https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

EGUsphere
Preprint repository

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

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 (DNS) 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 (HPC). In this study, we developed LCS.jl (Lagrangian Cloud Simulator in Julia), a single-source and multi-platform multiphase turbulence simulation model implemented in Julia language and KernelAbstractions.jl. Validation results confirmed that the present fluid and particle statistics agree well with those obtained in prior studies. A GPU-native particle communication algorithm based on prefix-scan reduced the particle communication cost from approximately 78% (CPU-delegated) to 10% of total execution time. LCS.jl achieved computational performance equivalent to the Fortran implementation in many-processes computations. For GPUs, strong scaling efficiency was maintained above 85% (up to 256 GPUs) and weak scaling efficiency above 90% (up to 216 GPUs) on TSUBAME4.0 (a GPU supercomputer at the Institute of Science Tokyo). LCS.jl achieved a maximum speedup of 18.0× on GPUs over CPUs. A trial heterogeneous execution achieved a 72% 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 environments. In environments flows, one example is the growth process 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 form clustering, i.e., distribute non-uniformly in space. It has been recognized that this clustering significantly affects the collision-coalescence growth of cloud droplets. Extensive 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. Direct numerical simulations (DNSs) of such flows require spatial resolutions down to the Kolmogorov scale (Kolmogorov, 1941; Pope, 2000). The number of computational grid points increases rapidly with increasing Reynolds number, requiring more computational cost. In case of flow-particle

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

25 multiphase simulations, for, e.g., droplet growth in turbulent environment, the particle solver adds extra cost. It is a kind of great challenge to conduct multiphase DNS for environmental studies even on the latest HPC systems. Onishi et al. (2015) and Onishi and Seifert (2016) addressed this challenge and clarified the Reynolds number dependence of 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
30 highly optimized for GPU platforms, which represent the current mainstream in high-performance computing (HPC). With 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
35 et al. (2015). LCS adopts distributed-memory parallelization via MPI and achieved large-scale parallel computation on 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.

There are two challenges in GPU-porting such models. The first one concerns the particle communication algorithm. In fluid
40 computation, the communication range can be determined statically by the domain decomposition. In particle computation, by contrast, the number and destination of particles migrating across subdomain boundaries are changing in time, requiring dynamic communication. Conventional CPU-based implementations handle this communication sequentially, which is incompatible with GPU execution. 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
45 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 – 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 (Ra-
50 madhan et al., 2020) and MPAS-Ocean (Bishnu et al., 2023) have been reported to achieve performance comparable to or exceeding that of Fortran. However, only a limited number of Julia-based implementations of Euler–Lagrangian multiphase DNS solvers have been reported (Ramadhan et al., 2020). To the authors' knowledge, only a single Julia-based implementation of Euler–Lagrangian multiphase DNS solvers has been reported (Ramadhan et al., 2020) and it is limited to tracer particles. No Julia-based multiphase DNS solver for inertial particles has been reported, for which particle communication costs would
55 be more significant.

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 Julia language and KernelAbstractions.jl. We validate the developed model through comparison of fluid and particle statistics with prior studies. We propose a GPU-native particle communication algorithm and evaluate its performance against a CPU-delegated implementation. We further evaluate the computational performance of the single-

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

60 source design, including strong- and weak-scaling in large-scale parallel computations, GPU-CPU performance comparison using the identical 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

65
2.1 Flow Phase

The carrier fluid obeys the following governing equations:

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

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

where U is the fluid velocity, ρ is the density, p is the pressure, and F is the external forcing. The kinematic viscosity ν was
70 set to $1.5 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$, corresponding to atmospheric conditions (1013 hPa, 298 K). Turbulence is sustained by the forcing F, which injects energy into large scales at wavenumber $|k| < 2.5$. For the turbulent forcing, the Reduced Communication Forcing (RCF) (Onishi et al., 2011), an improved large-scale forcing scheme, was adopted. RCF reduces communication and computational costs by applying a box-mean filter to the velocity field before transforming it to wavenumber space, rather than transforming the entire field, while still confining energy injection to large scales. The governing equations were discretized on
75 a uniform Cartesian grid, with variables stored in a marker and cell (MAC) arrangement (Harlow and Welch, 1965). 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 a two-stage, second-order Runge–Kutta method (RK2). Pressure–velocity coupling was handled by the highly simplified marker and cell (HSMAC) scheme (Hirt and Cook, 1972) with Red–
80 Black coloring. Iteration was continued until the root-mean-square (RMS) of the velocity divergence fell below δ/Δ, where Δ is the grid spacing and δ was set to 10^{-3} following Onishi et al. (2015). The time step Δt was determined to satisfy both the convective and viscous CFL constraints for stability:

$$\mathrm{CFL_{conv}} = \max \frac{\|U\|\Delta t}{\Delta} \leq C_1, \tag{3}$$

$$\mathrm{CFL_{vis}} = \frac{\nu \Delta t}{\Delta^2} \leq C_2, \tag{4}$$

85 where $C_1 = C_2 = 0.3$. Onishi et al. (2013) demonstrated that the present finite-difference model employing these numerical methods maintains sufficient accuracy compared to spectral methods. The computational domain is a cube of side $2\pi L_0$ with periodic boundary conditions applied in all directions. The grid spacing is $\Delta = 2\pi L_0/N$, and the bulk Reynolds number is defined as $\mathrm{Re} = U_0 L_0/\nu$, where U_0 is the reference velocity and L_0 is the reference length. Three-dimensional domain decomposition was used for inter-process parallelization. The decomposition was chosen so that each subdomain is as close to
90 cubic as possible, minimizing the surface-area-to-volume ratio and reducing communication cost.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

2.2 Particle Phase

Particle motion obeys the following system of ordinary differential equations:

$$\frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}t} = \boldsymbol{V}, \quad (5)$$

$$\frac{\mathrm{d}\boldsymbol{V}}{\mathrm{d}t} = -\frac{f}{\tau_\mathrm{p}}\left(\boldsymbol{V} - U(\boldsymbol{X},t)\right), \quad (6)$$

95 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. $\tau_\mathrm{p} = \frac{2}{9}(\rho_\mathrm{p}/\rho)(r^2/\nu)$ is the relaxation time of a particle with radius r and density ρ_p. The particle-to-fluid density ratio ρ_p/ρ was set to 8.43×10^2, corresponding to water under atmospheric conditions (1013 hPa, 298 K). The non-linear drag correction factor f and the particle Reynolds number $\mathrm{Re_p}$ are defined as follows (Rowe and Henwood, 1961):

$$f = 1 + 0.15\,\mathrm{Re_p}^{0.687}, \quad (7)$$

100 $$\mathrm{Re_p} = \frac{2r|\boldsymbol{V} - U(\boldsymbol{X})|}{\nu}. \quad (8)$$

The fluid velocity $U(\boldsymbol{X})$ at the particle position $\boldsymbol{X}$ was computed by trilinear interpolation from the eight surrounding cell-center values. Time integration was performed using a two-stage, second-order Runge–Kutta method.

2.3 Implementation and Optimization in Julia

2.3.1 Performance Portability: Single Source, Multi-Platform Strategy

105 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, abstraction functions at an even higher level than 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. An explicit data movement design was also adopted. Compared to directive-based GPU porting
110 approaches such as OpenACC, 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 problem size, `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., CPU, CUDA, AMD, Metal, OpenAPI, etc.).

(a)

```
1: Parallel.foraxes(
2:   backend, (2:N-1, 2:N-1, 2:N-1)
3: ) do i, j, k
4:   dUdt[i, j, k] = - (U[i+1, j, k] - U[i-1, j, k]) / (2 * dx)
5: end
```
115

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

(b)

```
1: do concurrent (k = 2:N-1, j = 2:N-1, i = 2:N-1)
2:   dUdt(i, j, k) = - (U(i+1, j, k) - U(i-1, j, k)) / (2.0 * dx)
3: 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). 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, dynamic communication is therefore required.

In the original LCS (Fortran implementation), particle communication was implemented sequentially. 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 makes the scheme incompatible with GPU environments where multiple threads update the same counter concurrently.

In LCS.jl, a GPU-native particle communication algorithm based on prefix-scan was implemented. 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 GPU threads.

2. **Prefix-scan**: The cumulative sum (prefix-scan) of the mask array is computed to determine the storage position of each particle in the send buffer in one pass. The storage position of particle i is given by $\mathrm{scan}[i] = \sum_{j=1}^{i} \mathrm{mask}[j]$.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

3. **Parallel packing**: For particles with mask $= 1$, each particle is written in parallel by GPU threads to the position indicated by $\mathrm{scan}[i]$. Particles with mask $= 0$ are excluded from writing, so no threads overwrite 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:

155 $$\mathrm{mask}: \quad 0,\ 1,\ 0,\ 1,\ 0$$
$$\mathrm{scan}: \quad 0,\ 1,\ 1,\ 2,\ 2$$

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, while the next particle cannot be processed until the previous one is complete. In the present prefix-scan implementation, both position
160 determination and packing were executed in parallel by GPU threads. This eliminates the sequential dependency and enables efficient GPU-native particle communication.

3 Results

3.1 Validation

3.1.1 Flow Phase

165 Table 1 shows the statistically-steady state statistics obtained from GPU computations using LCS.jl. Computations were performed up to $N^3 = 2048^3$. All cases satisfy $k_{\max}\eta \approx 2$, where $k_{\max} = N/2$ is the maximum wavenumber and η is the Kolmogorov length scale, ensuring sufficient spatial resolution down to the Kolmogorov scale. The obtained statistics, including root-mean-square velocity u_{rms}, Taylor-scale based Reynolds number Re_λ, skewness of velocity derivative $-S$, and flatness of velocity derivative F, agreed well with the prior study (Onishi et al., 2011). Figure 1 shows kinetic energy spectra $E(k)$,
170 defined as $E(k) = \frac{1}{2}\sum_k' |\hat{u}(\boldsymbol{k})|^2$, where $\hat{\ }$ denotes the Fourier transform, $\sum_k' \equiv \frac{4\pi k^2}{N_k}\sum_{k-1/2\leq|\boldsymbol{k}|<k+1/2}$, and N_k is the num-

Table 1. Flow statistics in the statistically-steady state at different resolutions. $-S$ denotes the skewness of the velocity gradient; F denotes the flatness.

N^3	u_{rms}	Re_λ	$k_{\max}l_\eta$	$-S$	F
128^3	0.966	79.3	2.09	0.482	4.82
256^3	0.993	127	2.05	0.488	5.45
512^3	1.03	209	2.02	0.564	6.58
1024^3	1.00	333	2.14	0.570	7.47
2048^3	0.999	536	2.16	0.604	9.02

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

ber of wavevectors $\boldsymbol{k}$ satisfying $k-1/2 \leq |\boldsymbol{k}| < k+1/2$. For the largest case with $\mathrm{Re}_\lambda \approx 536$, an inertial subrange spanning two decades is confirmed, 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

175 Figure 2 shows the dependence of the radial distribution function at contact $g(r = R)$ on Stokes number St and Re_λ, where St is the ratio of particle relaxation time τ_p to Kolmogorov timescale. The quantity $g(r = R)$ quantifies the clustering intensity, where unity indicates a uniform distribution. This quantity is used in the calculation of collision kernels in cloud microphysics. The results agreed well with the prior study (Onishi and Seifert, 2016) across all St and Re_λ conditions. This confirmed that LCS.jl quantitatively reproduced the St and Re_λ dependence of particle clustering in turbulence, validating the particle tracking
180 implementation.

3.2 Computational Performance

Computational performance was evaluated on TSUBAME4.0 at the Institute of Science Tokyo. Table 2 shows the node configuration.

3.2.1 HALO Communication Optimization

185 Figure 3 shows the speedup for HALO communication optimizations with respect to the number of GPUs. Measurements were performed for grid size 1500^3 and particle count 750^3, ranging from 8 to 64 GPUs. The speedup was defined as the ratio of execution time relative to the baseline with no optimization applied. The combined use of the time-blocking method and communication–computation overlap yielded the highest speedup of $1.34\times$ at 64 GPUs. These results confirmed the performance improvement obtained by communication optimization.

Table 2. TSUBAME4.0 node configuration.

Component	Specification
CPU	AMD EPYC 9654 2.4 GHz $\times$ 2 Socket
Cores / Threads	96 cores / 192 threads $\times$ 2 Socket
Memory	768 GiB (DDR5-4800)
GPU	NVIDIA H100 SXM5 94 GB HBM2e $\times$ 4
CPU–GPU Interconnect	PCIe Gen5 $\times$16 (64 GB/s one-way)
SSD	1.92 TB NVMe U.2 SSD
Network Interconnect	InfiniBand NDR200 200 Gbps $\times$ 4

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

190 ### 3.2.2 Particle Communication Optimization Based on Prefix-Scan

Table 3 shows the particle communication cost for two implementations: a GPU-native implementation based on prefix scan and a CPU-delegated sequential implementation. 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
195 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

$$17\times 8\,\mathrm{B}\times 5.3\times 10^7 \approx 7.2\,\mathrm{GB}. \tag{9}$$

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

200 $$t_\mathrm{D2H} = t_\mathrm{H2D} = 7.2\,\mathrm{GB}/64\,\mathrm{GB/s} \approx 113\,\mathrm{ms}. \tag{10}$$

Sequential packing on the CPU was limited by the single-thread 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

$$t_\mathrm{pack} = t_\mathrm{unpack} = 7.2\,\mathrm{GB}/41\,\mathrm{GB/s} \approx 176\,\mathrm{ms}. \tag{11}$$

205 In the GPU-native implementation, D2H/H2D transfer of the full particle arrays is not required. The measured packing and unpacking times were 12 ms and 60 ms, respectively. The communication cost per RK2 stage in the GPU-native implementation based on prefix-scan was $12\,\mathrm{ms} + 60\,\mathrm{ms} = 72\,\mathrm{ms}$, which was approximately $578\,\mathrm{ms}/72\,\mathrm{ms} \approx 8.0$ times faster than the theoretical lower bound of the CPU-delegated approach, $(113\,\mathrm{ms} + 176\,\mathrm{ms})\times 2 = 578\,\mathrm{ms}$. The higher unpacking cost (60 ms vs. 12 ms) is attributed to the compaction step, in which invalid particles are moved to the end of the array via intermediate
210 buffer copies in GPU memory. Note that this estimate excluded the computational cost of packing itself; the actual cost of CPU delegation is therefore even higher.

The impact of this communication speedup on overall performance was evaluated. The wall-clock time per time-integration step under the measurement conditions of 8 GPUs, $N^3 = 1500^3$, and $N_\mathrm{p} = 750^3$ was 1490 ms (Sect. 3.2.4). With CPU delegation, the particle communication cost for the two RK2 stages accounted for approximately 78% of the total execution
215 time ($578\,\mathrm{ms}\times 2/1490\,\mathrm{ms} \approx 0.78$). The GPU-native implementation reduced this fraction to approximately 10% ($0.78/8.0 \approx 0.098$).

This reduction eliminates the communication bottleneck that would otherwise negate the computational advantage of GPU execution, enabling effective utilization of GPU performance in large-scale multiphase flow simulations.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

Table 3. Particle communication cost per RK2 stage for the GPU-native (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 $\times16$ one-way bandwidth $\sim$64 GB/s and EPYC 9654 single-core bandwidth $\sim$41 GB/s.

Phase	GPU-native (measured)	CPU-delegated sequential (estimated)
D2H transfer (full particle arrays)	—	$\sim$113 ms
Packing	12 ms	$\sim$176 ms
Unpacking	60 ms	$\sim$176 ms
H2D transfer (full particle arrays)	—	$\sim$113 ms
Total (per stage)	**72 ms**	**$\sim$578 ms**

3.2.3 Julia vs Fortran Comparison

220 Figure 4 shows wall-clock time per time-integration step as a function of the number of CPU processes for LCS.jl (Julia) and the Fortran implementation. Computations were performed with grid size $N^3 = 256^3$ and particle count $N_\mathrm{p} = 128^3$, with identical computational settings in both implementations. Execution time was measured as a function of the number of processes to evaluate parallel performance.

At 1, 2, and 4 processes, LCS.jl showed an execution time up to approximately 10% longer than the Fortran implementation.
225 At 8 processes, both implementations exhibited comparable execution times.

At small process counts, the contribution of communication cost was small, and differences in raw computational performance were dominant. LCS.jl uses identical kernel code for both CPU and GPU. 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 prevent equivalent optimizations from being applied in full. This
230 difference manifests as a performance gap of up to approximately 10% at small process counts. However, as the number of processes increases, where the contribution of communication cost grows, the performance became comparable to the Fortran implementation at 8 processes due to communication optimizations applied (Sect. 3.2.1).

These results confirmed that LCS.jl achieves computational performance equivalent to the Fortran implementation in computations using many processes. This demonstrated that the single-source and multi-platform design does not sacrifice compu-
235 tational performance.

3.2.4 Scaling

Scaling measurements were performed on TSUBAME4.0 using up to 64 nodes. The baseline problem size was set to 750^3 grids and 325^3 particles, which is the maximum problem size that fits within a single GPU memory. The particle count was set to range from 1/10 to 10 times the grid count, which is the range used in typical cloud microphysics computations. Perfor-

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

240 mance evaluation was conducted independently of physical statistical analysis. To isolate scaling performance from physical transients, the number of Poisson iterations was fixed at 6, which is the typical value observed at statistically steady state.

Julia employs just-in-time (JIT) compilation, which compiles code on its first execution. To exclude the effect of JIT compilation, the first 10 steps were treated as warmup and excluded from the measurements.

To compare GPU and CPU under identical conditions, one device (1 CPU / 1 GPU) was assigned per process. However, 245 for CPUs, assigning 1 process to 1 CPU (96 threads) is not optimal. In memory-bound computations, a high thread count reduces memory bandwidth efficiency. To determine the optimal balance, performance measurements were conducted with the total thread count fixed at 384 (4 CPUs × 96 threads) with $N^3 = 1500^3$ grids and $N_\mathrm{p} = 750^3$ particles, while the number of threads per process was varied. A maximum speedup of 2.7× was achieved at 4 threads per process over the 1 CPU (96 thread) per process baseline. This optimization efficiency of 2.7 will be used to estimate the effective performance on CPUs when 250 comparing performances on CPUs and GPUs.

Figure 5 shows strong scaling results for GPU and CPU. Total problem sizes of $N^3 = 1500^3$ and 3000^3 grids were used. Since TSUBAME4.0 has 4 GPUs and 2 CPUs per node, scaling was measured up to 256 GPUs and 128 CPUs due to node count constraints. Strong scaling efficiency is defined as $E_\mathrm{s} = T_0/(NT_N)$, where T_0 is the execution time at the baseline device count and T_N is the execution time at N devices. The baseline device count was 8 GPUs / 4 CPUs for $N^3 = 1500^3$, and 64 255 GPUs / 32 CPUs for $N^3 = 3000^3$.

Figure 5 confirms that strong scaling efficiency was maintained above 85% for GPUs (up to 256) and above 70% for CPUs (up to 128). The lower strong scaling efficiency of CPUs (70%) compared to GPUs can be attributed to memory bandwidth contention among threads within a single CPU socket, not to a limitation of the multi-platform design. This contention can be mitigated by adjusting the process and process-thread balance.

260 Figure 6 shows weak scaling results for GPU and CPU. The problem size per device was set to $N^3 = 750^3$ grids. Since TSUBAME4.0 has 4 GPUs and 2 CPUs per node, scaling was measured up to 216 GPUs and 108 CPUs due to node count constraints. Weak scaling efficiency is defined as $E_\mathrm{w} = T_0/T_N$, where T_0 is the execution time at the baseline device count and T_N is the execution time at N devices. For the scaling measurements on CPUs, 8 CPUs were used as the baseline. For the scaling measurements on GPUs, the baseline device count was 8 for power-of-two counts (8 and 64) and 27 for non-power-of-265 two counts (27, 125, and 216).

The execution time of GPUs showed two distinct trends depending on whether the device count was a power of two. Power-of-two counts achieved approximately 25% shorter execution time than non-power-of-two counts. This difference can be attributed to the node assignment of processes. Each node contains 4 GPUs. Therefore, GPUs 1–4, 5–8, 9–12, and so on are assigned to the same node. Figure 7 illustrates the difference in inter-node communication patterns between a power-of-two 270 and a non-power-of-two GPU configuration. In the non-power-of-two case (Fig. 7a), certain processes—such as GPU 9—have neighbors across node boundaries in all directions, forcing all three communications to be inter-node. In the power-of-two case (Fig. 7b), some directions of communication remain intra-node. Since execution time is bottlenecked by the slowest process, configurations in which all three communications are inter-node result in a significant performance penalty. Therefore, power-of-two counts used 8 GPUs as the baseline, and non-power-of-two counts used 27 GPUs.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

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 time is estimated by multiplying the measured optimization efficiency of 2.7 (see Sect. 3.2.4).

N^3	N_p	Nodes	GPUs	GPU Time [s]	CPUs	CPU Time [s] (Optimized)	Speedup
1500^3	750^3	2	8	1.491	4	20.4	13.7×
1500^3	750^3	4	16	0.779	8	12.5	16.0×
1500^3	750^3	8	32	0.439	16	7.19	16.4×
1500^3	750^3	16	64	0.249	32	4.47	18.0×
3000^3	1500^3	16	64	1.657	32	23.3	14.1×
3000^3	1500^3	32	128	0.826	64	13.0	15.7×
3000^3	1500^3	64	256	0.484	128	7.50	15.5×

Table 5. 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	—	80.0	—
Heterogeneous	32	1	22.3	0.72

275 Figure 6 confirms that weak scaling efficiency was maintained above 90% for GPUs (up to 216) for both power-of-two and non-power-of-two configurations, and above 95% for CPUs (up to 108).

3.2.5 GPU and CPU Performance Comparison

Figure 4 shows the performance comparison between GPU and CPU executions using the same number of nodes on TSUBAME4.0. Since each node has 4 GPUs and 2 CPUs, the GPU and CPU counts differ. Because the process and thread balance
280 optimization was not applied during the scaling experiments, the CPU execution time was estimated by multiplying the measured optimization efficiency of 2.7 (Sect. 3.2.4).

LCS.jl achieved a maximum speedup of $18.0\times$ on GPUs over CPUs. This demonstrated that the single-source and multi-platform design can achieve an acceleration ratio comparable to GPU-dedicated implementations.

3.2.6 Heterogeneous Execution

285 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 in which time-stepping runs on CPUs and particle statistics computation runs on a GPU.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

Computation was performed on a workstation, assuming an environment in which a GPU is installed as an auxiliary device. The CPU was 2 × Intel Xeon Gold 6326 and the GPU was an NVIDIA RTX A6000 (48 GB VRAM); this configuration represents a typical computing environment in which assigning one high-performance GPU per CPU is not feasible. Consequently, the working dataset often exceeds GPU memory capacity, resulting in degraded performance.

Table 5 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 72%. 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 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.

The particle communication cost in the GPU-native implementation based on prefix-scan was 8.0 times faster than the theoretical lower bound of the CPU-delegated approach. The GPU-native implementation reduced the particle communication cost from approximately 78% (CPU-delegated) to 10% of total execution time. This reduction eliminated the communication bottleneck that would otherwise negate the computational advantage of GPU execution, enabling effective utilization of GPU performance in large-scale multiphase flow simulations.

We confirmed that the present LCS.jl achieves computational performance equivalent to the Fortran implementation in computations using many processes. This demonstrated that the single-source and multi-platform design does not sacrifice computational performance. The performance results confirmed that large-scale DNS computations can be performed efficiently on any platform. For example, for GPUs, strong scaling efficiency was maintained above 85% (up to 256 GPUs) and weak scaling efficiency above 90% (up to 216 GPUs). For CPUs, strong scaling efficiency was maintained above 70% (up to 128 CPUs), and weak scaling efficiency above 95% (up to 108 CPUs).

LCS.jl achieved a maximum speedup of $18.0\times$ on GPUs over CPUs. This demonstrated that the single-source and multi-platform design achieves an acceleration ratio comparable to GPU-dedicated implementations. A trial heterogeneous execution, delegating only statistics computation to a GPU, achieved a 72% 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.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

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.

325 *Code availability.* The current 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 used in this paper is archived on Zenodo under DOI https://doi.org/10.5281/zenodo.19515891, as are configuration files and scripts to run the model and produce the plots for all the simulations presented in this paper.

Author contributions. T. Tominaga developed the model code, performed the simulations, and prepared the manuscript. R. Onishi supervised
330 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 Tokyo Institute of Technology 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).
335 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.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

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.

340 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.

345 Churavy, V.: KernelAbstractions.jl, https://doi.org/10.5281/zenodo.4021259, 2020.

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.

350 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.

355 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 360 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.

365 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, 370 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.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

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.

375 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.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

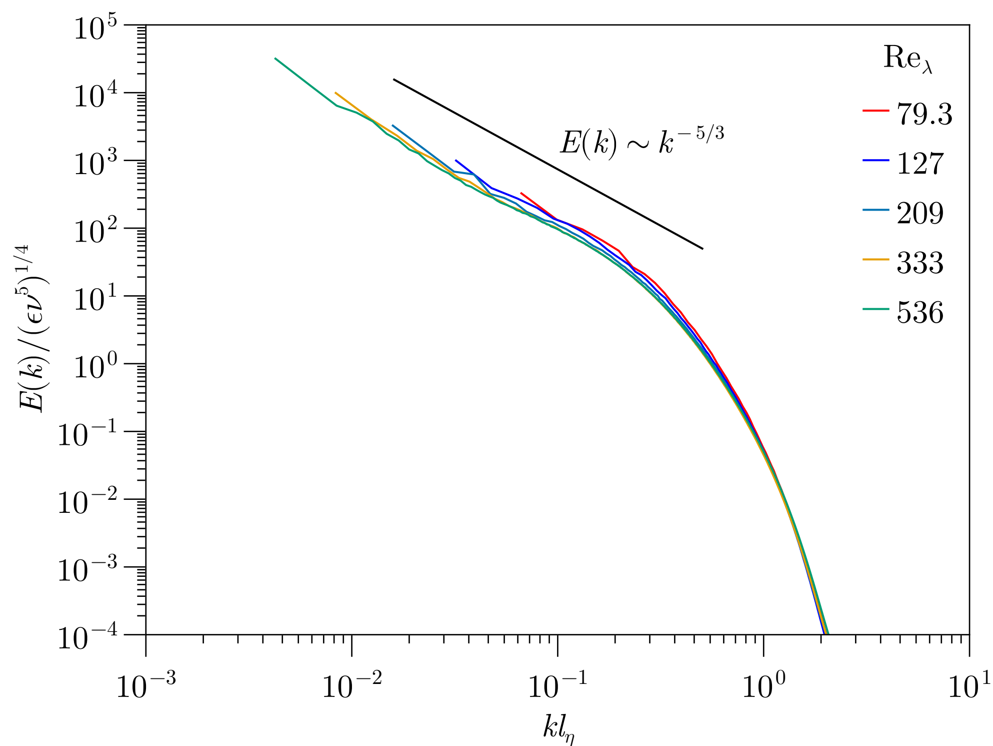


Figure 1. Energy spectra $E(k)$ normalized by the Kolmogorov velocity scale $(\varepsilon\nu^5)^{1/4}$, where ε is the mean energy dissipation rate and ν is the kinematic viscosity, as a function of the normalized wavenumber kl_η, where $l_\eta = (\nu^3/\varepsilon)^{1/4}$ is the Kolmogorov length scale. Results are shown for resolutions $N^3 = 128^3$ through 2048^3 ($\mathrm{Re}_\lambda \approx 79.3$ to 536). The dashed line indicates the Kolmogorov $k^{-5/3}$ scaling.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

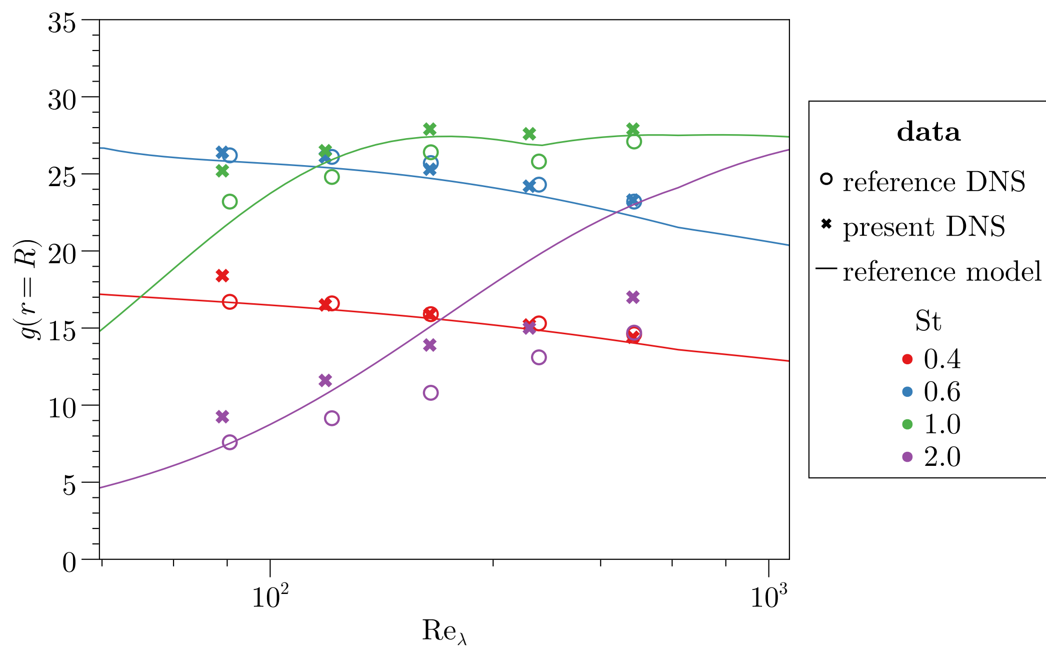


Figure 2. Radial distribution function at contact $g(r = R)$ as a function of the Taylor-microscale Reynolds number Re_λ. Crosses denote the present results; open circles denote the corresponding reference results of Onishi and Seifert (2016) for $\mathrm{St} = 0.4$, 0.6, 1.0, and 2.0. Solid lines denote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

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

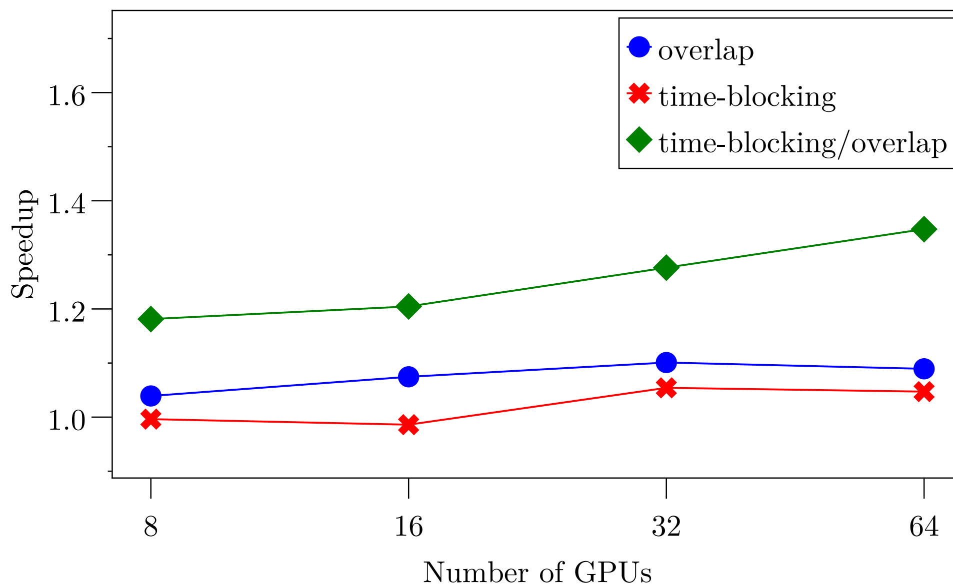


Figure 3. Speedup of wall-clock time per time-integration step as a function of the number of GPUs ($N^3 = 1500^3$, $N_\mathrm{p} = 750^3$), for three HALO communication optimizations: communication–computation overlap, time-blocking, and their combination. The speedup is defined as the ratio of execution time relative to the baseline with no optimization applied.

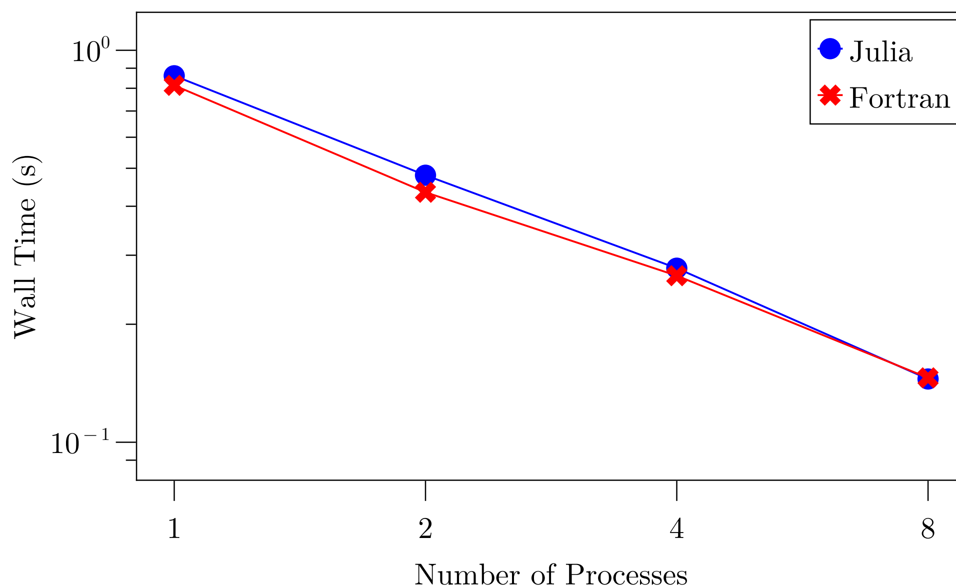
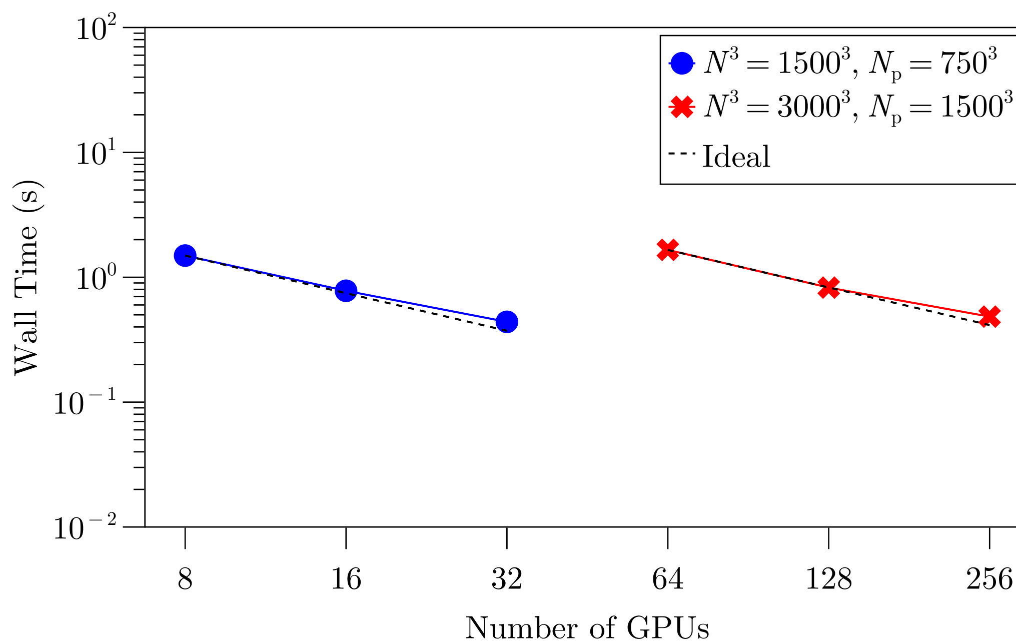
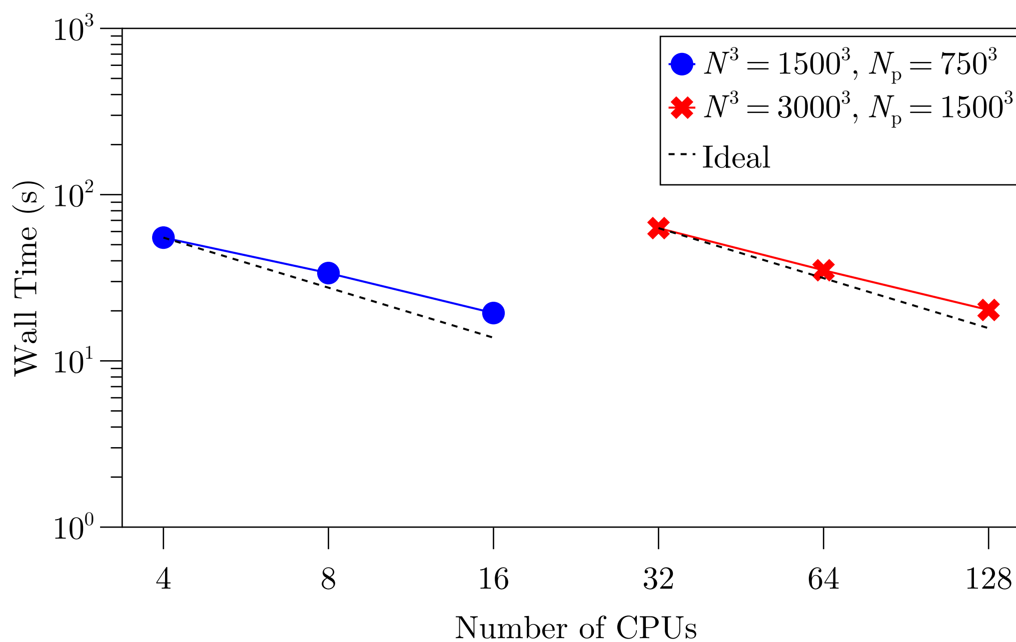


Figure 4. Wall-clock time per time-integration step as a function of the number of CPU processes for LCS.jl (Julia) and the Fortran implementation ($N^3 = 256^3$, $N_\mathrm{p} = 128^3$).

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.



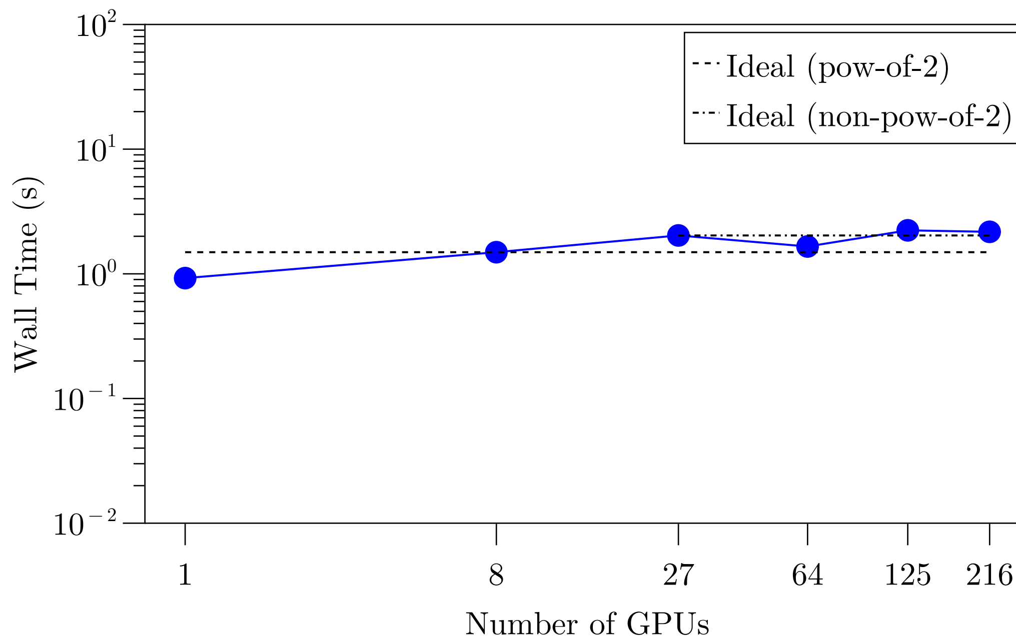
(a)



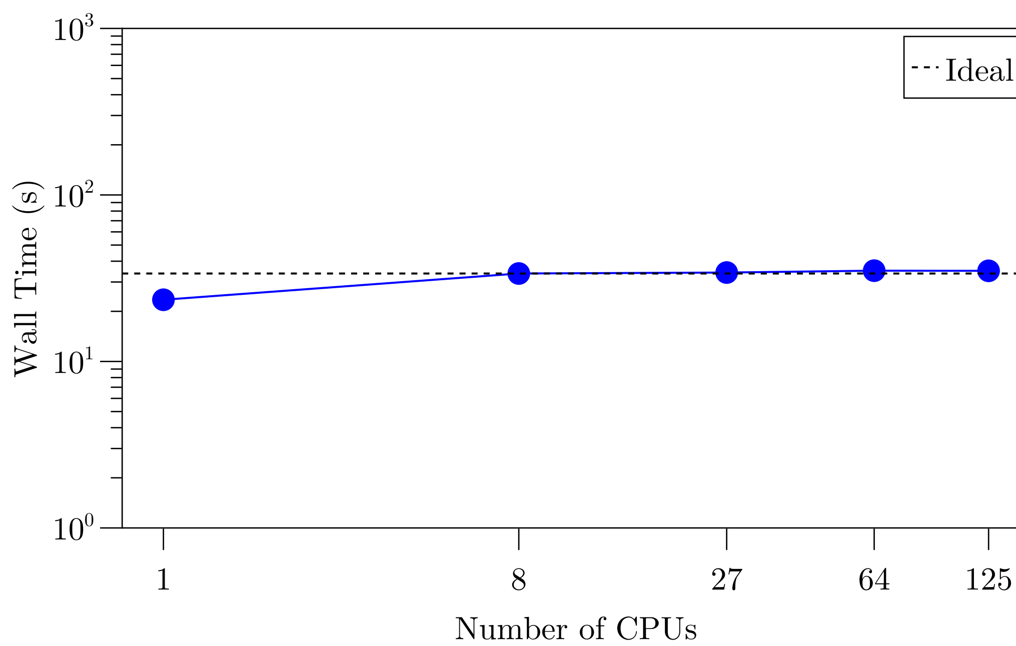
(b)

Figure 5. Wall-clock time per time-integration step as a function of the number of devices for fixed total problem sizes. (a) GPU execution; (b) CPU execution. The black line indicates ideal strong scaling.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.



(a)



(b)

Figure 6. Wall-clock time per time-integration step as a function of the number of devices for a fixed problem size per device. (a) GPU execution; (b) CPU execution. The black horizontal lines indicate ideal weak scaling.

https://doi.org/10.5194/egusphere-2026-2214
Preprint. Discussion started: 8 May 2026
© Author(s) 2026. CC BY 4.0 License.

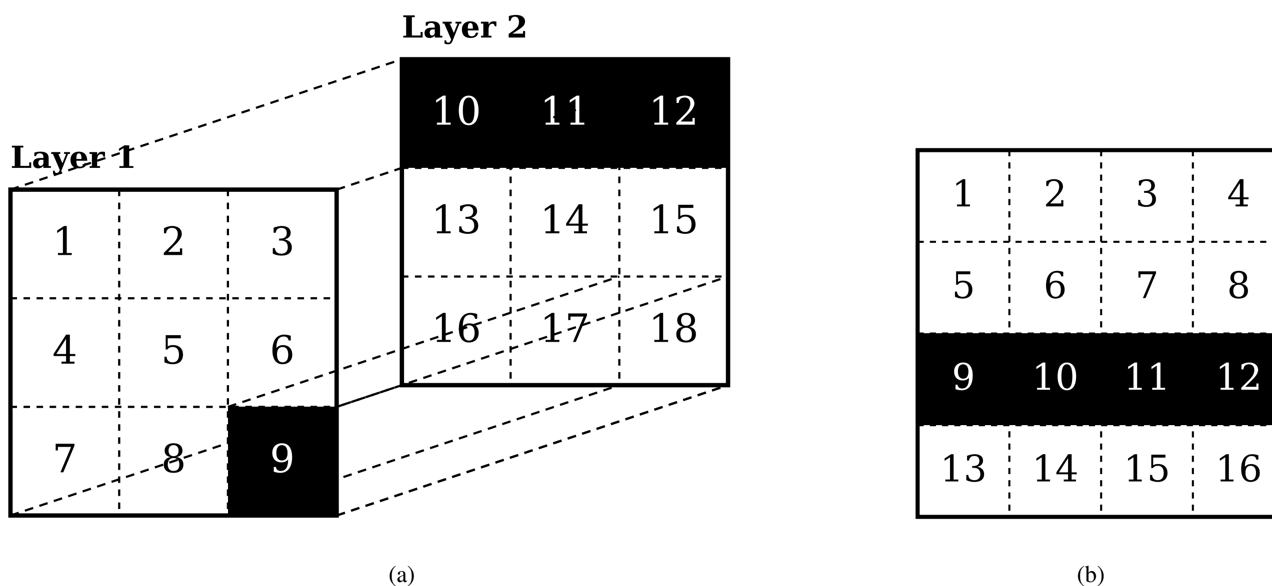


Figure 7. Communication patterns for (a) a non-power-of-two $3 \times 3 \times 3$ GPU topology and (b) a power-of-two $4 \times 4 \times 4$ GPU topology. GPUs within the same node are shown in black. In (a), certain GPUs such as GPU 9 have no intra-node neighbors in any direction. In (b), at least one direction of communication remains intra-node for all GPUs.