} = (1.188/0.4261)(64/256) = 0.697$, or 69.7%. For the CPU series, the corresponding efficiency at 128 CPU sockets relative to 32 CPU sockets was $E_{\mathrm{s}} = (35.57/13.61)(32/128) = 0.653$, or 65.3%. Figure 6 confirms that strong-scaling efficiency was 69.7% for GPUs (up to 256 GPUs) and 65.3% for the CPU series (up to 128 CPU sockets).

Figure 7 shows weak scaling results for GPU and CPU execution. For weak scaling, the problem size per resource was fixed at $N^3 = 750^3$ grid points and $N_{\mathrm{p}} = 375^3$ particles. The plotted distributed-memory series start at 8 resources, the smallest tested configuration involving inter-node communication in both the CPU and GPU series. Weak-scaling efficiency is defined relative to the 8-resource baseline as

$$E_{\mathrm{w}} = \frac{T_8}{T_P}, \tag{14}$$

where T_8 is the execution time at 8 resources and T_P is the execution time at resource count P. For the GPU series, the weak-scaling efficiency at 216 GPUs was $E_{\mathrm{w}} = 1.141/1.435 = 0.795$, or 79.5%. For the CPU series, the efficiency at 125 CPU sockets was $E_{\mathrm{w}} = 21.21/21.78 = 0.974$, or 97.4%. Figure 7 confirms that weak-scaling efficiency was 79.5% for GPUs (up to 216 GPUs) and 97.4% for the CPU series (up to 125 CPU sockets).

3.2.6 GPU and CPU Performance Comparison

Table 4 shows the performance comparison between GPU and CPU execution using the same number of nodes on TSUBAME4.0. Since each TSUBAME4.0 node has four GPUs and two CPU sockets, the numbers of GPUs and CPU sockets differ. Because the process and thread balance optimization was not applied during the inter-node CPU scaling experiments, the CPU execution time was estimated by dividing the measured CPU time by the measured process–thread speedup of 2.18 (Sect. 3.2.5).

LCS.jl achieved a maximum estimated speedup of approximately $16.3\times$ on GPUs over CPUs. This demonstrated that the single-source and multi-platform design can achieve an acceleration ratio comparable to GPU-dedicated implementations.

3.2.7 GPU Architecture Portability

Table 5 shows the wall-clock time per time-integration step for LCS.jl using the CUDA backend on an NVIDIA RTX A6000 and the AMDGPU backend on an AMD Radeon RX 9070 XT. The two GPUs were selected in part because their nominal memory bandwidths are comparable. The execution times on the two GPU platforms remained of the same order, with the

AMD GPU results approximately 1.2–1.5 times those on the NVIDIA GPU. These results support the performance portability of LCS.jl across different GPU architectures while retaining a single-source implementation.

3.2.8 Heterogeneous Execution

LCS.jl supports heterogeneous execution, in which fluid and particle time-stepping and particle statistics computation are assigned to different computational resources, all from a single codebase. A configuration was evaluated with time-stepping on CPUs and particle-statistics computation on a GPU. The radial distribution function and particle-number-density spectrum were evaluated at every time step. The RDF requires particle-pair searching, binning, and accumulation, whereas the particle-number-density spectrum requires accumulation over the particle population for multiple wavenumber shells and sampling directions, followed by MPI reduction. These statistics therefore constitute a substantial particle-statistics workload.

Computation was performed on a workstation containing two Intel Xeon Gold 6326 CPU sockets and an NVIDIA RTX A6000 GPU (48 GB of GPU memory). This configuration represents an environment in which the GPU is available as an auxiliary accelerator rather than as the primary compute resource.

Table 6 shows the execution time per time-integration step for 256^3 grids and 256^3 particles. Compared to the CPU-only configuration, the heterogeneous configuration reduced execution time by approximately 68%. The configuration change was realized by switching only the resource assignment, without modifying the algorithm or physical model.

This trial heterogeneous execution, delegating only statistics computation to a GPU, was effective even in configurations where the GPU was not the primary compute device. LCS.jl enables flexible utilization of computational resources in scenarios where budget or power constraints prevent assigning high-performance GPUs to all processes, or where GPUs are added incrementally to an existing CPU cluster.

4 Conclusions

We developed LCS.jl (Lagrangian Cloud Simulator in Julia), a single-source and multi-platform multiphase turbulence simulation model implemented in the Julia language and KernelAbstractions.jl.

Validation results confirmed that the present fluid and particle statistics agree well with those obtained in prior studies. This confirms the validity of both the fluid solver and the particle tracking implementation.

For GPU execution, the GPU-side prefix-scan implementation reduced the particle communication cost by approximately 88% relative to the CPU-delegated sequential implementation. This reduction removes a major communication bottleneck, enabling effective utilization of GPU performance in large-scale multiphase flow simulations.

At 1–4 CPU cores, LCS.jl showed execution times approximately 8–11% longer than the Fortran implementation, whereas the difference was approximately 1% at 8 CPU cores. These results confirmed that LCS.jl achieves CPU performance comparable to the Fortran implementation over the tested range of CPU cores while retaining a single-source and multi-platform design. In the large-scale scaling tests, the GPU series showed a strong-scaling efficiency of 69.7% (up to 256 GPUs) and a

weak-scaling efficiency of 79.5% (up to 216 GPUs). For the CPU series, strong-scaling efficiency was 65.3% (up to 128 CPU sockets), and weak-scaling efficiency was 97.4% (up to 125 CPU sockets).

LCS.jl achieved an estimated maximum speedup of approximately $16.3\times$ on GPUs over CPUs. This demonstrated that the single-source and multi-platform design can achieve an acceleration ratio comparable to GPU-dedicated implementations. The execution times per time-integration step using the AMDGPU backend on the AMD GPU were approximately 1.2–1.5 times those using the CUDA backend on the NVIDIA GPU. These results support the performance portability of LCS.jl across different GPU architectures while retaining a single-source implementation. A trial heterogeneous execution, delegating only statistics computation to a GPU, achieved a 68% reduction in execution time compared to the CPU-only configuration even in configurations where the GPU was not the primary compute device. This means that LCS.jl enables flexible utilization of computational resources in scenarios where budget or power constraints prevent assigning high-performance GPUs to all processes, or where GPUs are added incrementally to an existing CPU cluster.

The present LCS.jl is not merely a GPU-ported code, but a multiphase turbulence simulation platform that achieves both portability and scalability across a variety of computational resource configurations. This design philosophy can provide a guideline for building high-performance computational models that can adapt to future architectural changes and heterogeneous computing environments.

Code and data availability. The current development version of LCS.jl is available from the project website https://github.com/0samuraiE/LCS.jl under the MIT License. The exact version of the model used to produce the results in this paper is archived on Zenodo at https://doi.org/10.5281/zenodo.22137561 (Tominaga and Onishi, 2026), together with the configuration files used to run the simulations and the processed data and scripts used to produce the figures presented in this paper.

Author contributions. T. Tominaga developed the model code, performed the simulations, and prepared the manuscript. R. Onishi supervised the research and contributed to manuscript revision.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. This work used computational resources of the TSUBAME4.0 supercomputer at the Institute of Science Tokyo through the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures (JHPCN) and the High Performance Computing Infrastructure (HPCI) in Japan (Project ID: jh240041).

The authors used Claude (Anthropic) for language editing and translation assistance in the preparation of this manuscript. The authors take full responsibility for the content of the manuscript.

References

Besard, T., Foket, C., and De Sutter, B.: Effective Extensible Programming: Unleashing Julia on GPUs, IEEE Transactions on Parallel and Distributed Systems, 30, 827–841, https://doi.org/10.1109/tpds.2018.2872064, 2019.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B.: Julia: A Fresh Approach to Numerical Computing, SIAM Review, 59, 65–98, https://doi.org/10.1137/141000671, 2017.

Bishnu, S., Strauss, R. R., and Petersen, M. R.: Comparing the Performance of Julia on CPUs versus GPUs and Julia-MPI versus Fortran-MPI: a case study with MPAS-Ocean (Version 7.1), Geoscientific Model Development, 16, 5539–5559, https://doi.org/10.5194/gmd-16-5539-2023, 2023.

Churavy, V.: KernelAbstractions.jl, Zenodo [code], https://doi.org/10.5281/zenodo.4021259, 2020.

Gatignol, R.: The Faxén formulae for a rigid particle in an unsteady non-uniform Stokes flow, Journal de Mécanique Théorique et Appliquée, 2, 143–160, 1983.

Harlow, F. H. and Welch, J. E.: Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface, Physics of Fluids, 8, 2182–2189, https://doi.org/10.1063/1.1761178, 1965.

Hirt, C. and Cook, J.: Calculating three-dimensional flows around structures and over rough terrain, Journal of Computational Physics, 10, 324–340, https://doi.org/10.1016/0021-9991(72)90070-8, 1972.

Ireland, P. J., Bragg, A. D., and Collins, L. R.: The effect of Reynolds number on inertial particle dynamics in isotropic turbulence. Part 1. Simulations without gravitational effects, Journal of Fluid Mechanics, 796, 617–658, https://doi.org/10.1017/jfm.2016.238, 2016.

Kolmogorov, A. N.: Local structure of turbulence in an incompressible viscous fluid at very high Reynolds numbers, Doklady Akademii Nauk SSSR, 30, 299–303, in Russian; reprinted in Uspekhi Fizicheskikh Nauk **93**, 476–481 (1967), https://doi.org/10.3367/UFNr.0093.196711h.0476; English translation: Sov. Phys. Usp. **10**, 734–746 (1968), https://doi.org/10.1070/PU1968v010n06ABEH003710, 1941.

Lamorgese, A. G., Caughey, D. A., and Pope, S. B.: Direct numerical simulation of homogeneous turbulence with hyperviscosity, Physics of Fluids, 17, 015 106, https://doi.org/10.1063/1.1833415, 2005.

Maxey, M. R. and Riley, J. J.: Equation of motion for a small rigid sphere in a nonuniform flow, The Physics of Fluids, 26, 883–889, https://doi.org/10.1063/1.864230, 1983.

McCalpin, J. D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers, https://web.archive.org/web/20260412211805/https://www.cs.virginia.edu/stream/, 1991.

Morinishi, Y., Lund, T., Vasilyev, O., and Moin, P.: Fully Conservative Higher Order Finite Difference Schemes for Incompressible Flow, Journal of Computational Physics, 143, 90–124, https://doi.org/10.1006/jcph.1998.5962, 1998.

Onishi, R. and Seifert, A.: Reynolds-number dependence of turbulence enhancement on collision growth, Atmospheric Chemistry and Physics, 16, 12 441–12 455, https://doi.org/10.5194/acp-16-12441-2016, 2016.

Onishi, R., Baba, Y., and Takahashi, K.: Large-scale forcing with less communication in finite-difference simulations of stationary isotropic turbulence, Journal of Computational Physics, 230, 4088–4099, https://doi.org/10.1016/j.jcp.2011.02.034, 2011.

Onishi, R., Takahashi, K., and Vassilicos, J.: An efficient parallel simulation of interacting inertial particles in homogeneous isotropic turbulence, Journal of Computational Physics, 242, 809–827, https://doi.org/10.1016/j.jcp.2013.02.027, 2013.

Onishi, R., Matsuda, K., and Takahashi, K.: Lagrangian Tracking Simulation of Droplet Growth in Turbulence–Turbulence Enhancement of Autoconversion Rate, Journal of the Atmospheric Sciences, 72, 2591–2607, https://doi.org/10.1175/jas-d-14-0292.1, 2015.

Pope, S. B.: Turbulent Flows, Cambridge University Press, Cambridge, UK, https://doi.org/10.1017/CBO9780511840531, 2000.

Ramadhan, A., Wagner, G., Hill, C., Campin, J.-M., Churavy, V., Besard, T., Souza, A., Edelman, A., Ferrari, R., and Marshall, J.: Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs, Journal of Open Source Software, 5, 2018, https://doi.org/10.21105/joss.02018, 2020.

Rowe, P. N. and Henwood, G. A.: Drag forces in a hydraulic model of a fluidized bed. Part I, Transactions of the Institution of Chemical Engineers, 39, 43–54, https://catalog.hathitrust.org/Record/000520989, 1961.

Sundaram, S. and Collins, L. R.: Numerical considerations in simulating a turbulent suspension of finite-volume particles, Journal of Computational Physics, 124, 337–350, https://doi.org/10.1006/jcph.1996.0064, 1996.

Sundaram, S. and Collins, L. R.: Collision statistics in an isotropic particle-laden turbulent suspension. Part 1. Direct numerical simulations, Journal of Fluid Mechanics, 335, 75–109, https://doi.org/10.1017/s0022112096004454, 1997.

Tominaga, T. and Onishi, R.: LCS.jl v1.0, Zenodo [code], https://doi.org/10.5281/zenodo.22137561, 2026.

Wang, L.-P., Wexler, A. S., and Zhou, Y.: Statistical mechanical description and modelling of turbulent collision of inertial particles, Journal of Fluid Mechanics, 415, 117–153, https://doi.org/10.1017/s0022112000008661, 2000.

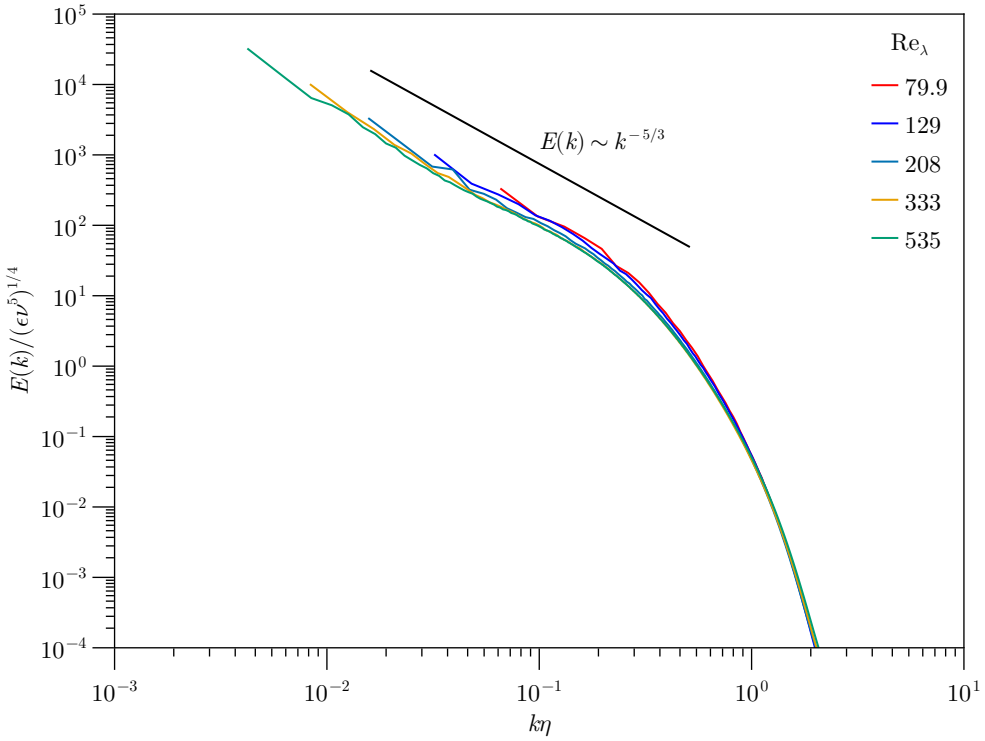


Figure 1. Energy spectra $E(k)$ normalized by the Kolmogorov energy-spectrum scale $(\varepsilon\nu^5)^{1/4}$ as a function of the normalized wavenumber $k\eta$. Results are shown for resolutions $N^3 = 128^3$ through 2048^3 ($\mathrm{Re}_\lambda = 79.9$–$535$). The dashed line indicates the Kolmogorov $k^{-5/3}$ scaling.

Table 2. TSUBAME4.0 node configuration.

Component	**Specification**
CPU	AMD EPYC 9654 2.4 GHz × 2 sockets
Physical CPU cores	96 cores/socket; 192 cores/node
Simultaneous multithreading	Not used in reported CPU measurements
Memory	768 GiB (DDR5-4800)
GPU	NVIDIA H100 SXM5 94 GB HBM2e × 4
CPU–GPU interconnect	PCIe Gen5 ×16 (64 GB/s one-way)
SSD	1.92 TB NVMe U.2 SSD
Network interconnect	InfiniBand NDR200 200 Gbps × 4

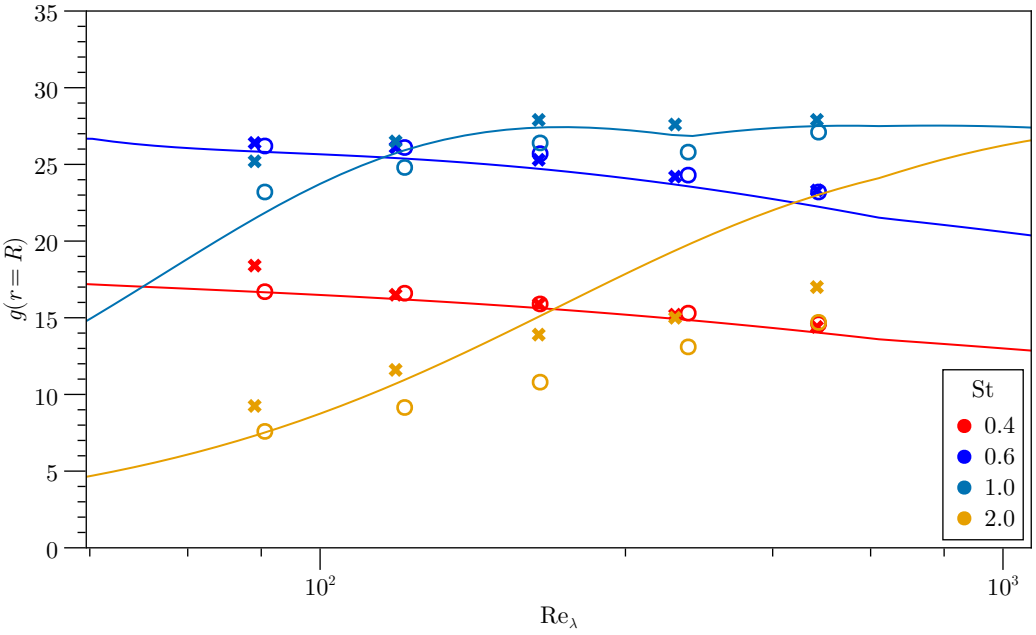


Figure 2. Radial distribution function (RDF) at contact $g(r = R)$ as a function of the Taylor-microscale Reynolds number Re_λ. Crosses denote the present results; open circles denote the corresponding reference results of Onishi and Seifert (2016) for $\mathrm{St} = 0.4$, 0.6, 1.0, and 2.0. Solid lines denote the RDF parameterization of Onishi and Seifert (2016). Colors indicate the Stokes number St.

Table 3. Particle communication cost per RK2 stage for the GPU-side prefix-scan (measured) and CPU-delegated sequential (estimated) implementations ($N^3 = 1500^3$, $N_\mathrm{p} = 750^3$, 8 GPUs). MPI communication cost is equivalent in both implementations (<1 ms per direction) and is omitted. The CPU-delegated estimates use PCIe Gen5 ×16 one-way bandwidth ∼64 GB/s and EPYC 9654 single-core bandwidth ∼41 GB/s.

Phase	**GPU-side prefix-scan (measured)**	**CPU-delegated sequential (estimated)**
D2H transfer (full particle arrays)	—	∼113 ms
Packing	12 ms	∼176 ms
Unpacking	60 ms	∼176 ms
H2D transfer (full particle arrays)	—	∼113 ms
Total (per stage)	**72 ms**	**∼578 ms**

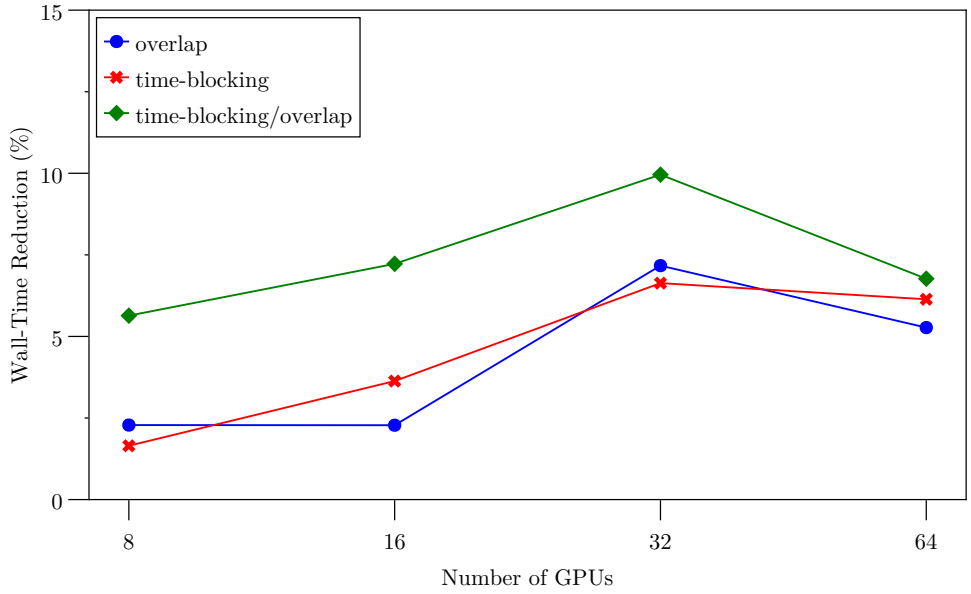


Figure 3. Wall-time reduction relative to the unoptimized baseline as a function of the number of GPUs ($N^3 = 1500^3$, $N_\mathrm{p} = 750^3$) for three halo communication optimizations: communication–computation overlap, time-blocking, and their combination. The reduction is defined as $(1 - T_\mathrm{opt}/T_\mathrm{base}) \times 100\%$.

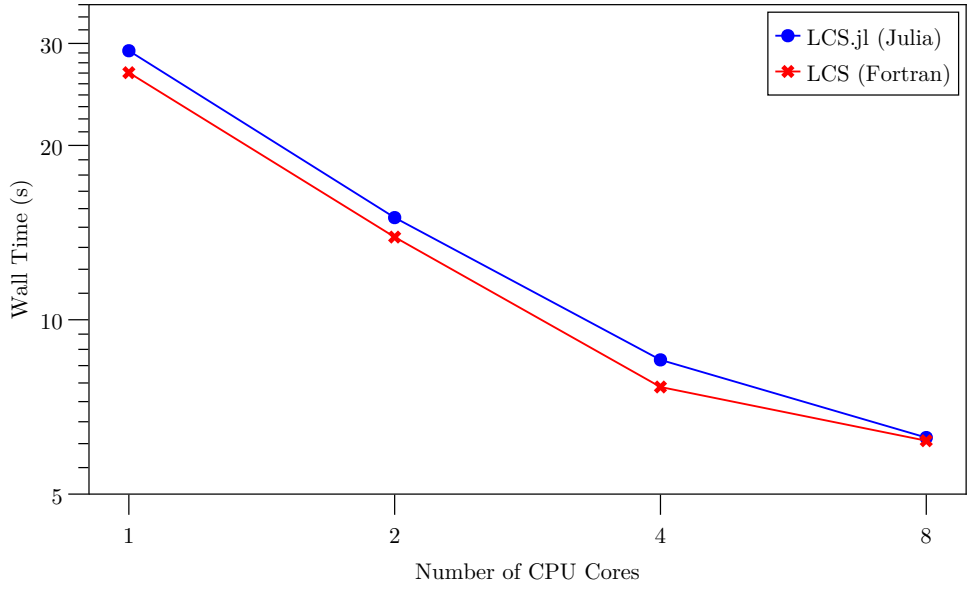


Figure 4. Mean wall time per time-integration step as a function of the number of CPU cores for LCS.jl (Julia) and the original LCS (Fortran) implementation ($N^3 = 384^3$, $N_\mathrm{p} = 192^3$). One MPI process was assigned to each physical CPU core.

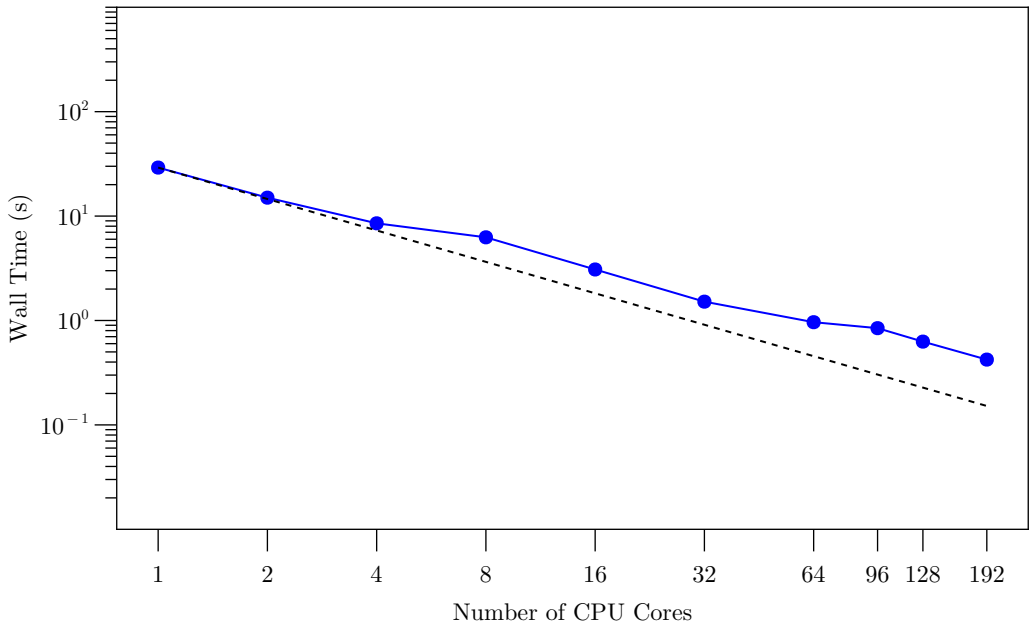
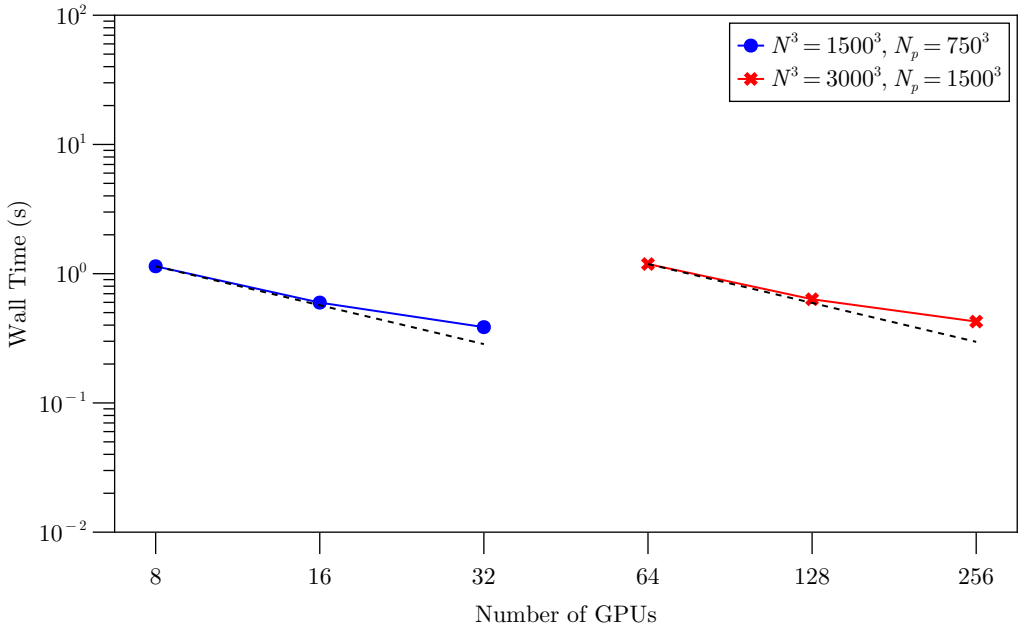


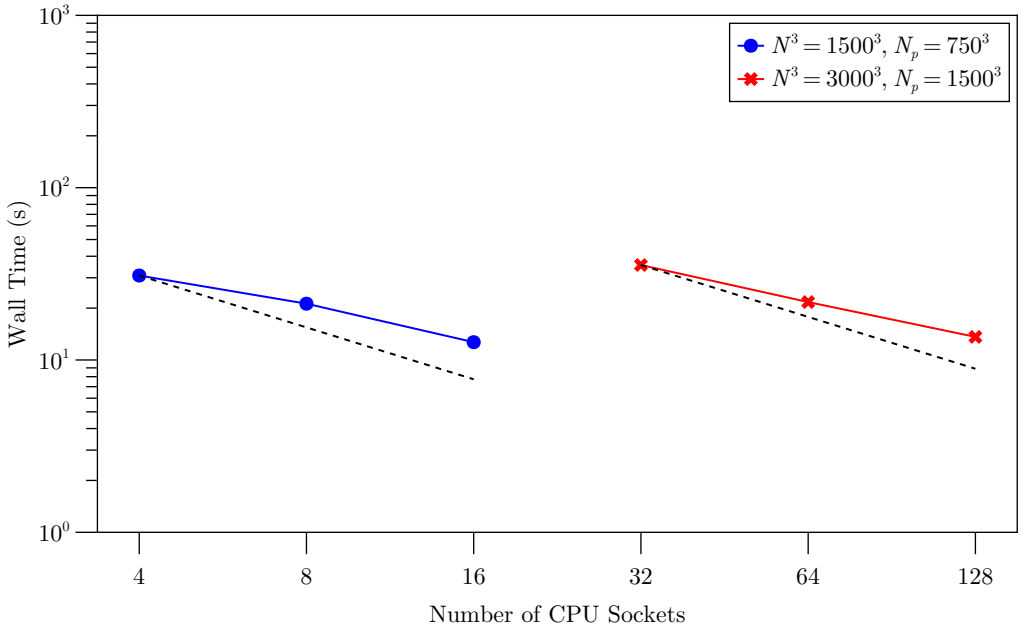
Figure 5. Fixed-size intra-node CPU strong scaling on one TSUBAME4.0 node, with one MPI process per physical CPU core and no simultaneous multithreading. The solid line with circles shows the measured wall time, and the dashed line indicates ideal strong scaling from the one-core result.

Table 4. Wall-clock time per time-integration step for GPU and CPU execution on the same number of nodes on TSUBAME4.0. CPU execution times are estimates obtained by dividing the measured CPU times by the measured process–thread speedup of 2.18 (see Sect. 3.2.5).

N^3	N_p	Nodes	GPUs	GPU Time (s)	CPU Sockets	CPU Time (s) (Estimated)	Speedup
1500^3	750^3	2	8	1.14	4	14.2	12.4×
1500^3	750^3	4	16	0.598	8	9.71	16.3×
1500^3	750^3	8	32	0.386	16	5.81	15.1×
3000^3	1500^3	16	64	1.19	32	16.3	13.7×
3000^3	1500^3	32	128	0.634	64	9.93	15.7×
3000^3	1500^3	64	256	0.426	128	6.23	14.6×

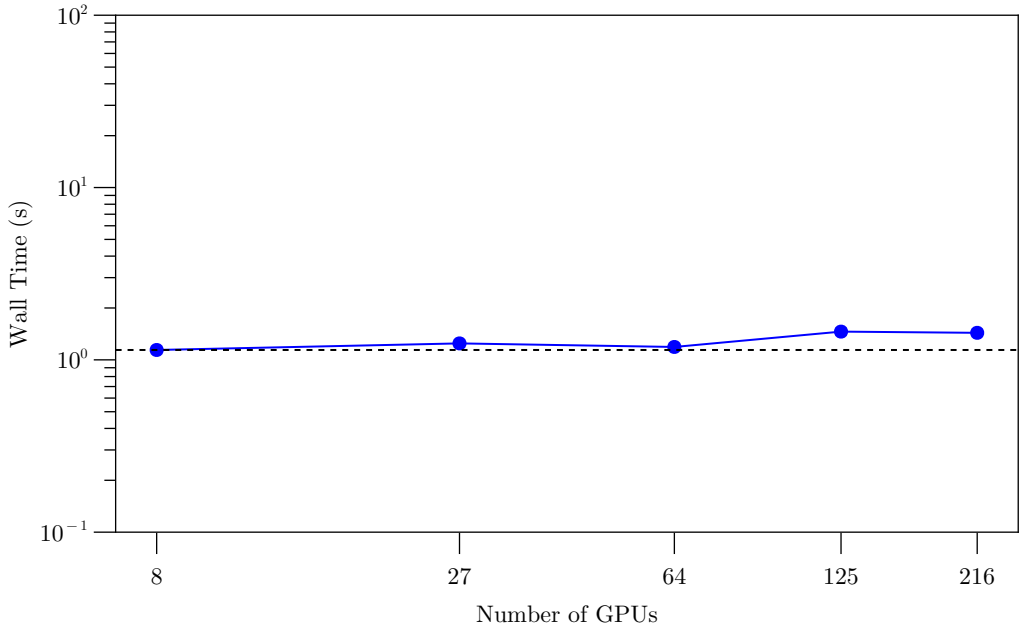


(a)

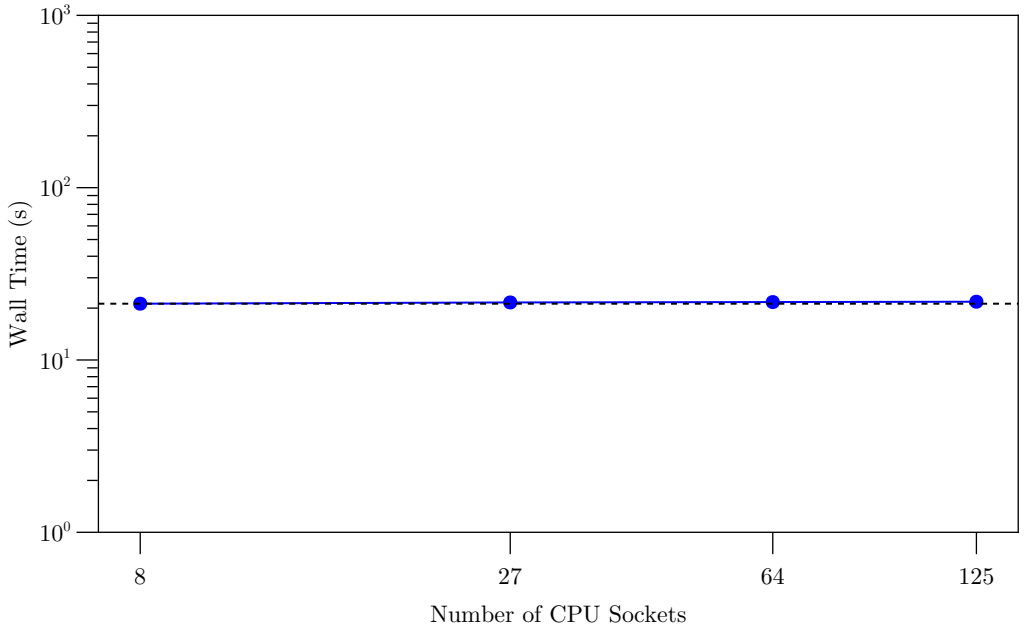


(b)

Figure 6. Wall-clock time per time-integration step as a function of resource count for fixed total problem sizes. (a) Number of GPUs for GPU execution; (b) number of CPU sockets for CPU execution. The dashed black lines indicate ideal strong scaling from the smallest resource count of each problem-size series.



(a)



(b)

Figure 7. Wall-clock time per time-integration step for weak scaling with a fixed problem size per resource. (a) Number of GPUs for GPU execution; (b) number of CPU sockets for CPU execution. The plotted distributed-memory series start at eight resources, and the dashed black horizontal lines indicate ideal weak scaling based on the eight-resource result.

Table 5. Wall-clock time per time-integration step on an NVIDIA RTX A6000 using the CUDA backend and an AMD Radeon RX 9070 XT using the AMDGPU backend, with grid size N and $N_{\mathrm{p}} = (N/2)^3$ particles.

Grid size N	NVIDIA RTX A6000 (CUDA) (s/step)	AMD Radeon RX 9070 XT (AMDGPU) (s/step)	AMD/NVIDIA
128	0.0297	0.0360	1.21
192	0.0790	0.106	1.34
256	0.163	0.242	1.48
384	0.524	0.804	1.54

Table 6. Wall-clock time per time-integration step for CPU-only and heterogeneous configurations ($N^3 = 256^3$, $N_{\mathrm{p}} = 256^3$). "Reduction from CPU-only" denotes the percentage reduction in wall-clock time relative to the CPU-only configuration.

Configuration	**CPU Cores**	**GPUs**	**Time (s)**	**Reduction from CPU-only**
CPU-only	32	—	71.0	—
Heterogeneous	32	1	22.7	68%